THE EFFICIENT EVALUATION OF VISUAL QUERIES WITHIN A LOGIC-BASED FRAMEWORK

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE,
FACULTY OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN
IN FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Anish Maharaj
1995

Supervised by
Associate Professor P. T. Wood

Abstract

There has been much research in the area of visual query systems in recent years. This has stemmed from the need for a more powerful database visualization and querying ability. In addition, there has been a pressing need for a more intuitive interface for the non-expert user. Systems such as Hy^+, developed at the University of Toronto, provide environments that satisfy a wide range of database interaction and querying, with the advantage of maintaining a visual interface abstraction throughout.

This thesis explores issues related to the translation and evaluation of visual queries, including semantic and optimization possibilities. The primary focus will be on the GraphLog query language, defined in the context of the Hy^+ visualization system. GraphLog is translated to the deductive database language Datalog, which is subsequently evaluated by the CORAL logic database system.

We propose graph semantics, which define the meaning of visual queries in terms of paths in a graph, for monotone GraphLog. This provides a more intuitive meaning which is not linked to any particular translation. Therefore, Datalog generated by a translation may be compared to well-defined semantics to ensure that the translation preserves the intended meaning. By examining various queries in terms of the graph semantics, we uncover a shortcoming in the existing GraphLog translation.

In addition, an alternative translation to Datalog, based on the construction of a nondeterministic finite state automaton, is described for GraphLog queries. The translation has the property that visual queries containing constants are optimized using a technique known as factoring. In addition, the translation performs an optimization on queries with multiple edges that contain no constants, referred to here as variable constraining.

The incorporation of these optimizations into the translation results in significant performance improvements when evaluating certain visual queries. This contrasts with the existing GraphLog translation, which relies on CORAL to choose and perform optimizations.

Acknowledgements

Various people need to be thanked for their assistance during the course of my research.

Firstly, I would like to thank my supervisor, Peter Wood, for his interest and willingness to advise on problems and issues. His patience and unaffected style have allowed me to appreciate the field of Computer Science, without taking away my enthusiasm.

My colleagues in the Database Laboratory have been very co-operative and supportive during my research. I am very grateful to Andrew Luppnow whose discussions have given me the encouragement and perspective required.

For financial support, I would like to thank the CSIR Foundation for Research Development.

I would also like to thank those in my family who have shown me understanding and support during my studies.

Most importantly, I would like to thank Meenakshe, who has helped me turn the seemingly impossible into reality.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The increasing computational demands of users, in conjunction with the increasing power of computer systems and volume of data available, has resulted in the development of interactive scientific visualization systems.

These systems allow the user to manipulate, visualize and query data in order to determine interesting features. An important aspect of this process is the speed at which such analysis is performed, which dictates the degree to which the user may interact with the system. Another consideration in the design of such a system is the maintenance of a consistent visual abstraction and method of interaction to allow intuitive and easy use.

Large volumes of data have traditionally been stored on database systems, which also allow the retrieval and querying of this data. However, various inadequacies have been identified in the established relational data model, particularly with respect to the power of queries based on relational algebra. The search for more powerful query languages has resulted in the development of logic database systems, which incorporate aspects of logic programming. These systems make use of logic-based data models and query languages, whose expressive power surpasses that of relational algebra.

The field of logic query languages has been described as "deductive databases" as well as "knowledge-bases", with reference to the attempt to integrate database technology with artificial intelligence technology [Ull88]. Much work has been concentrated on the logic query language, Datalog, and its derivatives. Some of the drawbacks of data manipulation and querying using Datalog include the considerable expertise required from the user, as well as potentially poor performance.

The Hy^+ system [CM93] has been developed by a team of researchers at the University

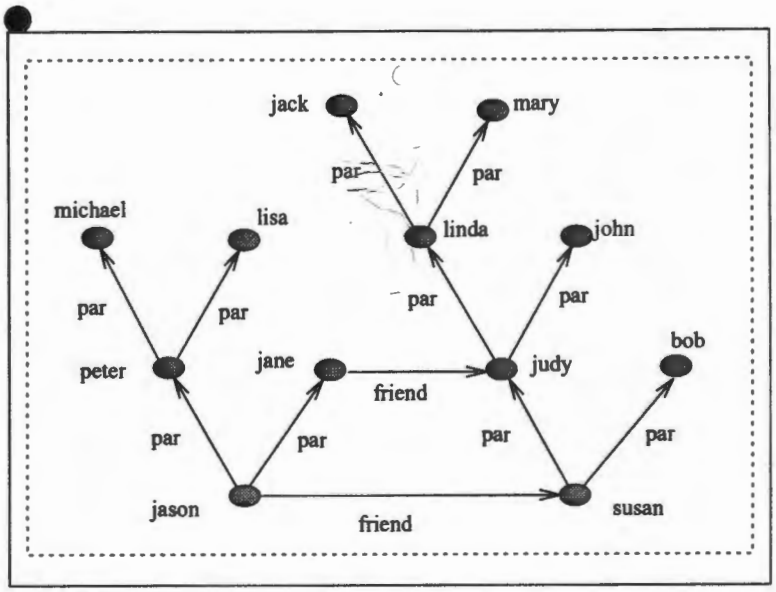

Figure 1.1: Parents and Friends Database Graph

of Toronto. It addresses the need to capture the power and robustness of an established database system while trying to manipulate and visualize data quickly and easily.

In Hy$^+$, relationships between data are represented using directed graph structures. An example is shown in Figure 1.1, which depicts the parents and friends of various people. Graphs are a natural way of representing such data, and appeal to the intuition of the user. Also, the querying of such graphs can be simply understood in terms of finding and displaying paths in the graph that satisfy various criteria.

An illustrative example of a GraphLog query is presented in Figure 1.2 which provides some perspective on the layout and format of queries. The exact details of the notation and meaning of this query will be examined more closely in the next chapter. The query is visually equivalent to finding all ancestors of every person in the sample database in Figure 1.1.

The Hy$^+$ system runs in a Smalltalk environment and incorporates a number of features required for complete and flexible database visualization. These include an interactive visual interface (the browser), querying capability, graph-layout facilities and data-filtering tools.

The focus of this thesis is primarily on the query translation and evaluation component of Hy$^+$. Visual queries, formulated using the GraphLog query language [Con89], are translated

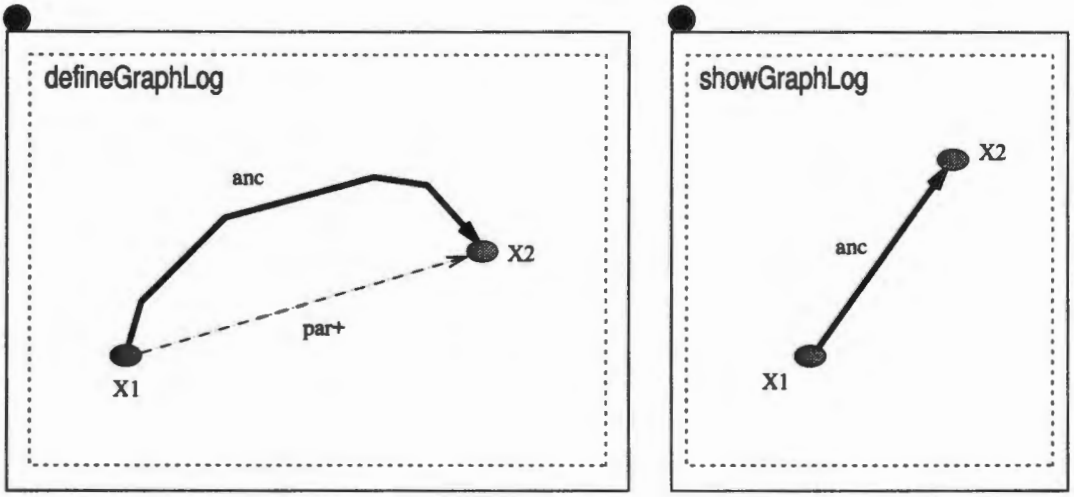

Figure 1.2: Ancestor Query in GraphLog

to Datalog, which is evaluated using a back-end logical database system, which by default is the CORAL system [RSS92] developed at the University of Wisconsin-Madison.

The aims of my research involve the examination of various aspects of the translation of GraphLog, both in terms of the meaning of visual queries and the efficiency of their evaluation.

1.1 Outline of the Thesis

The background required for the rest of the thesis is provided in Chapter 2. An overview of Datalog is presented along with a description of the CORAL logic database system. In addition, various optimization techniques relevant to the thesis are described. These encompass the most well known and general methods proposed. The GraphLog query language and its translation to Datalog are described, along with some background to the related language, GRE[Woo90], from which the alternative translation has evolved.

Since the visual abstractions used to represent queries and data in Hy^+ are evaluated using Datalog, it is important to be able to define formally the semantics of such queries in order to obtain consistency when answers are produced, and to ensure that the corresponding Datalog program always produces results which are intuitively expected. Current GraphLog semantics are defined purely in terms of the semantics of the Datalog program produced

by the translation. In a sense, this subverts the intention of GraphLog to be understood independently of Datalog, especially since the translation may not necessarily maintain the intuitive understanding of the query.

One of the features of the GRE query language proposed in [Woo90] is the definition of semantics which are independent of the translation described for the visual queries. As a result, the only acceptable translation to Datalog is one which preserves these semantics. This ensures consistency and predictability when formulating queries in the language. In Chapter 3, we define a semantics for GraphLog based on finding paths in graphs by modifying and extending the semantics of GRE. As a result, we uncover an inconsistency in the Datalog semantics of certain queries produced by the standard translation in Hy^+.

A number of techniques have been developed for the optimization of Datalog programs, including program rewriting techniques to generate a more efficient form for various classes of queries. Studies have indicated that these techniques can be extremely effective in reducing the amount of computation and therefore the time required to produce answers to a query [BR86]. Since one of the objectives of GraphLog is efficient evaluation of queries, and being part of an interactive system, it is important to try and minimize query processing time by making use of suitable optimizations.

The current translation to Datalog defined for GraphLog is based on the structure of the queries and the regular expressions used in the queries. No attempt is made to optimize the resultant logic query during this translation, but this is instead left to the logic database system which evaluates the query.

The proposed translation, which is an extension of work done in [VW95] and [Woo90], is based on the structure of non-deterministic finite state automata (NFA) constructed from regular expressions present in the query. This translation performs an optimization known as factoring [NRSU89b] [NRSU89a], which propagates constants in the query into base relations. The translation has been extended to handle visual queries with multiple edges. Various other optimizations which also contribute to the efficiency of resultant Datalog program are also included in the extended translation. Details of this translation are given in Chapter 4.

There are certain advantages in attempting to optimize a query as part of the translation. One motivation for doing this concerns the restricted class of Datalog programs generated by the visual query translation, as compared to the generality of the queries that a logic database system is required to handle. It is reasonable to assume that optimization techniques for dealing with a more general class of queries often need to be more conservative

than for a well defined subset of queries.

This is evident in the strategies of existing systems, where even small changes to the format of a Datalog program suitable for optimization result in the program not being optimized, even though this is possible. Additionally, when optimizations are available for restricted classes of queries, these are usually not selected automatically by the database system. Instead, the user is expected to understand the details of the Datalog evaluation, as well as the various optimizations, which is far removed from some of the objectives of GraphLog.

While the inadequacies in selecting a suitable optimization method may partly be a practical limitation of the database system, the problem does exist and there are significant consequences for the user. Regarding optimization, the CORAL manual advises the user as follows:

> "CORAL is a high-level language, and the compiler attempts to optimize programs to ensure efficient evaluation. However, completely automatic optimization in a language this powerful can only be an ideal, and the compiler often selects a less than optimal evaluation strategy"
>
> [RSSS94, Page 118]
>
> "... our conclusion, after using CORAL extensively, is that a user must have some minimal understanding of how programs are evaluated in order to write efficient programs ... You *can* write programs that can be understood non-operationally, and often your programs will run just fine. But if your program runs slower than you'd like, you may have to understand the underlying evaluation method at least in broad terms in order to make it run efficiently."
>
> [RSSS94, Page 14]

Another important motivation for building optimizations into translation is improving the flexibility in the choice of the query evaluation system. This means that the resultant Datalog program is portable across a number of logic database systems without sacrificing efficiency of evaluation. Performance of queries is not dictated by the optimizing techniques which are available on a particular back-end.

In Chapter 5, we discuss the implementation of the EVOQ visual query translation system, which performs the translation described in Chapter 4. We introduce queries on a database that has been obtained from a Geographical Information System (GIS) containing information about features like roads and towns in South Africa. We also describe some query

examples from previous work [VW95] [CMVW95], which were used to generate performance results.

Performance tests and results for these queries translated using our method are presented. A comparison is made between these results and results for the standard translation method. In both cases, identical visual queries, databases and query evaluation systems are used, with only the translation performed being different. We find that there are significant performance benefits for certain queries when using our translation method.

Finally, in Chapter 6, we summarize the various threads and contributions that have been discussed in this thesis, and suggest topics for future research.

Chapter 2

Background

The visual query languages GraphLog [Con89] [CMV94] and GRE [Woo90] allow the expression of powerful visual queries on graph structures. GraphLog and GRE have many similarities. Both languages are descended from the G^+ query language proposed in [CMW88] [Cru87] and its predecessor, G [Woo88]. Queries in GraphLog and GRE use a syntax based on regular expressions in addition to a graph format. Also, translations to the deductive database language Datalog have been defined for both languages. However, one significant difference between these translations is that the GRE translation is based on the construction of an automaton from regular expressions in the query, whereas the standard GraphLog translation is based on the structure of the regular expressions.

An algorithm presented in [VW95] describes how certain queries expressed in GraphLog may be translated using a method based on a non-deterministic finite state automaton. It is argued that there are performance benefits of this translation for various classes of queries over the standard GraphLog translation. These translations will be referred to as the NFA-translation and the RE-translation respectively, as used in [VW95].

In this chapter, the terminology required for the rest of the thesis is introduced. The Hy^+ system and the GraphLog language are described next, followed by some details of GRE. The Datalog language is defined and various related issues considered. The details of the CORAL system which is used to evaluate Datalog, and various optimization techniques that are currently available in the system are discussed. The RE-translation to Datalog is then presented, followed by the definition of the NFA-translation, which is one of the focuses of this thesis.

2.1 Hy$^+$ and GraphLog

Hy$^+$ is a data visualization system which allows the representation and manipulation of large volumes of structured data. It is based on an abstraction called a *hygraph*, which has features of hypergraphs [Ber73] and higraphs [Har88].

GraphLog is a visual query language that is used by the Hy$^+$ system to formulate visual queries. It has been described as a graphical formalism for visual manipulations of database visualizations. A class of queries known as *define* queries allow new relationships to be defined. These queries are evaluated by first translating them from GraphLog to logic programs, which are then processed by a suitable backend system on an appropriate data set, which may also have been translated from a visual format. The results are then presented to the user as a subset of the data set that the query is applied to, in a similar visual format. The results may also be used as data for subsequent queries. This cyclical process of query formulation and evaluation is illustrated in Figure 2.1.

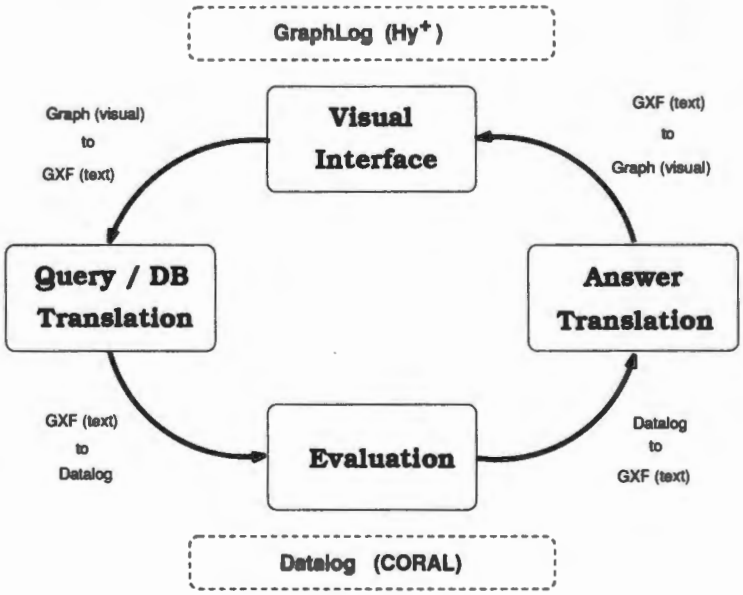

Figure 2.1: Hy$^+$ Query cycle and components

The original GraphLog language [Con89] has evolved with the development of the Hy$^+$ system to extend the meaning of queries with new visual formalisms such as *blobs* which depict containment, and another class of queries known as *filter* or *show* queries, which allow the selective filtering of the database without explicitly defining new relationships [CMV94].

At the moment, most of the additions are related to extending the visual formalisms used to represent data and queries, rather than increasing the power of queries in the system. For example, filter queries are implemented in Hy$^+$ by translating them to sets of define queries. Also, blobs may be defined purely in terms of sets of edges connected to a single node. However, the extensions do provide flexibility and allow the user to conceptualize relationships more naturally.

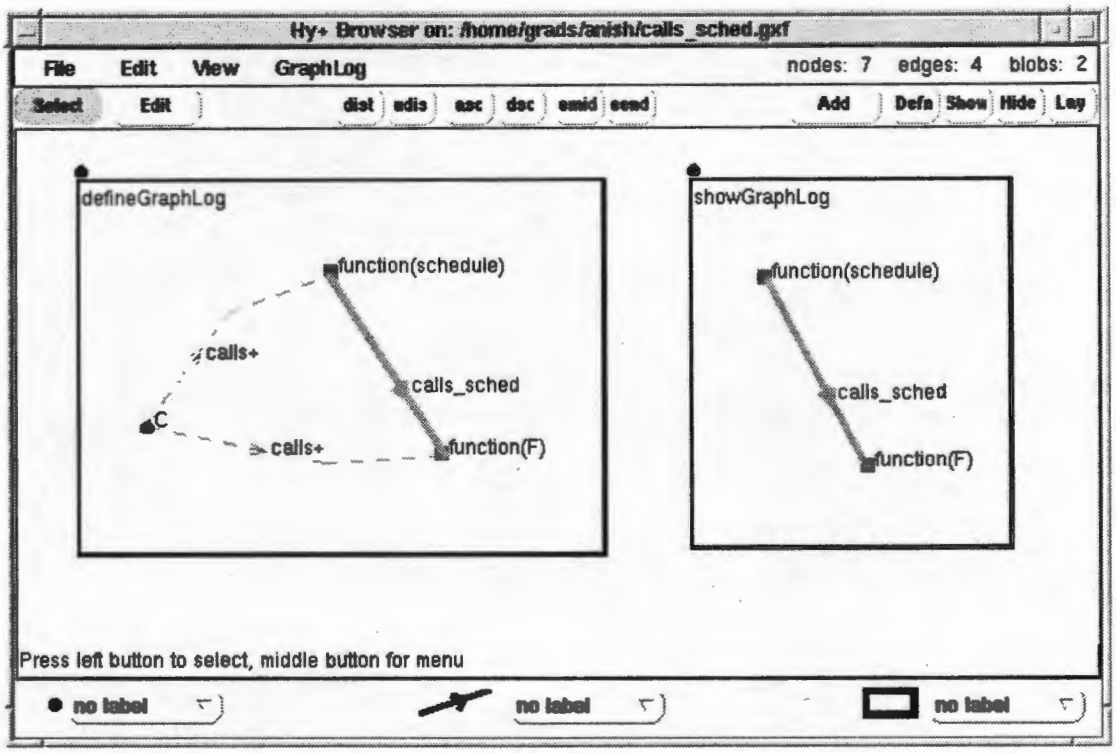

Figure 2.2: A Define and Show Query in Hy$^+$

Example 2.1 Figure 2.2 displays a screen from the Hy$^+$ system. The query **calls_sched** finds all functions F that are called (indirectly or directly) by a function C, which also calls (indirectly or directly) the function **schedule**. This is a form of the "common ancestor" query applied to function calls, and is similar to a query in [VW95].

■

In Hy$^+$, the visual interface component consists of a query browser that includes features such as tools to construct queries, as well as graph layout algorithms to format the data

[Noi93]. In order to translate and evaluate visual queries at a textual level, a format called the *Graph Exchange Format* (or *GXF*) has been specified [Eig93]. *GXF* is described as "a specification for a portable external representation of directed graphs and hygraphs". Queries in this format are translated to Datalog by the query translation component and are subsequently evaluated by a suitable database system. In this case, the CORAL system is chosen. These answers, which are generated as Datalog are then translated back to *GXF* which is then displayed at a visual level. As an example, the *GXF* query in Figure 2.2 is included in Appendix A.

The query translation that is proposed in this thesis is mostly based on the original GraphLog language and does not consider all of the Hy$^+$ extensions described above in detail. Nevertheless, the scope of visual queries even without using these extensions is significant. There is also the possibility of optimizing the programs produced by the translation of queries that make use of extensions such as filter queries.

2.1.1 Syntax

Definition 2.1 (from [Con94])

A *hygraph* H is a septuple

$$(N, L_N, \nu, L_E, E, L_B, B)$$

where: N is a finite set of *nodes*; L_N is a set of *node labels*; ν, the *node labelling function*, is a function from N to L_N that associates with each node in N a label from L_N; L_E is a set of *edge labels*; $E \subseteq N \times N \times L_E$ is a finite set of *labelled edges*; L_B is a set of *blob labels*; $B \subseteq N \times 2^N \times L_B$ is a finite set of *labelled blobs*.

A restriction is placed in the labelled blobs relation B to ensure that there is only one tuple (n, N, l) in B with the same values for the *container node* n and the blob label l.

The first and second component of an element e of set of edges E will be referred to as the *source node* and the *sink node* respectively.

For convenience, an *edge labelling function* λ, will be defined on the edge set E, such that $\lambda(e) \in L_E$ for $e \in E$. Then $e = (u, v, l) \in E$ iff $\lambda(e) = l$.

□

Definition 2.2 Let U be a set of constants and V be a set of variables, such that $U \cap V = \emptyset$.

A *term* is either a constant $c \in U$, a variable $v \in V$, an anonymous variable denoted _ (underscore), an aggregate function $f \in$ {MAX,MIN,SUM,AVG} applied to a variable, or a function f applied to a number of terms. The function f could be anonymous.

A *ground term* is either a constant or a function applied to a number of ground terms.

□

By convention adopted from Prolog, variables are denoted by strings of characters starting with an upper case letter and constants by strings of characters starting with a lower case letter, or numbers. The functions are strings of characters, starting with a lower case letter. The anonymous variable is used as a place-holder for variables that we want to project out.

Some of the terminology introduced for GraphLog is deliberately identical to terminology commonly used in logic programming and Datalog, described in the following section. Since one of the features of GraphLog is a translation to Datalog, and the evaluation of GraphLog queries uses Datalog, it is natural that there should be analogous concepts.

Definition 2.3 A *literal* is a label of the form $p(a_1, \ldots, a_n), n \geq 0$ (a *forward literal*), or alternatively, $-p(a_1, \ldots, a_n), n \geq 0$ (a *reverse literal*), where p is a predicate symbol, and each a_i is a term.

A *ground literal* is a literal where each a_i is a ground term.

□

Definition 2.4 (from [Con94])

Nodes are labelled by terms. Edge or blob labels are extended regular expressions, where the alphabet is the set of valid terms.

An *edge* or *blob label* is an expression generated by the following grammar, (where $\bar{T}$ is a sequence of terms and p is a predicate symbol) :

$$E \leftarrow E \mid E; E.E; \sim E; (E); E*; E+; p(\bar{T}); -p(\bar{T}); \epsilon$$

where $\mid$ is the alternation symbol, . is the concatenation symbol, $\sim$ is the negation symbol, $*$ is the Kleene closure symbol, $+$ is the transitive closure symbol, ϵ is the empty string, and $p(\bar{T})$ and $-p(\bar{T})$ are literals. For an extended regular expression R, the language generated by R will be denoted as $L(R)$.

□

Definition 2.5 The set of variables contained in an edge label will be referred to as the set of *edge variables*. Similarly, the set of variables contained in a node label will be known as the *node variables*.

□

Definition 2.6 A *database graph* or *database instance* (db-graph) is a hygraph where every node is labelled with a literal and every edge and blob is labelled with a literal.

□

In the database graph definition, edges are labelled with literals that define the nature of the relationship between pairs of nodes. Thus the same pair of nodes can be related in different ways described by different predicates. For example, nodes a and b may be connected by a flight or a train trip, and the literals labelling the edges would be *flight*($\bar{U}$) or *train*($\bar{V}$), where $\bar{U}$ and $\bar{V}$ may be a sequence of ground terms, possibly empty. If the edge label is a literal that contains no terms, such as *f()*, then the label may be shortened to the predicate name, i.e. f.

Usually, a db-graph is only labelled with forward ground literals, though we adopt a less restrictive definition, which allows forward non-ground literals. One reason for this is that the result of a particular query may contain variables, and would otherwise not qualify as a db-graph for another query that is composed on this result. Alternatively, one could always map any such variables in the result to a "special" constant.

Definition 2.7 A *query* Q is a hygraph where each node is labelled with a literal and each edge (blob) is labelled by an edge (blob) label.

A *define query* $Q_D = (Q, D)$ is a query Q with a single distinguished edge D, labelled with a positive literal.

A *filter query* $Q_F = (Q, F)$ is a query Q with a set F of distinguished edges $\{D_1 \ldots D_n\}$ $(n > 0)$.

□

The distinguished edge referred to in Definition 2.7 defines a format for the answers to visual queries. Any bindings for variables associated with the distinguished edge (blob) are displayed in the answer, in the form of a graph.

Example 2.2 The query in Figure 2.2 consists of a single define query labelled **defineGraphLog** and a single filter query labelled **showGraphLog**. In addition, the define query contains two edges labelled with regular expression **calls+** as well as a distinguished edge **calls_sched**. The filter query is used to display the distinguished edge defined in the define query.

■

We shall not consider filter queries in future examples, since for our purposes, they simply refer to the distinguished edge in the define query. More detail on filter queries may be found in [Con94].

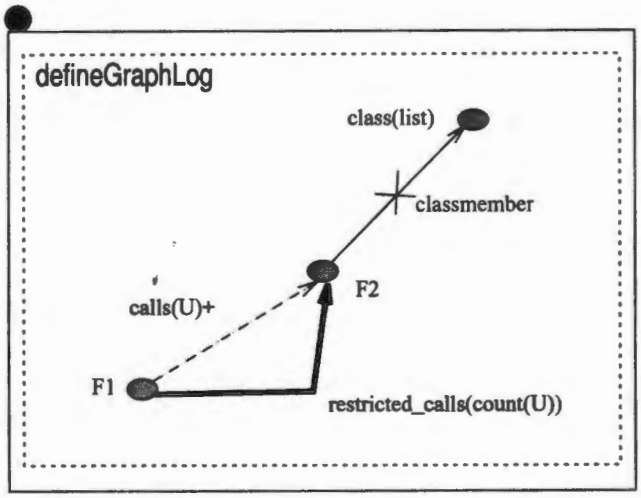

Figure 2.3: Query **restricted_calls**

The query in Figure 2.3 demonstrates how queries using negation and aggregation may be expressed. In this case, the aggregation operator is the **count** operator which returns the number of times variable U is instantiated in the answer. The inclusion of negation and aggregation in GraphLog has been examined in [Con89] [CM92] and [CMV94].

2.2 GRE

The GRE visual query language [Woo90] is also derived from G^+. It allowed a restricted set of queries that were single-edge, and introduced a translation to Datalog based on the structure of an NFA. No extensions like negation or aggregation were considered.

It is mentioned here mainly because the translation of GRE queries to Datalog defined a class of programs which were factorable, but was not comparable to any other class of factorable programs that had been described previously. In addition, semantics which were independent of Datalog were defined for it, unlike GraphLog whose semantics are defined in terms of Datalog programs generated by the translation.

Although GRE was a distinct language from GraphLog there were many similarities in syntax. This led to the use of the NFA-based technique to translate a restricted set of GraphLog queries [VW95]. An important aspect of this new translation, the NFA-translation, is the generation of factored Datalog as part of the translation, whereas the GRE translation generated Datalog which has to be factored subsequently. This NFA-translation forms the basis for the translation described in this thesis, and is discussed in Section 2.6.

2.3 Datalog

The Datalog database query language is based on Horn-clause logic. Such Horn-clause based query languages have been the subject of much research in recent years and various database systems incorporating these languages have been developed. One such system, CORAL, is the primary system with which the visual queries described in this thesis are evaluated. Other examples of database systems capable of evaluating logic queries include Aditi [VRK+90] and LDL [NT89].

In this section, we provide an overview of Datalog since it plays an important part in understanding the relationship between visual queries and their evaluation. Details of CORAL, as well as the actual query evaluation method and techniques for optimizing the evaluation will be considered in Section 2.4.

The purpose of the following definitions and descriptions is to introduce the required concepts, without providing extensive details of the theory of relational and logic programming systems. Therefore, some basic knowledge of these systems is assumed. This has been well documented in a number of publications including [Ull85] [Ull88] [BR86] [Cod70].

2.3.1 Syntax

The following definitions relating to Datalog are fairly standard, and have been obtained from various sources, including [Ull88] and [Con94].

Definition 2.8 Let U be a set of constants and V be a set of variables, such that $U \cap V = \emptyset$.

A *term* is either a constant $c \in U$, a variable $v \in V$ or an anonymous variable denoted _ (underscore).

□

We use the Prolog convention to denote variables and constants. This definition is based on the standard Datalog language. An extension of this is described in Section 2.3.4.

As mentioned earlier, there is an overlap of terminology between Datalog and GraphLog. This is purely for convenience reasons because of the close relationship between them. Datalog does not define various concepts such as "reverse literal" explicitly, and the semantics of these are only applicable to GraphLog.

Example 2.3 An example of a term is jason. The term NAME or Name is a variable.

■

Definition 2.9 An *atomic formula* is of the form $p(t_1, t_2, \ldots, t_n)$, where p is a predicate symbol, each *argument* $t_i (1 \leq t_i \leq n)$ is a term and n is the number of terms, called the *arity* of the atomic formula. An atomic formula with no arguments has an arity of zero, and is abbreviated p.

An *atom* is an atomic formula $p(t_1, t_2, \ldots, t_n)$, or $t_i = t_j$.

A *literal* is a *positive* (non-negated) atom or a *negative* (negated) atom, which is denoted by $\neg p(t_1, t_2, \ldots, t_n)$. A negated atom is also called a *negative literal*; a non-negated formula is a *positive literal*. A literal that contains no variables is *ground*.

If $p(t_1, \ldots, t_n)$ is a literal, then we refer to $(t_1, \ldots, t_n)$ as a *tuple*.

□

Example 2.4 To illustrate the terminology defined above, we consider some examples.

For instance, parent(X,jason) is a literal with arity 2 which has a variable and a constant term. The literal sister(martine,gary) contains constant terms and is ground.

■

Definition 2.10 A *clause* is a disjunction of literals. A *Horn-clause* is a clause with at most one positive literal, which can be written as

$$p_0 \vee \neg p_1 \vee \ldots \vee \neg p_n$$

which is logically equivalent to

$$p_0 \leftarrow p_1 \wedge \ldots \wedge p_n$$

where $n \geq 0$.

The second form is referred to as a *logic rule*, or informally as a rule. It is also written using the Prolog notation where the implication ($\leftarrow$) is replaced with a ":-" symbol and the conjunction ($\wedge$) is replaced with the "," symbol, mainly for practical programming purposes.

We call p_0 the *head* of the rule, $p_1, \ldots, p_n$ the *body* of the rule, and the individual literals in the body *subgoals*. A rule in which the body is empty is referred to as a *fact*. A rule which contains no variables is *ground*.

A Datalog *program* is a finite non-empty set of rules.

A clause with no positive literal is called a *query goal*. The Prolog convention of preceding query goals with a "?" is used.

□

Datalog can be viewed as an extension of relational database systems. There is a way of associating predicates used in Datalog with relations, by regarding the predicates as relations whose attributes appear in the the same order as the predicate arguments, and are referenced by their position. Thus a logic program may be regarded as representing a database.

Definition 2.11 If the predicate symbol e denotes a relation E which is stored in the database, e is called an *extensional database* (EDB) or *base* predicate and E is a *base* relation. EDB predicates do not appear in the head of a logic rule. A rule whose body consists entirely of EDB predicates is called a *base rule*.

If a predicate symbol i which denotes a relation I is defined exclusively by logic rules, then i is called an *intensional database* (IDB) or *derived* predicate and I is a *derived* relation.

```
anc(X,Y) :- par(X,Y).
anc(X,Y) :- par(X,T), anc(T,Y).
```

Figure 2.4: Ancestor program in Datalog

The set of base predicates make up the *extensional database* and the set of derived predicates make up the *intensional database.*

The *dependency graph* of a Datalog program P is a directed graph whose nodes are the IDB and EDB predicates of P. There is an edge from predicate p to predicate q if there is a rule with a subgoal whose predicate is p and with a head whose predicate is q.

A Datalog program is said to be *recursive* if its dependency graph has one or more cycles. We refer to a predicate that is on one or more cycles as a *recursive predicate.* A Datalog program with an acyclic dependency graph is said to be *non-recursive.* A *recursive* rule contains one or more recursive predicates as a subgoal. □

The underlying mathematical model of data for Datalog is essentially that of the relational model. Therefore, there may be some mixing of terminology such as referring to predicates as relations, or describing sets of values derived for predicates by rules, or contained in predicates in the extensional database, as tuples. In the relational model, all relations are EDB relations.

Example 2.5 An example of a Datalog program is displayed in Figure 2.4. This program demonstrates the standard method of recursively computing ancestors from a database containing parent information. The program consists of two rules, with the first rule containing a single literal in the body of the rule, and the second rule containing two body literals. The second rule is recursive. The query

```
?anc(X,Y).
```

computes all ancestors for a given database.

We may refer to the **par** predicate in the body of the rules as the EDB predicate, corresponding to a **PAR** relation stored in the database. The **anc** predicate is derived by the program, and is an IDB predicate. The first rule is a base rule.

■

2.3.2 Language Semantics

The meaning or semantics of a Datalog program can be defined in various ways, as described in [Ull88]. One possibility is to derive all facts that are derivable from each rule in the Datalog program. This corresponds to the *proof-theoretic* interpretation of the program.

Another possibility is to view the rules in the program as defining possible models or worlds. If the predicate arguments are chosen from an infinite domain of constants, truth or falsehood is assigned to every instance of the predicates in the rules, called an *interpretation*. A *model* is an interpretation of a set of rules which make those rules true. The minimal model defines the meaning of the program. This is known as the *model-theoretic* interpretation of the program. [1] It has been demonstrated that the *proof-theoretic* and the *model-theoretic* semantics coincide.

The third definition of semantics is to define an algorithm to compute the meaning of the Datalog program to determine whether a certain fact is true or false. This is called the *computational* interpretation or *fixpoint semantics* of a program.

The computational meaning is generally obtained by using a well-known algorithm, called *naive* evaluation. This algorithm computes the *least fixpoint* of a Datalog program. This coincides with the other two semantics, which is an important property of the algorithm that maintains the logical soundness and completeness of the results.

The naive algorithm computes "bottom-up" from a set of ground facts (EDB predicates) and derives new facts iteratively. The computation halts when no new facts can be derived. It has been demonstrated that for standard Datalog (without function symbols etc.), the algorithm terminates because of the finiteness of the extensional database, and is polynomially bounded in the number of iterations.

For the classes of Datalog program that we will consider, there is an improved algorithm called *semi-naive* evaluation which is used. Bottom-up computation, in the form of naive or semi-naive evaluation, is assumed for the optimizing techniques that will be discussed later. Semi-naive evaluation seems to be the generally accepted method of bottom-up computation of Datalog programs, and the majority of logic database systems, including CORAL, make use of it.

The following example demonstrates how semi-naive evaluation is used to compute the answer to a Datalog query

[1] Note that there may be multiple minimal models, particularly when negation is considered.

Example 2.6 Consider again the program in Figure 2.4, which derives the ancestor relation. Suppose the query

?anc(jason,Y).

is evaluated on a database of facts corresponding to the database graph in Figure 1.1 on page 2. In other words, each edge from a node labelled X to a node labelled Y with an edge label p in the graph corresponds to a Datalog fact $p(X,Y)$. We will refer to the "node with label ..." simply by the node label; similarly for edge labels. For example, the edge **par** from node **jason** to node **peter** in Figure 1.1 corresponds to the Datalog fact par(jason,peter). which is interpreted to mean "jason's parent is peter". The number of facts is equal to the total number of edges in the database graph. In Figure 1.1, there are 12 facts in total, of which 10 are **par** facts and 2 are **friend** facts.

The correspondence between database graphs and Datalog facts is formally defined in Section 2.5, and examined further in Chapter 3 where the semantics of visual queries and databases is compared to the semantics of Datalog queries and databases.

On the first evaluation step, the following set of answers is computed for **anc** from the first (base) rule of the program:

{ (jason,peter), (jason,jane), (susan,judy), (susan,bob), (peter,michael), (peter,lisa), (judy,linda), (judy,john), (linda,jack), (linda,mary)}

These answers correspond to paths of length 1 in the database graph; in other words, all the parents of each person in the graph.

On the second evaluation step, the following answers are computed for **anc**:

{ (jason,michael), (jason,lisa), (susan,linda), (susan,john), (judy,jack), (judy,mary). }

This is obtained by joining the set of answers computed in the first step with all the parent relationships in the database.

On the third evaluation step, the benefits of semi-naive evaluation become evident. Whereas naive evaluation would compute the next set of ancestors by finding the parents of all ancestors computed thus far (thus recomputing most of the information), semi-naive evaluation only uses the set of answers computed in the previous step to derive the next generation of ancestors:

{ (susan,jack), (susan,mary). }

No more answers can be computed.

After all the steps are complete for the anc predicate, the answers relevant to the query are filtered, namely:

{ (jason,peter), (jason,jane), (jason,michael), (jason,lisa). }

∎

The previous example demonstrates one of the classic disadvantages of bottom-up evaluation since the entire ancestor tree is first computed for every person in the graph. Subsequently, the relevant set of answers is selected from the derived anc predicate, which may contain a large amount of ancestor information which is entirely irrelevant for this query.

Note that there may be a number of equivalent Datalog programs that are syntactically different. The evaluation of these different syntactic forms has performance implications, and can result in significantly different amounts of computation required. This is the reason why the optimization of queries is possible, which is considered in Section 2.4.

The details of the various evaluation algorithms may be obtained from [Ull88] [CGT90].

2.3.3 Safety

An important aspect of Datalog queries is whether they result in a finite answer. We say that a query is *safe* if it has a finite answer independent of the database domain (which may be finite or not) [GV89].

For Datalog programs, sources of infiniteness in the rules are as follows:

1. Evaluable predicates, like arithmetic expressions and comparisons, in the rule bodies. For example, the query ?greater-than(42,X) is unsafe.
2. Variables in the head of the rule which do not appear in the rule body.

Example 2.7 Consider the rules in Figure 2.5. They are not safe, since the predicates in the heads of each rule may represent infinite relations, if the domain was infinite.

∎

The problem of ensuring safe Datalog programs has been examined in [Ull88], where syntactic constraints on the form of Datalog rules is proposed. A more general evaluation method, which permits unsafe queries, is described in [Ram88] by permitting the existence of "non-ground facts". This method is used in the CORAL system described in Section 2.4.1.

```
big_number(Y) :- par(A,B).
dislikes(X,Y) :- bully(Y).
```

Figure 2.5: Unsafe rules in Datalog

2.3.4 Extensions

There have been a number of extensions to Datalog which have added power and flexibility to the language, and generalized it to full logic programming. However, they have also brought about various difficulties, related to the desired meaning and implementation of a suitable algorithm to compute this meaning.

The definition of a term has been extended from the original Datalog definition in Definition 2.2 as follows:

Definition 2.12 Let U be a set of constants and V be a set of variables, such that $U \cap V = \emptyset$.

A *term* is either a constant $c \in U$, a variable $v \in V$, an anonymous variable denoted _, or a function f applied to a number of terms. The function f could be anonymous.

We call a term without variables, a *ground term* or *constant term.* A *variable term* is a term that contains at least one variable.

□

The major difference is that this definition allows functions which may be applied to a number of terms. The recursive nature of the definition means that terms which are not variables or constants are referred to as *structured* or *complex* terms. The models of programs using these terms is not always finite and no longer conforms to the relational model where values must be atomic.

Unlike the definition of a term in standard Datalog, this definition is closer to Definition 2.2 for GraphLog. This extended definition is assumed for the translation described for GraphLog in future sections.

Example 2.8 The term person(jason) is ground and complex, whereas person(Name) is a complex variable term.

We may use the extended definition of terms in literals as well. For instance, we may create a literal **parent(X,person(jason))** with arity 2 which has a variable and a constant term. The literal **sister(person(martine),person(gary))** consists of complex terms and is ground.

∎

Another substantial extension to Datalog has been the inclusion of *evaluable* or *built-in* predicates that are not stored as EDB or IDB predicates but are computed instead. These include arithmetic operators like $\text{sum}(X, Y, Z)$ (also denoted infix as $Z = X + Y$) and comparison operators like $<(X, Y)$ (infix as $X < Y$). Because they represent infinite relations, these predicates can result in unsafe programs, which was mentioned in Section 2.3.3.

CORAL also allows the use of sets and multisets in rules, and *aggregate functions*, which are a common feature of relational database systems. These functions are used in *aggregation* operations, such as computing the average, the maximum value or the minimum value of a set or multiset of values.

Another significant extension to Datalog is the use of *negation*. Although this has been a very useful addition, it has brought with it a number of semantic consequences such as multiple minimal models. By restricting the class of programs with negation, it is possible to establish the computational meaning of a program in terms of a model that coincides with one of the program's minimal models, called the *perfect fixpoint*. This form of negation is known as *stratified negation*, which is adequate for GraphLog.

2.4 Query Evaluation and Optimization

This section examines the features and details of the CORAL logic database system, and the way in which Datalog programs are evaluated by it. CORAL is the standard evaluation system used by Hy$^+$ although various alternatives are provided, including evaluation using Prolog [Fuk91], as well as the LDL system [CMV94] [NT89]. We also consider the background to the various standard optimizations that may be performed by CORAL.

Since a significant aspect of the NFA-translation (described in Section 2.6) is the efficiency of evaluation of the Datalog program generated, it is important to be aware of the optimization alternatives that are available. The techniques implemented in CORAL are well established and have been described in various articles. They include magic sets [BMSU86], supplementary magic sets [BR87], and context rewriting [KRS90]. We describe these below as well as the factoring optimization [NRSU89a] which forms the basis of the NFA-translation.

2.4.1 CORAL

CORAL is a database programming system being developed at the University of Wisconsin-Madison [RSS92] [RSSS94] [RSSS93]. It attempts to provide the benefits of a (deductive) database query language, such as efficient treatment of large relations, aggregation operators and declarative semantics, and combines them with the features of a logic programming language, such as powerful inference features and support for incomplete and structured data.

In addition to this, CORAL provides extended functionality including negation, extensibility using an interface to C++ and persistent storage using the EXODUS storage manager [CDRS86]. Although there are a number of extensions to standard Datalog supported by CORAL, we will mainly concentrate on those features of Datalog that were described in Section 2.3.

An interesting and very powerful feature of CORAL, that has distinguished it from similar systems, is its support of non-ground facts, in other words, facts that contain variables. The consequences of this for visual queries is significant and it is discussed in Section 4.7 on safety.

Various optimization techniques are supported by CORAL. Those that interest us for the purpose of this thesis are the bottom-up rewriting techniques such as magic and supplementary magic templates as well as context rewriting. These are referred to as *annotations* in CORAL. For example, the magic rewriting is specified using the `@magic` annotation.

The system defaults to supplementary magic templates rewriting. The user must specify the other optimizations explicitly for CORAL to attempt them. If the chosen optimizations are not possible for a particular query, the default method is used. Note that context rewriting is referred to as context factoring in CORAL documentation, and is enabled using the `@factoring` annotation. This method is related to the factoring rewriting technique described in [NRSU89a]. However, factoring is not supported by CORAL.

Example 2.9 An example of a CORAL program using an annotation is:

```
module anc_example.
export anc(bf).

@factoring.
```

```
anc(X,Y) :- par(X,Y).
anc(X,Y) :- par(X,T), anc(T,Y).
```

■

The major types of rewriting techniques performed by CORAL are described in the following sections. Since our interest lies primarily in their use in the optimization of visual queries where database relations are viewed as directed graphs, we describe these techniques from a perspective of finding paths in a graph. Therefore, a graph representation of a database, shown in Figure 1.1 on page 2, will be used to describe how different techniques reduce the traversal of this database graph, and so reduce the amount of computation required.

The examples used below are relatively straightforward, as they are intended simply to illustrate each optimization method. However, each technique is more general than a single example might suggest, and the various references describe the techniques for the full classes of queries to which they are applicable.

2.4.2 Magic Rewriting Techniques

Probably the most well-known optimization techniques for bottom-up computation are those based on magic sets [BMSU86]. These techniques have been extended and generalized to include all Datalog queries.

It is easy to observe that the semi-naive evaluation in Example 2.6 on page 19 does much unnecessary computation and stores intermediate data which is not required for the query described, particularly by examining the database graph in Figure 1.1. The method evaluates and stores values for the subgraph reachable from "susan" which are clearly not required in the answer.

Although semi-naive evaluation is reasonably efficient when a query does not contain instantiated variables, it is very inefficient when the bindings are given for some variables in the query. In this case, a top-down evaluation strategy (as used by Prolog), would be more efficient, since computation proceeds using the variable bindings contained in the query.

Computation is reduced when bindings are passed in the query. If we are given a rule and some bindings for a subgoal in the rule, we can solve the subgoal and obtain bindings for other variables in the subgoal. These bindings can be "passed" to other subgoals in the same rule, and so reduce computation for these subgoals. This is known as *sideways information passing.*

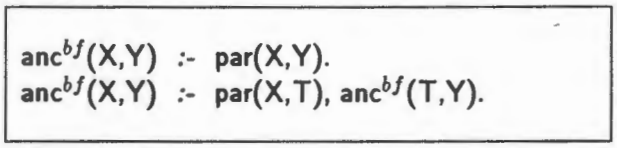

Figure 2.6: Adorned ancestor program

Information passing is the standard behaviour of top-down evaluation. In order to simulate the binding passing strategy of top-down methods, various additional subgoals (called *magic predicates*) are introduced in the bodies of rules, and additional rules to define these goals are added. This has the effect of restricting the set of values which variables can be bound to during bottom-up evaluation, and usually makes the program more efficient.

We can describe the bindings determined and passed in a program by annotating each IDB predicate in the program with a binding pattern or *adornment*. This is simply a mapping that assigns a string 'b' for bound arguments, and string 'f' for free arguments. For example, the adorned program derived from the Datalog program in Figure 2.4 on page 17 and query goal ?anc(jason,Y) is shown in Figure 2.6.

This adorned program is used to determine the magic predicates and rules for the magic rewritten program. We can see sideways information passing in the second subgoal of the second rule, where variable T is bound since it is also present in the first subgoal which contains bound variable X. In fact, the principle of sideways information passing, described by an adorned program, is used by most rewriting methods for efficient bottom-up evaluation, including the factoring and context-rewriting method described in the following sections.

A simple example of a magic set rewriting is described, to provide insight into the optimization. Further details on magic sets and its generalization to larger classes of Datalog programs can be found in [BR87] and [CGT90].

Example 2.10 Figure 2.7 shows the rules obtained by using the magic sets transformation on the program in Figure 2.4 on page 17, for the same query used in Example 2.6. The rules containing the magic predicate m_anc in the head are the *magic rules*.

These magic rules are used to compute the bindings corresponding to "jason" and all his ancestors. The predicate m_anc therefore contains:

{ (jason), (peter), (jane), (michael), (lisa) }.

```
anc(X,Y)   :- m_anc(X), par(X,Y).
anc(X,Y)   :- m_anc(X), par(X,T), anc(T,Y).
m_anc(T)   :- m_anc(X), par(X,T).
m_anc(jason).
```

Figure 2.7: Magic set transformation of ancestor program

The magic predicate is used to restrict bindings in the other rules. Therefore the computation considers only paths that are routed from "jason" in Figure 1.1.

The set of answers computed for the anc predicate is :

{ (jason,peter), (jason,jane), (peter,michael), (peter,lisa), (jason,michael), (jason,lisa) }

These answers reflect the restriction of the bindings by the magic set; no values related to "susan" or her ancestors are present.

The result of the query is then filtered from this, namely:

{ (jason,peter), (jason,jane), (jason,michael), (jason,lisa) }

However, suppose the relation for par looks something like the following set of facts:

$\{(\text{jason}, c_1), (c_1, c_2), \ldots, (c_{n-1}, c_n)\}$

Although the answer to the query contains only n tuples, the number of tuples generated for anc is $\Omega(n^2)$.

■

It is quite obvious that the number of generations of ancestors for "susan" in Figure 1.1 can be increased, so that the amount of unnecessary computation performed by the semi-naive evaluation of the program in Figure 2.4 on page 17, as well as the number of irrelevant answers generated for anc, is arbitrarily large. However, even though the magic sets transformation is potentially much better than a non-rewritten program, we saw at the end of the previous example that it can be inefficient for various classes of queries. This has led to the development of more efficient techniques such as factoring and context rewriting, which are introduced next. However, the magic sets transformation does handle various classes of program which the other techniques cannot.

```
m_anc(jason).
m_anc(Y) :- m_anc(X),par(X,Y).
anc(jason,Y) :- m_anc(X), par(X,Y).
```

Figure 2.8: Factored ancestor program

The particular forms of magic rewriting used in CORAL are magic and supplementary magic templates [Ram88], which are powerful extensions of the original method that allow the rewriting of queries that include complex terms.

2.4.3 Factoring

The term factoring refers, in general, to the process of replacing a recursive predicate $p(\bar{X}, \bar{Y})$ by predicates $bp(\bar{X})$ (the *bound* part of p) and $fp(\bar{Y})$ (the free part of p) [NRSU89a]. However, our focus is on the translation and evaluation of GraphLog queries, and the bound parts of predicates can always be deleted from Datalog programs produced by the NFA-translation (described in Section 2.6) that compute the results of a single-edge GraphLog query. Therefore, we will use the term "factoring" to refer to the reduction of the arity of p by replacing it with fp. This means that the more general factoring techniques described in [NRSU89a] are not necessary for the NFA-translation to generate factored Datalog output.

There are restrictions on the class of programs considered in [NRSU89a]. Only programs containing right-linear, left-linear and combined-linear rules, in terms of a single IDB predicate and one exit rule are considered, in addition to other restrictions. There are programs generated by the NFA-translation which are outside this class, yet are factored.

Example 2.11 A closer look at the magic rules described in Example 2.10 will reveal that the magic set contains all the answers in the final result. The factored program in Figure 2.8, generated from the program in Figure 2.4 on page 17, is based on this observation.

Therefore if we were to formulate the query

```
?anc(jason,Y).
```

evaluated on the same database corresponding to the database graph in Figure 1.1, we observe that the rewritten program generates the ancestors of jason exactly for anc, as

```
mc_anc(C,C)  :- friend(B,C).
mc_anc(C,Z)  :- mc_anc(C,X), par(X,Z).
ac_anc(C,Y)  :- mc_anc(C,Z), par(Z,Y).
result(B,C,Y) :- friend(B,C), ac_anc(C,Y).
```

Figure 2.9: Context Rewritten ancestor program

compared to the magic sets rewriting that generates unnecessary ancestor information for "susan" as well:

{ (peter), (jane), (michael), (lisa) }.

The first rule restricts the search to start at the node **jason** and the second (recursive) rule follows all paths that begin at the restricted node. In this way, duplication of computation is minimized.

Using the relation for **par** described in Example 2.10, there is a major improvement for the worst case compared to the magic sets transformation, as the number of tuples generated for **anc** is only $O(n)$.

■

2.4.4 Context Rewriting

The context-transformation algorithm is a rewriting technique that is described in [KRS90] as extending the work presented in [NRSU89b]. However, unlike the latter technique, context rewriting does not result in a reduction of the arity of predicates. The classes of rules handled are left-linear, right-linear and multi-linear rules [NRSU89b].

Additionally, context rewriting handles calls whose input arguments are not manifest constants, but are computed by other calls in the query [KRS90]. This last property allows the algorithm to optimize programs, described as programs with *context information*, which do not contain any constants. These programs may be optimized considering calls that occur in the bodies of the rules, in addition to those in queries.

Since context rewriting is considered to be an extension of the standard factoring technique, the following example demonstrates the additional benefits of context rewriting, namely its

improved efficiency for various queries that do not contain constants amongst their input arguments.

Example 2.12 Suppose that for some binary base predicate, **friend**, we wished to compute the query:

?friend(B,C),anc(C,Y).

on the sample database in Figure 1.1. Clearly, magic sets rewriting is not applicable since no constants are present in the query arguments.

However, the original program in Figure 2.4 on page 17 may be rewritten using the context-transformation algorithm, to derive the program in Figure 2.9. The first rule stores context information in the form of a set of values that correspond to all the friends C of any person B. These values are then used to restrict the evaluation of all ancestors of C.

In other words, the predicate **mc_anc** contains the following tuples after the application of the first rule:

{ (susan,susan), (judy,judy) }

This set of bindings is used to constrain the search and the second rule only computes the ancestors of **susan** and **judy**, respectively. Thus, the final set of tuples for **mc_anc** is:

{ (susan,susan), (judy,judy), (susan,judy), (susan,bob), (susan,linda), (susan,john), (susan,jack), (susan,mary), (judy,linda), (judy,john), (judy,jack), (judy,mary). }

The third rule in the program effectively eliminates the tuples **(susan,susan)** and **(judy,judy)** from the set of tuples computed by the second rule, since these tuples only serve to restrict the bindings and are not part of the answer.

Whereas the original program would compute an entire set of ancestors for every person in the graph, and then attempt to join them with the **friend** predicate, the rewritten program passes the **friend** context information to constrain the evaluation of the ancestor computation, and reduce the size of intermediate results.

Again, the number of generations of ancestors for "jason" contained in the database graph may be increased, and in a similar manner described in Example 2.10, an arbitrary increase in efficiency may be obtained for this optimization. However, if the **friend** predicate has m tuples and the **par** predicate has n tuples, then the number of tuples generated is $O(mn^2)$.

■

```
common_anc(linda,Y) :- tc_par(T,linda), tc_par(T,Y).
tc_par(X,Y) :- par(X,Y).
tc_par(X,Y) :- par(X,T), tc_par(T,Y).
```

Figure 2.10: Common ancestor program

The following example describes a query where context rewriting is not applicable and CORAL defaults to a magic rewriting method, which may be substantially less efficient.

Example 2.13 Consider the query in Figure 2.10. This program finds all the ancestors of people who have **linda** as an ancestor. This query cannot be context rewritten because each **tc_par** subgoal in first rule is called with different bindings.

∎

2.4.5 Discussion

The principles of factoring have been applied to an extended class of programs, the *augmented regular chain programs* [Woo90]. The standard factoring and context rewriting methods handle classes of programs restricted to left-linear, right-linear and multi-linear rules. In addition, context rewriting is applicable to programs whose input arguments are not manifest constants.

However, neither method handles programs which include multiple exit rules and mutually recursive rules. Also, these methods only work for queries where there is a single binding pattern for each IDB predicate. As a result, CORAL defaults to a more inefficient method for these programs, which may result in an order of magnitude slowdown. For certain classes of queries that include the above properties, a technique for producing factored programs is described in [Woo90]. This technique is performed as part of the NFA-translation for GraphLog queries [VW95] and is also a motivation for using the NFA-translation when translating certain queries in Hy$^+$. The potential benefits will be demonstrated with various examples and performance results in Chapter 5.

2.5 Standard Translation of GraphLog to Datalog

The format of the translation that is presented here is defined in [CMV94] and will be known as the *RE-translation.* This is the translation that is implemented in the Hy$^+$ system. For clarity, the notation used is similar to the translation defined in [VW95], although this translation is not identical. The reasons for the modified translation in [VW95] are discussed in Chapter 3.

Since GraphLog queries are translated to Datalog for evaluation, a Datalog translation of the db-graphs on which they operate is also required. Therefore, we describe how a db-graph G may be represented as a set of Datalog facts F.

Definition 2.13 Consider a labelled edge e in a db-graph G with source node label c_1, sink node label c_2 and edge label $p_i(c_3, c_4, \ldots)$. For convenience, we will denote e by the tuple $(c_1, c_2, p_i(\bar{c_3}))$, where $\bar{c_3}$ will represent all terms $(c_3, c_4, \ldots)$. If the edge label does not contain terms within the parentheses, as in edge label p_i, then the tuple will be denoted (c_1, c_2, p_i).

□

Definition 2.14 Let $G = (N, L_N, \nu, L_E, E, L_B, B)$ be a db-graph.

We define the set of Datalog facts F_G that *correspond* to G as follows:

For every edge $e = (X_1, X_2, p(\bar{X}_3)) \in E$, generate a fact f, denoted by $p(X_1, X_2, \bar{X}_3)$. We say that fact f *corresponds* to edge e.

Every fact $f \in F_G$ has a corresponding edge e in G and vice-versa.

□

The meaning of a define query is to define the distinguished edge of the query. The translation for a define query which has nondistinguished edges $p_1(X_1, Y_1, \bar{E}_1), \ldots, p_n(X_n, Y_n, \bar{E}_n)$ contains the following rule, also referred to as a *top-level rule*, which defines the predicate p of the distinguished edge $p(X, Y, \bar{E})$:

$$p(X, Y, \bar{E}) \text{ :- } l_1(\bar{F}_1), l_2(\bar{F}_2), \ldots, l_n(\bar{F}_n).^1$$

where $l_i(\bar{F}_i)$ is $p_i(X_i, Y_i, \bar{E}_i)$ if the edge is positive, $\neg p_i(X_i, Y_i, \bar{E}_i)$ if the edge is negated, and is defined recursively on the structure of the label according to the following algorithm:

[1] The idea of using $\bar{F}_i$ instead of $X_i, Y_i, \bar{E}_i$ is that constants and anonymous variables need not appear in the call of a predicate that is associated with an expression (defined with additional rules).

- *Inversion*: Label $-e_m(\bar{Z}_m)$ corresponds to the literal $e_m(Y, X, \bar{Z}_m)$.
- *Concatenation*: Label $e_m(\bar{Z}_m) \cdot e_n(\bar{Z}_n)$ corresponds to the rule

  $$conc\_e_m\_e_n(X, Y, \bar{F}_m, \bar{F}_n) \text{ :- } e_m(X, T, \bar{Z}_m),\ e_n(T, Y, \bar{Z}_n).$$

  where $\bar{F}_i$ contains the named free variables of $\bar{Z}_i$ (without bound terms and anonymous variables), and T is a variable appearing nowhere else in the rule.
- *Alternation*: Label $e_m(\bar{Z}_m)|e_n(\bar{Z}_n)$ corresponds to the rules

  $$alter\_e_m\_e_n(X, Y, \bar{F}_m, \bar{F}_n) \text{ :- } e_m(X, Y, \bar{Z}_m).$$

  $$alter\_e_m\_e_n(X, Y, \bar{F}_m, \bar{F}_n) \text{ :- } e_n(X, Y, \bar{Z}_n).$$

  where $\bar{F}_i$ contains the named free variables of $\bar{Z}_i$. Note that if $\bar{Z}_m$ and $\bar{Z}_n$ contain different variables, the translation generates rules with unbound variables (non-range-restricted or unsafe rules).
- *Closure*: Label $e_m(\bar{Z}_m)+$ corresponds to the rules

  $$tc\_e_m(X, Y, \bar{F}_m) \text{ :- } e_m(X, Y, \bar{Z}_m).$$

  $$tc\_e_m(X, Y, \bar{F}_m) \text{ :- } e_m(X, T, \bar{Z}_m),\ tc\_e_m(T, Y, \bar{F}_m).$$

  where $\bar{F}_m$ contains the named free variables of $\bar{Z}_m$ (which are restricted to have the same value along the path), and T is a variable appearing nowhere else in the rule.
- *Kleene Closure*: Label $e_m(\bar{Z}_m)*$ corresponds to the rules

  $$kleene\_e_m(X, X, \bar{F}_m) \text{ :- } e_m(X, Y, \bar{Z}_m).$$

  $$kleene\_e_m(Y, Y, \bar{F}_m) \text{ :- } e_m(X, Y, \bar{Z}_m).$$

  $$kleene\_e_m(X, Y, \bar{F}_m) \text{ :- } e_m(X, T, \bar{Z}_m),\ kleene\_e_m(T, Y, \bar{F}_m).$$

  where $\bar{F}_m$ contains the named free variables of $\bar{Z}_m$ (which are restricted to have the same value along the path), and T is a variable appearing nowhere else in the rule.

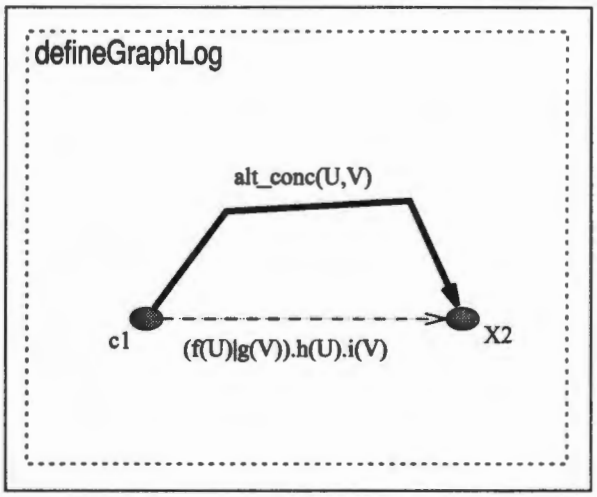

Figure 2.11: Query alt_cat

Example 2.14 Suppose that we create a query similar to the query in Figure 1.2 on page 3, except that variable X1 is replaced by a constant jason. The RE-translation produces the following Datalog translation for this query:

```
anc(jason, X2) :- tc_par(jason, X2).
tc_par(Xtemp1, Ytemp1) :- par(Xtemp1, Ytemp1).
tc_par(Xtemp1, Ytemp1) :- par(Xtemp1, Ztemp1), tc_par(Ztemp1, Ytemp1).
```

We observe that this is essentially the same query described in Example 2.6 on page 19. As demonstrated in that example, this translation leads to an extremely inefficient evaluation of the query, since all ancestors are computed in the second and third rules before the ancestors of jason are determined in the first rule. This program could be rewritten using magic sets, obtaining a program similar to the one in Figure 2.7 on page 26. As pointed out in Example 2.10, although the magic-rewritten program is more efficient, there is still redundant computation that is performed. The context-rewriting and factoring methods eliminate this redundant computation.

■

Example 2.15 Consider the query in Figure 2.11. We perform the RE-translation on the structure of the regular expression labelling the query edge, and produce the following Datalog program:

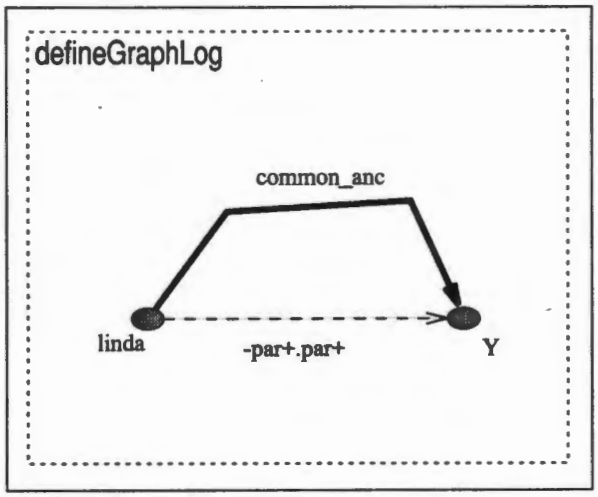

Figure 2.12: Query common_anc

```
alt_conc(c1,X2,U,V) :- compo1(c1,X2,U,V,U,V).
compo1(c1,X2,U,V,U,V) :- alter2(c1,Temp1,U,V), compo3(Temp1,X2,U,V).
alter2(c1,Temp1,U,V) :- f(c1,Temp1,U).
alter2(c1,Temp1,U,V) :- g(c1,Temp1,V).
compo3(Temp1,X2,U,V) :- h(Temp1,Temp2,U), i(Temp2,X2,V).
```

■

Example 2.16 Consider the query in Figure 2.12. When translated by the RE-translation, we obtain a program similar to the program in Figure 2.10 on page 30. Example 2.13 described why the context rewriting method is not applicable to this query. Instead, a magic rewriting method is used, with a possible order of magnitude slowdown compared to context rewriting. In Section 2.6, we shall see that if we use the NFA-translation instead of the RE-translation, a factored Datalog program is generated.

■

2.6 The NFA-translation

The *NFA-translation* [VW95] is based on the construction of a nondeterministic finite state automaton, as compared to the RE-translation which is based on the structure of the regular

expression in the GraphLog query. The NFA-translation is applicable to a GraphLog query comprising a single define query and a single filter query with a single distinguished edge. Furthermore, the define query has only a single non-distinguished edge, in addition to its distinguished edge. The non-distinguished edge can be labelled with an arbitrary regular expression. The filter query simply contains the distinguished edge of the define query.

Although this is a fairly restrictive class of GraphLog queries, a number of queries which do fall into the class show considerable performance improvements when they are translated into factored programs [VW95]. We will examine various extensions to the basic factoring translation that is presented below in Chapter 4.

We first recall the definition of a nondeterministic finite state automaton (NFA) [HU79]:

Definition 2.15 A *nondeterministic finite state automaton* M is a 5-tuple $(S, \Sigma, \delta, s_0, F)$, where S is a finite set of *states*, Σ is the *input alphabet*, δ is the *state transition function* which maps $S \times (\Sigma \cup \{\epsilon\})$ to the set of subsets of S, $s_0 \in S$ is the *initial state*, and $F \subseteq S$ is the set of *final states* (or *accept states*). The *extended* transition function δ^* is defined as follows. For $s, t \in S$, $a \in \Sigma$, and $w \in \Sigma^*$

$$\delta^*(s, \epsilon) = \{s\}, \text{ and}$$
$$\delta^*(s, wa) = \bigcup_{t \in \delta^*(s,w)} \delta(t, a)$$

The NFA M *accepts* $w \in \Sigma^*$ if $\delta^*(s_0, w) \cap F \neq \emptyset$. The language $L(M)$ *accepted* by M is the set of all strings accepted by M. □

Let R denote the regular expression in define query Q. The first stage in the translation is to construct an NFA M from R which accepts the language $L(R)$. The translation described is for the case where the source node of Q is labelled with a constant. The case where the sink node is labelled with a constant requires the automaton M to be reversed, as well as the inversion of each term labelling a transition in M.

The second stage of the translation involves generating a Datalog program using the following steps: [2]

1. Generate the fact $t\_s(c)$, where s is the initial state of the automaton, and c is the node constant in Q.

[2] In order to create predicates with valid Datalog names, we prefix each state number with t_.

2. For each transition t from p to q labelled with $e(\bar{Z})$, where $\bar{Z}$ may be a sequence of terms, generate a Datalog rule as follows:

   $$t\_q(Y) \text{ :- } t\_p(X), e(X, Y, \bar{Z}).$$

   If, instead, t is labelled with $-e(\bar{Z})$, generate:

   $$t\_q(Y) \text{ :- } e(Y, X, \bar{Z}), t\_p(X).$$

   Call the resulting program P.

3. From P, generate a new program Q by performing a bottom-up propagation of the edge variables as follows:

   (a) Add each fact $t\_s(c)$ to Q.

   (b) For each rule in P containing a t_s subgoal, say,

   $$t\_t(Y) \text{ :- } t\_s(X), e(X, Y, \bar{Z}).$$

   add the rule

   $$t\_t_{\bar{U}}(Y, \bar{U}) \text{ :- } t\_s(X), e(X, Y, \bar{Z}).$$

   to Q, where $\bar{U}$ comprises all the named variables in $\bar{Z}$.

   (c) Repeat the following process until no (syntactically) new rule is added to Q. If there is a rule with head $t\_t_{\bar{U}}(Y, \bar{U})$ in Q and a rule of the form

   $$t\_p(Y) \text{ :- } t\_t(X), e(X, Y, \bar{Z}).$$

   in P, then add the rule

   $$t\_p_{\bar{V}}(Y, \bar{V}) \text{ :- } t\_t_{\bar{U}}(X, \bar{U}), e(X, Y, \bar{Z}).$$

   to Q, where $\bar{V}$ is a sequence of named variables which includes all of those in $\bar{Z}$ and $\bar{U}$ such that the ordering of variables is consistent throughout the set of rules.

4. Finally, add rules for the distinguished edge. These rules have as their head the distinguished predicate and each body contains a single predicate $t\_p_{\bar{V}}$, where p is a final (or accept) state of the automaton.

The translation for GraphLog queries in which neither node is labelled with a constant is a simple modification of the above. The ground fact $s(c)$ is replaced by a non-ground fact $s(X, X)$ and each IDB predicate takes an additional, new first argument, say W.

The NFA-translation produces factored programs for queries containing node constants. The arity of IDB predicates in these programs is reduced, and the node constant is present in a base predicate. Intuitively, the translation may be considered from a graph-based perspective, where evaluation starts at a node labelled by the query constant, and proceeds to follow paths which satisfy the query regular expression as represented by an NFA.

Example 2.17 Suppose that we translate the query referred to in Example 2.14 with the NFA-translation. This is the query in Figure 1.2 on page 3, except that variable X1 is replaced by a constant jason. We contruct the automaton for the regular expression par+ and obtain the following Datalog program P from the first two steps:

```
t_0(jason).
t_1(Y) :- t_0(T), par(T,Y).
t_1(Y) :- t_1(T), par(T,Y).
```

Since there are no edge variables, program Q generated from P is identical. We also add a rule for the distinguished edge:

```
anc(jason,X2) :- t_1(X2).
```

∎

We observe that the program produced by the NFA-translation in the previous example is factored, and is similar to the program in Figure 2.8. This can be compared to the RE-translation of the same query in Example 2.14 on page 33 which produced a program that was not optimized.

Example 2.18 We reconsider the query in Figure 2.11 on page 33, describing how the NFA-translation is used to generate a Datalog query. This requires the construction of an NFA, depicted in Figure 2.13, from the regular expression (f(U) | g(V)).h(U).i(V) labelling the query edge.

Using the first two steps, we generate a program P:

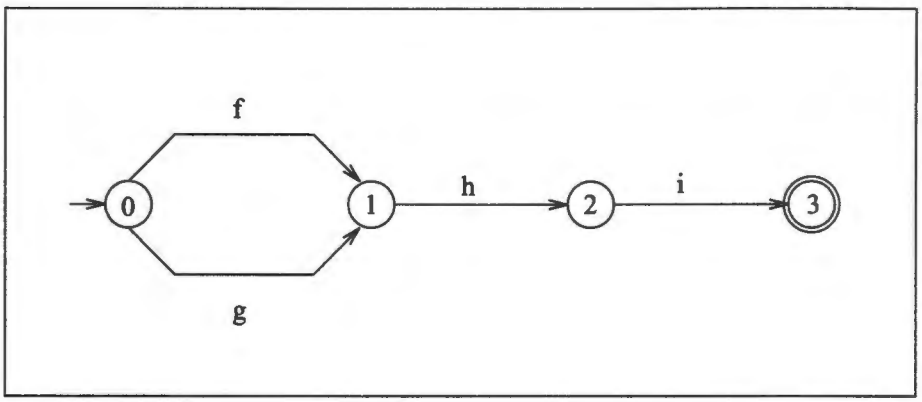

Figure 2.13: NFA for query **alt_conc**

```
t_0(c1).
t_1(Y) :- t_0(T), f(T,Y,U).
t_1(Y) :- t_0(T), g(T,Y,V).
t_2(Y) :- t_1(T), h(T,Y,U).
t_3(Y) :- t_2(T), i(T,Y,V).
```

Since we are interested in the values of the edge variables, and since they contribute towards the query semantics, we use the edge variable propagation method, described in third step, to generate a program Q.

We first consider the base rules which add rule

```
t_0(c1).
```

to Q.

Next, we consider step (2). In program P, the rules that have a `t_0` subgoal are the second and the third rules, so we propagate the variables and add rules

```
t_1_U(Y,U) :- t_0(T), f(T,Y,U).
t_1_V(Y,V) :- t_0(T), g(T,Y,V).
```

to Q.

We then consider step (3), and observe that the fourth rule in P has a `t_1` subgoal. Hence, we add the following two rules to Q.

```
t_2_U(Y,U) :- t_1_U(T,U), h(T,Y,U).
t_2_U_V(Y,U,V) :- t_1_V(T,V), h(T,Y,U).
```

Now the fifth rule in P has a t_2 subgoal, so we add the following rules to Q.

```
t_3_U_V(Y,U,V) :- t_2_U(T,U), i(T,Y,V).
t_3_U_V(Y,U,V) :- t_2_U_V(T,U,V), i(T,Y,V).
```

Finally, we add a rule for the distinguished edge.

```
alt_conc(c1,X2,U,V) :- t_3_U_V(X2,U,V).
```

∎

Example 2.19 Suppose that we replace constant c1 in Figure 2.11 with a variable X1. We translate the query using the NFA-translation with the modification described, and obtain:

```
t_0(X,X).
t_1_U(W,Y,U) :- t_0(W,T), f(T,Y,U).
t_1_V(W,Y,V) :- t_0(W,T), g(T,Y,V).
t_2_U(W,Y,U) :- t_1_U(W,T,U), h(T,Y,U).
t_2_U_V(W,Y,U,V) :- t_1_V(W,T,V), h(T,Y,U).
t_3_U_V(W,Y,U,V) :- t_2_U(W,T,U), i(T,Y,V).
t_3_U_V(W,Y,U,V) :- t_2_U_V(W,T,U,V), i(T,Y,V).
alt_conc(X1,X2,U,V) :- t_3_U_V(X1,X2,U,V).
```

∎

The need for variable names to be included as part of the predicate names during propagation is demonstrated in the following example.

Example 2.20 Suppose we translate the query in Figure 2.11 as described in the previous example, but do not include the propagated edge variable names as part of the predicate name.

We obtain the following program Q_2:

```
t_0(X,X).
t_1(W,Y,U) :- t_0(W,T), f(T,Y,U).
t_1(W,Y,V) :- t_0(W,T), g(T,Y,V).
t_2(W,Y,U) :- t_1(W,T,U), h(T,Y,U).
t_2(W,Y,U,V) :- t_1(W,T,V), h(T,Y,U).
t_3(W,Y,U,V) :- t_2(W,T,U), i(T,Y,V).
t_3(W,Y,U,V) :- t_2(W,T,U,V), i(T,Y,V).
```

Suppose our EDB consisted of the following facts F_G:

```
f(c1,c2,b).
g(c1,c2,a).
h(c2,c3,a).
i(c3,c4,b).
```

To maintain equivalence with the RE-translation, the bindings of variable U must be the same for f and h in all answers, and similarly for variable V in g and i.

Therefore, we would expect that there would be no values satisfying the query in Figure 2.11 when applied to the database graph G represented by the facts F_G. This is evident if one considers the third argument in each fact, corresponding to an edge variable. This is different for f and h, as well as for g and i, which does not satisfy our interpretation of the semantics.

However, we find that when we evaluate program Q_2 on the EDB F_G, we obtain the bindings (X1/c1, X2/c4, U/a, V/b). This is because for the rules with head predicate t_1, we propagate variables U and V into the same (third) argument in the predicate. There is no way of distinguishing or separating these variables, and any rules which contain this predicate in the body instantiate all values for U and V in the third argument.

It is due to this semantic problem that we include the propagated variable names into the predicate name during edge variable propagation. This ensures that predicates containing different variable names in the same argument are separated, resulting in the correct query semantics.

■

The RE-translation generally generates fewer rules than the NFA-translation. This is largely due to the method of variable propagation, which results in multiple rules being created

for each variable that is propagated. In the worst case, the number of these rules could be exponential. However, since most queries tend to be quite small and involve a few edge variables, this is rarely problematic.

Example 2.21 Suppose that we translate the common ancestor query in Figure 2.12 on page 34 using the NFA-translation. This produces the following Datalog program:

```
t_0(linda).
t_1(Y) :- t_0(T), par(Y,T).
t_1(Y) :- t_1(T), par(Y,T).
t_2(Y) :- t_1(T), par(T,Y).
t_2(Y) :- t_2(T), par(T,Y).
common_anc(linda,Y) :- t_2(Y).
```

Every IDB predicate in this program is factored by the NFA-translation and evaluation starts at the constant linda in the first rule. The equivalent query, in Figure 2.10 on page 30, could not be context-rewritten or factored using the standard techniques described in [KRS90] or [NRSU89b].

■

Chapter 3

Graph Semantics

There is a need to formalize the semantics of visual queries based on an intuitive meaning. The semantics of GraphLog were originally defined in terms of the RE-translation to Datalog. Therefore, although well-defined, these Datalog semantics do not necessarily derive results that correspond to an intuitive understanding for all queries.

As a parallel, when discussing the definition of semantics for Datalog, Ullman says:

> "...a purely operational definition of meaning for rules, "the program means whatever it is that this interpreter I've written does," is not acceptable ..."
>
> [Ull88, Page 99]

In this chapter, we describe an alternative formulation of meaning for GraphLog queries by defining "graph semantics". Because the intuitive nature of a graph structure is important to GraphLog, and all queries are expressed in terms of patterns in graphs, the graph semantics of GraphLog will also be defined in terms of patterns in a graph structure. We provide background details in Section 3.1, and define the graph semantics in Section 3.2.

For various GraphLog queries, the "standard" semantics (defined by the Datalog semantics of the RE-translation) are compared to the graph semantics in Section 3.3. These results are not identical which suggests that the RE-translation needs to be modified in order to match the graph semantics.

Such a change to the current GraphLog translation is described in Section 3.4. We also propose that the graph semantic meaning of queries defined in GraphLog is equivalent to the Datalog semantics of the query generated by the modified RE-translation, and prove that this is true for a restricted set of GraphLog queries.

3.1 Background

The intuitive meaning of GraphLog suggested in [Con89] is that "GraphLog queries are expressed in a way that suggests the graph patterns that must be present or absent in a database when represented as a directed labeled multigraph". This differs from the actual semantics of a GraphLog query, which are defined in terms of the RE-translation of the query to Datalog. The Datalog query is invoked on a Datalog database, defined by a translation from a database graph to Datalog. The results of the query, generated as a collection of Datalog facts are then re-interpreted as a graph. While an informal understanding of the meaning of visual queries is generally sufficient, it would be appealing for the formal and informal meanings to converge. This is possible by defining graph semantics in terms of queries describing patterns on a database graph.

This is the approach used for the GRE visual query language discussed in Section 2.2. The semantics of GRE are independent of Datalog, but are instead defined in terms of patterns on a database graph. This definition satisfies the intuitive interpretation mentioned above. A translation to Datalog is described for GRE, but no reference is made to this translation or Datalog when defining the meaning of a visual query.

However, GRE does not permit certain queries expressible in monotone GraphLog, including queries that contain multiple edges. In addition, aspects of the GRE graph semantic definition are not complete, for example, the generation of the ϵ string for kleene closure operators is not considered. In this chapter, we resolve these issues, defining graph semantics, similar to those described in [Woo90], for monotone GraphLog.

One would expect that the Datalog semantics of the RE-translation and the graph semantics coincide, since visual queries are created based on the intuitive notion of patterns on database graphs. However, it appears that this is not the case.

Therefore changes to restrict the domain of the nodes are required so that the Datalog semantics of the RE-translation do coincide with the graph semantics. Hence, for any graph query Q and any database graph G, the results of applying the query to the database graph, as defined by the graph semantics, correspond to the results produced by the Datalog query produced by the RE-translation applied to the translated database graph. Here, "corresponds" means that the results in a graph format, when translated to Datalog, are identical to the results of the Datalog query and vice-versa.

3.2 Defining Graph Semantics

We first extend the definition of a db-graph (Definition 2.6 on page 12) by defining an explicit representation where every edge in the db-graph has a symmetric reverse edge, which is traversed from the sink node to the source node. This is done because queries containing expressions that require edges to be traversed from the sink node to the source node are possible in GraphLog. The edge label of reverse edges is prefixed by the "–" symbol to differentiate such edges.

While not altering the meaning of the db-graph, the extension allows a simpler, more concise description of various definitions in this section. There are alternatives which do not require any change to the db-graph, but these are less elegant.

Definition 3.1 An *expanded db-graph* is a db-graph where for every edge $e_i \in E$ with source node N_1, sink node N_2 and label L_i, there is an edge e_i' with source node N_2, sink node N_1, and label $-L_i$. Each such edge e_i' will be referred to as a *reverse edge.*

□

Definition 3.2 [Path]

Let G be an expanded db-graph. A *path* p in G is the null path (or ϵ-path) from any vertex v_i to itself (v_i, v_i, ϵ) or a sequence of one or more edges $e_1, e_2, \ldots, e_n$, where $e_i = (v_i, v_{i+1}, l_i) \in E$. The sequence p denotes a path from the source node of e_1 to the sink node of e_n. We define the source node of p to be the source node of e_1 and the sink node of p to be the sink node of e_n. The length of p is n. The length of the null path is 0.

The *path label* of path p is the string $\lambda(e_1)\ldots\lambda(e_n)$, denoted by $\lambda(p)$, where λ is the edge labelling function from Definition 2.1 on page 10.

The string ϵ is the path label of null paths in G.

□

Note that the definition of a path includes a path of length 0, the null or ϵ-path defined for every node in the db-graph and labelled with the empty string ϵ. This is distinct from a path from a node directly back to itself with an edge label in E, which would have a length of 1. Similar definitions may be found in [Lev80] and [Baa88].

The existence of the ϵ-path for each node in a graph is extremely important in the definition of graph semantics, and resolves some of the intuitive problems with the current GraphLog translation, which will be discussed in Section 3.3.

Definition 3.3 A ***substitution*** θ is a finite set of the form $\{v_1/t_1, \ldots, v_n/t_n\}$, where each v_i is a distinct variable and each t_i is a term. Each element v_i/t_i is called a ***binding*** for v_i [Llo87].

□

Definition 3.4 Let $\theta = \{v_1/t_1, \ldots, v_n/t_n\}$ be a substitution. Consider expression $p_r = (X, Y, Z)$ where X and Y are node labels, and $Z \in L(R)$ where R is an edge label, which is an extended regular expression (Definition 2.4). Then $p_r\theta$, the *instance* of p_r by θ, is the expression obtained from p_r by simultaneously replacing each occurrence of the variable v_i in p_r by the term t_i $(i = 1, \ldots, n)$.

□

Definition 3.5 Consider expression $p_r = (X, Y, Z)$ where X and Y are node labels and $Z \in L(R)$ for edge label R. We will call tuple p_r a ***path pattern***. We denote the set of path patterns $P_r = \{(X, Y, Z) \mid Z \in L(R)\}$.

□

Definition 3.6 Consider the set of paths P in expanded db-graph G, a substitution θ and a path pattern $p_r = (X, Y, Z) \in P_r$. Consider a path $p \in P$ where X_p,Y_p and $\lambda(p)$ are the source node, sink node and path label of p, respectively. We say that $p_G = (X_p, Y_p, \lambda(p))$ *matches* $p_s = p_r\theta$ if p_G is identical to p_s.

□

The following definition of the semantics of define queries considers a GraphLog query set without negation (*monotone* GraphLog) and does not include extensions such as negation, evaluable, arithmetic or other function symbols.

Definition 3.7 Let Q_D be a define query with distinguished edge D and G be a db-graph. Let G' be the expanded db-graph, constructed from G. Assume that Q_D has m edges $E_1, \ldots, E_m$.

Let p be a path in G' with label $\lambda(p)$, as defined above. Let the source and sink node of p be labelled p_x and p_y, respectively.

Let E_i $(1 \leq i \leq m)$ be an edge of Q_D with source label X_i, sink label Y_i, each corresponding to a term, and edge label R_i, i.e. $E_i = (X_i, Y_i, R_i)$. Let $P_{ri} = \{(X_i, Y_i, Z) \mid E_i = (X_i, Y_i, R_i) \wedge Z \in L(R_i)\}$ be a set of path patterns.[1]

Let U be the set of constants and V be the set of variables over which terms and labels are defined. We consider θ, a substitution where each term $t_i \in U \cup V$. We call θ an *answer substitution* if, for path pattern $p_r \in P_{ri}$, $(p_x, p_y, \lambda(p))$ matches $p_r\theta$.

Suppose distinguished edge D in Q_D is represented as a tuple $(X_D, Y_D, p(\tilde{Z}_D))$, where X_D is the source node label, Y_D is the sink node label and $p(\tilde{Z}_D)$ is the edge label, with predicate name p and a (possibly empty) set of terms $\tilde{Z}_D$. We denote by $\rho(D)$ the application of substitution ρ to each component of D, that is, $(\rho(X_D), \rho(Y_D), p(\rho(\tilde{Z}_D)))$.

The result of query Q_D on db-graph G is defined as an *answer graph* which is also a db-graph, $Q_D(G)$, with edge set:

$$\{\rho(D) \mid \forall E_i \ (1 \leq i \leq m) \ \theta_i \text{ is an answer substitution for } E_i,$$
$$\text{and } \rho = \bigcup_{i=1}^{m} \theta_i \text{ is also a substitution } \}$$

Every variable in D which is not bound to a constant by ρ is called an *unbound* variable and denoted with an underscore ("_") symbol.

□

The above result defines the *graph semantics* of a visual query. We are finding paths in the db-graph that match strings in the language defined by the edge regular expressions of the query, with the substitution of variables by constants or other variables. These bindings are then used to construct an answer graph, in a format similar to the distinguished edge. Further queries may be composed on this answer graph by regarding it as a db-graph. Such query composition is a powerful visualization tool, since it allows hierarchical construction of queries and displaying of information.

For the graph semantics, unbound variables can only occur for edge variables on the distinguished edge, since paths in the db-graph always start at a bound value and end at a bound value (assuming that the db-graph only contains bound values). We can safely define unbound variables to mean whatever we decide, because the graph semantics are not defined in terms of existing semantics. However, in order to show that the graph semantics

[1]The anonymous variable "_" is handled by treating each instance of it in $L(R_i)$ as a separate variable.

are equivalent to the Datalog semantics of queries generated the RE-translation, we can equate unbound variables to Datalog unbound variables.

The safety of Datalog queries has been introduced in Section 2.3.3. The problem of safety involving unbound variables resulting from the evaluation of GraphLog queries has been examined in [Con89], as well as in [Fuk91]. We shall examine safety related to the translation method further in Chapter 4.

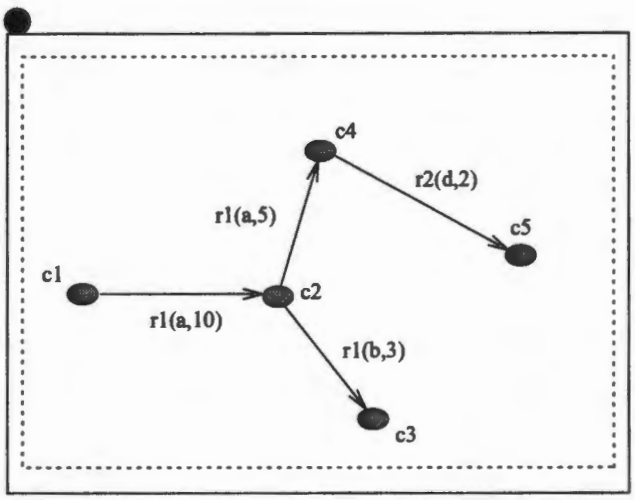

Figure 3.1: Database graph G_C

Example 3.1 The db-graph G_C in Figure 3.1 represents the synthesis of chemical compounds using various reactions. The edge labelled **r1(a,10)** represents some reaction **r1**, using some process **a** which produces compound **c2** from compound **c1** after 10 minutes.

The query Q_C, in Figure 3.2, applied to db-graph G_C will be used to illustrate the graph semantics. Clearly, each path in the expanded db-graph G'_C that contains a reverse edge will not satisfy Q_C, since there are no reverse literals in the edge labels of the query. Therefore, we will not consider paths with reverse edges in this example.

Paths in G'_C are therefore as follows :

p_1 $(c1, c1, \epsilon)$ p_2 $(c2, c2, \epsilon)$

p_3 $(c3, c3, \epsilon)$ p_4 $(c4, c4, \epsilon)$

p_5 $(c5, c5, \epsilon)$ p_6 $(c1, c2, r1(a, 10))$

p_7 $(c2, c3, r1(b, 3))$ p_8 $(c2, c4, r1(a, 5))$

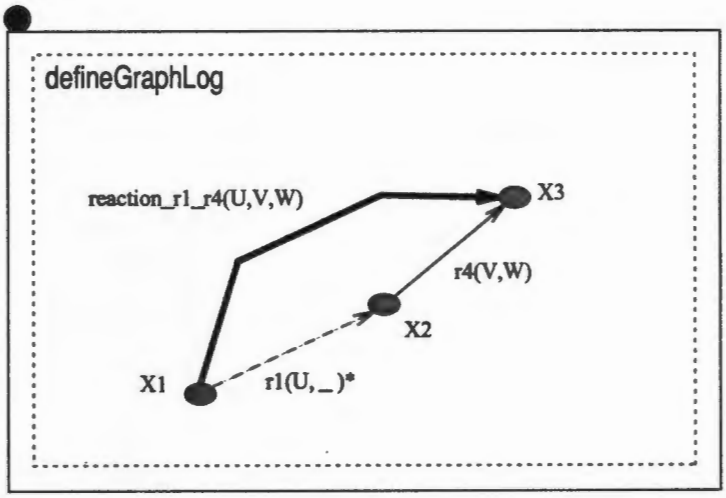

Figure 3.2: Chemical Reaction Query Q_C

p_9 $(c4, c5, r2(d, 2))$ p_{10} $(c1, c4, r1(a, 10)\ r1(a, 5))$
p_{11} $(c2, c5, r1(a, 5)\ r2(d, 2))$ p_{12} $(c1, c3, r1(a, 10)\ r1(b, 3))$
p_{13} $(c1, c5, r1(a, 10)\ r1(a, 5)\ r2(d, 2))$

where each tuple p_i represents (source node, sink node, path label).

Consider the first edge E_1 in query Q_C, namely, that from $X1$ to $X2$ labelled $r1(a,\_)*$. We generate the following set : $ER_1 = \{(X_1, X_2, Z) \mid Z \in L(r1(U,\_)*)\}$.

Elements in ER_1 include :

$e_{1.1}$ (X_1, X_2, ϵ)

$e_{1.2}$ ($X_1, X_2, r1(U,\_)$)

$e_{1.3}$ ($X_1, X_2, r1(U,\_)\ r1(U,\_)$)

...

We see that path p_1 matches an instance of $e_{1.1}$, with the substitution $\theta = \{X_1/c1, X_2/c1\}$. Therefore θ is an answer substitution. Similarly for paths p_2 to p_5 above.

For element $e_{1.2}$, it is clear that only paths p_6, p_7 and p_8 match an instance of $e_{1.2}$.

For element $e_{1.3}$, only path p_{10} matches an instance of $e_{1.3}$. Path p_{12} does not qualify, since in this case $\theta = \{X_1/c1, X_2/c2, U/a, U/b, \_T_1/10, \_T_2/3\}$[2]. Clearly, θ is not a valid substitution because of the different bindings for variable U.

We do not have to consider any more elements since there are no paths of length 3 or more with all edges labelled "r1".

Therefore, the set of substitutions θ_1 derived from paths that satisfy the first edge in query Q_C are :

1. $\theta_1 = \{X_1/c1, X_2/c1\}$
2. $\theta_1 = \{X_1/c2, X_2/c2\}$
3. $\theta_1 = \{X_1/c3, X_2/c3\}$
4. $\theta_1 = \{X_1/c4, X_2/c4\}$
5. $\theta_1 = \{X_1/c5, X_2/c5\}$
6. $\theta_1 = \{X_1/c1, X_2/c2, U/a, \_T_1/10\}$
7. $\theta_1 = \{X_1/c2, X_2/c3, U/b, \_T_1/3\}$
8. $\theta_1 = \{X_1/c2, X_2/c4, U/a, \_T_1/5\}$
9. $\theta_1 = \{X_1/c4, X_2/c5, U/d, \_T_1/2\}$
10. $\theta_1 = \{X_1/c1, X_2/c4, U/a, \_T_1/10, \_T_2/5\}$

Similarly we consider the second edge E_2 in Q_C. We find that ER_2 is a single element :

$e_{2.1}$ (X_2, X_3,r4(V,W))

The only path that matches an instance of $e_{2.1}$ is path p_9. Here the substitution is $\theta = \{X_2/c4, X_3/c5, V/d, W/2\}$.

Therefore, the set of substitutions θ_2 derived from paths that satisfy the second edge in query Q_C is:

$\theta_2 = \{X_2/c4, X_3/c5, V/d, W/2\}$.

[2]Each underscore symbol in the tuple is treated as a separate variable, i.e. $\_T_1$ and $\_T_2$.

We can form the answer to the query by finding the union of all answer substitutions θ_i that are valid substitutions.

Suppose we take the union of $\theta_1 = \{X_1/c1, X_2/c1\}$ and $\theta_2 = \{X_2/c4, X_3/c5, V/d, W/2\}$. Clearly the resulting substitution is not valid since we have elements $X_2/c1$ and $X_2/c4$ as part of the set.

However, the union of $\theta_1 = \{X_1/c4, X_2/c4\}$ and $\theta_2 = \{X_2/c4, X_3/c5, V/d, W/2\}$ does form a valid substitution and $\rho = \theta_1 \cup \theta_2 = \{X_1/c4, X_2/c4, X_3/c5, V/d, W/2\}$. Notice that the variable U on the distinguished edge does not have a binding in ρ.

We find A, the union of all substitutions θ_1 and θ_2 which are valid:

1. $\rho = \{X_1/c4, X_2/c4, X_3/c5, V/d, W/2\}$
2. $\rho = \{X_1/c2, X_2/c4, X_3/c5, U/a, V/d, W/2\}$
3. $\rho = \{X_1/c1, X_2/c4, X_3/c5, U/a, V/d, W/2\}$

From this we can derive the answer graph $Q_C(G_C)$ in Figure 3.3, from the set of edges below. The answer may be regarded as a db-graph and used for further querying.

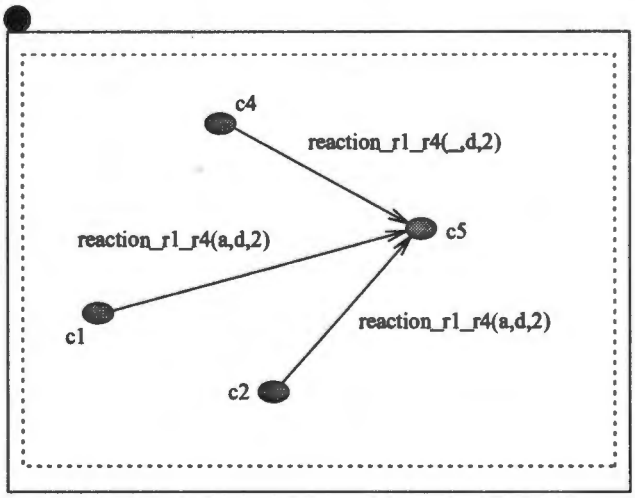

Figure 3.3: Answer graph $Q_C(G_C)$

3.3 Comparison of Graph and RE-translation Semantics

In this section, the graph semantics of various closure query examples are compared with the Datalog semantics produced by the RE-translation. Datalog queries will be used to generate answers to a sample database, displayed in Figure 3.4. It will be shown that certain answers lead to inconsistencies which need to be discussed. The resolution of these inconsistencies is considered in the following section.

As we have mentioned, one of the primary motivations for choosing a graph representation is that it is interpreted in a natural and intuitive manner. Obviously, this natural interpretation should be preserved in all instances by the translation method. Hence, there is a requirement that the Datalog semantics should always be consistent with the graph semantics in the sense that the Datalog program should only produce answers that correspond to the graph interpretation.

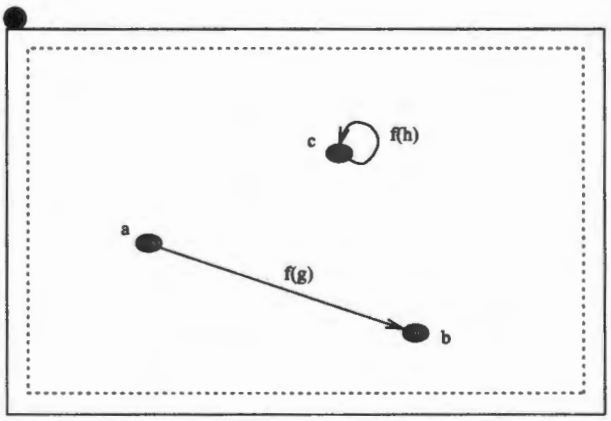

Figure 3.4: Database Graph G_O

Example 3.2 Consider the db-graph G_O of Figure 3.4. Suppose that we generate the expanded db-graph G from G_O. The Datalog facts F_G that correspond to G are :

```
f(a,b,g).
f(c,c,h).
-f(b,a,g).
-f(c,c,h).
```

These facts constitute the database for this discussion.

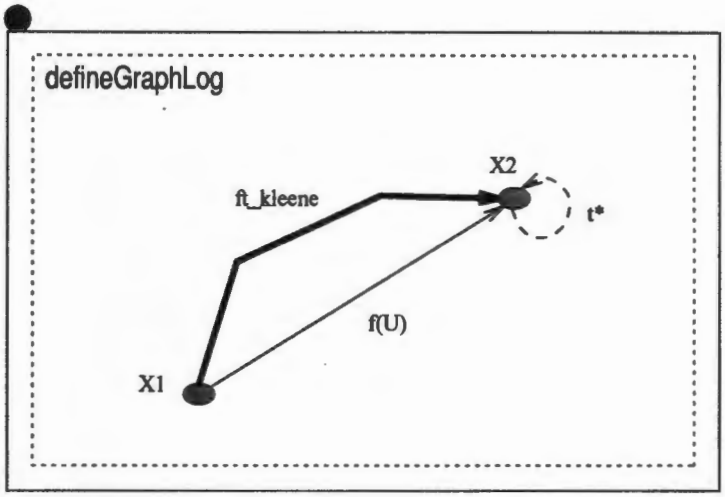

Figure 3.5: Multiple-edge query Q

The query Q in Figure 3.5 is an example where the RE-translation will not compute the expected set of answers.

If one was to generate $L(Q)$, the language described by the regular expressions on the edges of Q, one would derive sequences of labels which should correspond to paths in G as follows: $L(Q) = \{f(U), f(U)t, f(U)tt, \ldots\}$.

The Datalog query P_Q generated by the RE-translation would be :

```
ft_kleene(X1,X2) :- f(X1,X2,U),t_kleene(X2,X2).
t_kleene(X,X) :- t(X,Y).
t_kleene(Y,Y) :- t(X,Y).
t_kleene(X,Y) :- t(X,T), t_kleene(T,Y).
```

If P_Q is evaluated using facts F_G, no answers are generated. This is because there are no `t` predicates in F_G, since there are no t edges in G. Therefore, the `ft_kleene` predicate above is empty.

This result does not correspond to what we would have intuitively expected, which would have included answers derived from all edges that have a label `f`, since this label is in the language $L(Q)$, describing path labels in G for query Q.

We now consider the graph semantic interpretation of this query Q, using expanded db-graph G. The result, according to Definition 3.7 would have included the edges:

```
(a,b,ft_kleene).
(c,c,ft_kleene).
```

■

The RE-translation is essentially the same translation that is presented in [CMV94], and tests on the Hy$^+$ system have confirmed the results presented in the previous example.

The original definition of GraphLog [Con89] defined the Kleene closure operator in terms of the transitive closure operator and the equals operator. This is the definition that is used in [VW95].

Example 3.3 Using the original Kleene closure definition, the above query would be translated as follows :

```
ft_kleene(X1,X2) :- f(X1,X2,U),t_kleene(X2,X2).
t_kleene(X,Y) :- X = Y.
t_kleene(X,Y) :- t(X,T), t_kleene(T,Y).
```

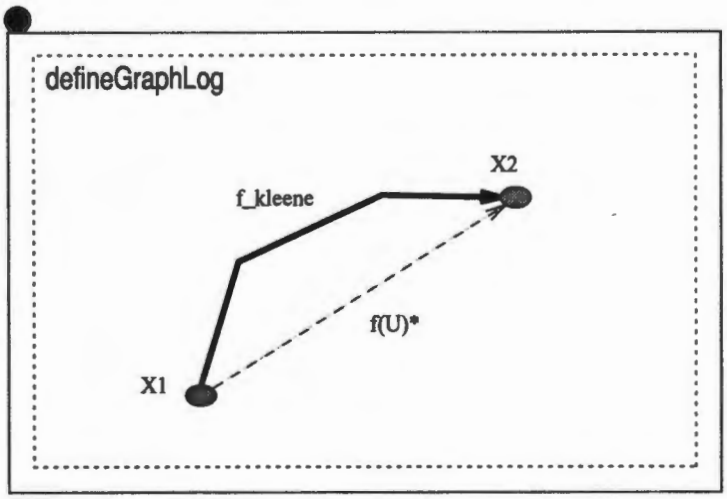

Figure 3.6: Query Q_2

In this case, the translation produces the expected answer. However, for query Q_2, in Figure 3.6, the translation would be:

```
f_kleene(X1,X2) :- kleene(X1,X2,U).
kleene(X,Y,U) :- X = Y.
kleene(X,Y,U) :- f(X,T,U), kleene(T,Y,U).
```

The set of answers produced by applying this Datalog query to the set of facts generated from the expanded db-graph G in the previous example is:

```
f_kleene(c,c).
f_kleene(a,b)
f_kleene(_,_).
```

Notice that the node variables in the third fact remain unbound, due to the second rule in the program. If we use the graph semantics instead, we obtain the result (as a set of distinguished edges):

```
(a,a,f_kleene)
(b,b,f_kleene)
(c,c,f_kleene)
(a,b,f_kleene)
```

Variables `X1` and `X2` remain unbound in the first rule of the program produced by the RE-translation. This is not the same as the results derived from the graph semantics, because the Datalog semantics means that the unbound variables range the entire set of values in the universe, provided the values are equal. This set of values would include edge as well as node constants in the db-graph. We could, for example, infer the fact `f_kleene(e,e)`, which would correspond to a distinguished edge `f_kleene` from a node `e` back to itself. Clearly there are no nodes labelled `e` in the db-graph. In comparison, the graph semantics restrict the values of `X1` and `X2` to node values only, since all paths in the db-graph start and end at a node. Ways of addressing this difference in semantics will be discussed in the next section.

∎

3.4 Modification of GraphLog Semantics

From the discussion in the previous section, it should be clear that the Datalog semantics of the RE-translation and the graph semantics do not coincide in certain instances. In

this section, a proposal will be made for the modification of the RE-translation so that the semantics do coincide.

We will assume that the RE-translation from GraphLog to Datalog is defined as in Section 2.5.

Since a significant aspect of the graph semantics involves defining paths of length 0 which resolves the problems described in the previous section, we need to consider the symbols in GraphLog that involve paths of length 0. These would include the Kleene closure operator and the ϵ symbol.

At the moment, the RE-translation of the Kleene closure operator is:

$$kleene\_e_m(X, X, \bar{F}_m) \text{ :- } e_m(X, Y, \bar{Z}_m).$$

$$kleene\_e_m(Y, Y, \bar{F}_m) \text{ :- } e_m(X, Y, \bar{Z}_m).$$

$$kleene\_e_m(X, Y, \bar{F}_m) \text{ :- } e_m(X, T, \bar{Z}_m), kleene\_e_m(T, Y, \bar{F}_m).$$

where $\bar{F}_m$ contains the named free variables of $\bar{Z}_m$ (which are restricted to have the same value along the path), and T is a variable appearing nowhere else in the rule.

One of the differences between the semantics stems from the presence of predicates in the body of the first two rules, so that a fact can only be derived from the predicate in the rule head if there are facts that satisfy the body predicate. This means that there has to be at least one edge in the db-graph labelled with the predicate defined in the rule body. This is not always true as demonstrated in the discussion of the query in Figure 3.5, and does not give us the results that we expect.

One possible solution is to eliminate the body predicates in the first two rules. This would coincide with the graph semantic requirement that for ϵ, all nodes would be connected to themselves via the predicate $kleene\_e(X, X, \bar{F}_m)$. This is equivalent to the form

$$kleene\_e(X, Y, \bar{F}_m) \text{ :- } X = Y.$$

which was suggested in the original translation of GraphLog [Con89], and has been included in the version of the translation described in [VW95]. This was the approach that was adopted in Example 3.3.

However, as we discovered with Example 3.3, terms in the rule head are unbound, which poses two problems. Firstly, the graph semantics always binds the node variables in the define query to all values corresponding to node labels in the db-graph. Secondly, the Datalog semantics of these variables being unbound means that the predicate kleene_e would be true for all values from the database domain. This would mean that terms like X, which should correspond to node values by virtue of their position in the predicate could represent constant values on the edges of the db-graph. Similarly, terms in $\bar{F}_m$, which should correspond to edge constants, could also theoretically represent values which are in the node domain. A practical consideration is the representation of such unbound node variables in a visual environment.

Of course, we could redefine the context in which unbound variables are viewed in GraphLog as compared to Datalog. This would mean that there would be an implicit restriction on nodes and edges, so that unbound node variables would only range over the universe of node variables, whereas unbound edge variables would range over the universe of edge variables. From a practical perspective, making this assumption might be justifiable, since the subtleties of such a domain distinction may well be lost on the user, who is usually more concerned with ease-of-use and the power of the system rather than perceived minor semantic differences.

Nevertheless, since the problem is of theoretical interest, an attempt to define a domain restrictive translation that does not result in unbound node variables will be made. The problem stems from the ϵ-string being one of the elements in the language generated by the regular expressions in the query. The approach that has been taken by the graph semantics to resolve this issue is to define a path of zero length existing from any node to itself. Therefore, such paths appear in the answer to queries containing the ϵ-string. If we were to leave node variables unbound, besides not restricting variables contained in nodes to node values, we would not have a representation in the answer graph for these zero-length paths.

Therefore, by using the translation of the Kleene closure with a rule containing the equals subgoal, the RE-translation semantics still does not correspond to the graph semantics. What is required is some way of specifying that only constants in *nodes* are instantiated by the terms in the predicate that correspond to nodes and similarly for edges.

A solution to the problem of restricting node variables to the node domain is to separate all nodes in the database by defining a "node fact" for every node in the database. This would require an extension to Definition 2.14 as follows:

Definition 3.8 For every node $n \in N$ with label X, we generate a fact $node(X)$. We say

that each fact *node*(X) *corresponds* to node $n \in N$ with label X.

□

Definition 3.9 We redefine the RE-translation of the Kleene closure operator as:

Label $e_m(\bar{Z}_m)*$ corresponds to the rules

kleene_e$(X, X, \bar{F}_m)$:- *node*(X).

kleene_e$(X, Y, \bar{F}_m)$:- $e_m(X, T, \bar{Z}_m)$, *kleene_e*$(T, Y, \bar{F}_m)$.

where $\bar{F}_m$ contains the named free variables of $\bar{Z}_m$ (which are restricted to have the same value along the path), and T is a variable appearing nowhere else in the rule.

In addition, we define a translation of the symbol ϵ as:

epsilon_e(X, X) :- *node*(X).

□

We restrict the domain of the nodes because the graph semantics does not result in unbound variables for nodes. However, edge variables can be unbound in graph semantics, so we do not enforce the edge domain restriction explicitly in the RE-translation. This can be regarded as a compromise between unifying theoretical with intuitive meaning, and ensuring that the graph and Datalog semantics are similar.

The diagram in Figure 3.7 shows the inter-relationships between the graph semantics and the Datalog semantics defined by the RE-translation. The left half of the diagram represents the semantics from a graph perspective whereas the right half represents semantics at the Datalog level. The node G is the db-graph and when query Q is evaluated on it, it produces a set of answers $Q(G)$ according to the graph semantics. This is depicted by the transition labelled Q from G to $Q(G)$. The node F_G represents the set of facts corresponding to db-graph G, represented by transition I. The Datalog program P_Q produced from query Q by the RE-translation, when evaluated on F_G, generates a set of answer facts $P_Q(F_G)$. These facts are defined by the Datalog semantics. If the set of results $Q(G)$, according to the graph semantics, corresponds to the set of Datalog answer facts $P_Q(F_G)$, then the transition I between them holds.

We will now prove that transition I holds for a restricted set of GraphLog queries, in other words, $Q(G)$ corresponds to $P_Q(F_G)$. We consider only monotone queries that are single-edge. The result can be extended for multiple-edge monotone queries as well. A related proof for simple queries can be found in [Woo88].

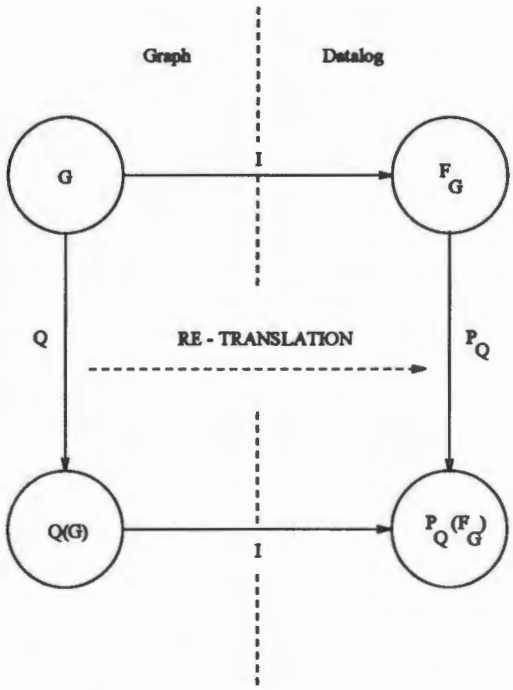

Figure 3.7: Relation Diagram

Theorem 3.1 [Equivalence of Graph and Modified RE-translation Semantics]

Let Q be a monotone GraphLog query with a single edge from X to Y labelled with regular expression r which uses variables $\bar{Z}$. Let P_r be the Datalog program produced by the modified translation described above. Let G be a database graph and F_G the corresponding set of facts. Fact $r(\theta(X), \theta(Y), \theta(\bar{Z}))$ is derived by P_r on F_G iff there is a path p from $\theta(X)$ to $\theta(Y)$ in G satisfying $\theta(r)$, where θ is an answer substitution.

Proof:

By induction on the number of operators in the regular expression labelling the edge in Q.

Basis: Zero Operators

Regular expression r_0 is either the null string ϵ, a forward literal $p(T_1, \ldots, T_m)$ or a reverse literal $-p(T_1, \ldots, T_m)$. We consider each fact in $P_{r_0}(F_G)$ and show that there is a path p_0 in G that satisfies $\theta(r_0)$ and vice-versa.

Base Case 1

If $r_0 = \epsilon$, $L(r_0) = \{\epsilon\}$.

The Datalog program P_{r_0} is as follows:

$$r_0(X, X) \text{ :- } node(X).$$

Every fact $f_a = r_0(c_1, c_1)$ in $P_{r_0}(F_G)$ is derived from a fact $node(c_1)$ in F_G. From Definition 2.14, there is a node labelled c_1 in G and hence a path p_0 in G from node c_1 to itself denoted by tuple (c_1, c_1, ϵ).

Conversely, we consider a path $p_0 = (c_1, c_2, \epsilon)$ in G, i.e., from c_1 to c_2 satisfying $\theta(r_0)$. The only paths labelled with ϵ are paths of length 0 (by definition), where $Y = X$. (Definition 3.2) Therefore, for p_0 in G, $c_1 = c_2$.

For the ϵ-path from node c_1 in G, there is a corresponding fact $node(c_1)$ in F_G by Definition 2.14. The Datalog semantics for program P_{r_0} define a fact $r_0(c_1, c_1)$ derived from fact $node(c_1)$ in F_G.

Base Case 2

If r_0 is the predicate $p(\bar{Z})$, $L(r_0) = \{p(\bar{Z})\}$.

The Datalog program P_{r_0} defined for the query is:

$$r_0(X, Y, \bar{Z}) \text{ :- } p(X, Y, \bar{Z}).$$

Consider some fact $f_a = r_0(c_1, c_2, \bar{c_3})$ derived from a fact $p(c_1, c_2, \bar{c_3})$ in F_G, where $\bar{c_3}$ is a tuple of terms. From Definition 2.14, there is a path p_0 consisting of edge e_o in G with label $p(\bar{c_3})$ from source node c_1 to sink node c_2, denoted $(c_1, c_2, p(\bar{c_3}))$.

Conversely, we consider a path $p_0 = (c_1, c_2, p(\bar{c_3}))$ in G. The only path that has a single forward literal as a label is an edge $e_o = (c_1, c_2, p(\bar{c_3}))$ in G (from definition of a label).

Fact $p(c_1, c_2, \bar{c_3})$ in F_G corresponds to edge e_o in G by Definition 2.14. Thus $P_{r_0}(F_G)$ contains fact $r_0(c_1, c_2, \bar{c_3})$.

Base Case 3

Assume r_0 is the predicate $-p(\bar{Z})$. $L(r_0) = \{-p(\bar{Z})\}$. The Datalog program P_{r_0} defined for the query is :

$$r_0(X, Y, \bar{Z}) \text{ :- } p(Y, X, \bar{Z}).$$

We use similar reasoning to the second base case. Consider fact $f_a = r_0(c_1, c_2, \bar{c_3})$, derived by fact $f_o = p(c_2, c_1, \bar{c_3})$ in F_G, where $\bar{c_3}$ is a tuple of terms. From Definition 2.14, f_o corresponds to edge $e_o = (c_2, c_1, p(\bar{c_3}))$ which is a path in G.

Conversely, we consider a path $p_0 = (c_1, c_2, -p(\bar{c_3}))$ in G. This is equivalent to a path from c_2 to c_1 with label $p(\bar{c_3})$. The only path in G that has a single predicate as a label is an edge $e_o = (c_2, c_1, p(\bar{c_3}))$ (from definition of a label). Fact $p(c_2, c_1, \bar{c_3})$ in F_G corresponds to edge e_o in G by Definition 2.14. Thus $P_{r_0}(F_G)$ contains fact $f_a = r_0(c_1, c_2, \bar{c_3})$.

Inductive Hypothesis:

We assume that for any regular expression r_n with n or fewer operators, Datalog program P_{r_n} on F_G derives a fact $r_n(\theta(X), \theta(Y), \theta(\bar{Z}))$ iff there is a path p_n from $\theta(X)$ to $\theta(Y)$ in G satisfying $\theta(r_n)$ where θ is an answer substitution.

Inductive Step:

Consider regular expression r_{n+1} with $n+1$ operators, where $r_{n+1} = r_i \cdot r_j$, $r_{n+1} = r_i \mid r_j$ or $r_{n+1} = r_i*$. We do not consider the transitive closure case, because such queries can be constructed using the other operators.

Induction Case 1: Alternation

Assume that r_{n+1} is the expression $r_i \mid r_j$. Let $Q = (X, Y, r_{n+1})$. $L(r_{n+1}) = L(r_i) \cup L(r_j)$. The Datalog program $P_{r_{n+1}}$ produced by the translation is:

$$r_{n+1}(X, Y, \bar{F_i}, \bar{F_j}) \text{ :- } r_i(X, Y, \bar{Z_i}).$$

$$r_{n+1}(X, Y, \bar{F_i}, \bar{F_j}) \text{ :- } r_j(X, Y, \bar{Z_j}).$$

A fact $f_a = r_{n+1}(c_1, c_2, \bar{c_3}, \bar{c_4})$ is derived from some fact $f_{o1} = r_i(c_1, c_2, \bar{b_1})$ or some fact $f_{o2} = r_j(c_1, c_2, \bar{b_2})$ in F_G.[3] Without loss of generality, we assume that f_a is derived from f_{o2}. From the inductive hypothesis, we know that there is there is a path p from c_1 to c_2 in G where $\lambda(p) \in L(\theta(r_i))$, where θ is an answer substitution such that $\theta(X) = c_1$, $\theta(Y) = c_2$ and $\theta(\bar{Z_i}) = \bar{b_2}$. Hence p satisfies $\theta(r_{n+1})$.

Conversely, we show that if there is a path in G satisfying $\theta(r_{n+1})$, then there is a corresponding r_{n+1} fact derived by program $P_{r_{n+1}}$.

Let θ be an answer substitution such that there is a path p from c_1 to c_2 in G where $\lambda(p) \in L(\theta(r_{n+1}))$, i.e., p satisfies $\theta(r_{n+1})$. Since $r_{n+1} = r_i | r_j$, p must satisfy $\theta(r_i)$ or $\theta(r_j)$. Without loss of generality, assume p satisfies $\theta(r_i)$ where $\bar{Z}$ includes all variables in r_i. By the inductive hypothesis, $r_i(c_1, c_2, \theta(\bar{Z_i}))$ is derived by P_{r_i}. Hence, $r_{n+1}(c_1, c_2, \theta(\bar{F_i}), \theta(\bar{F_j}))$ is derived by $P_{r_{n+1}}$.

Induction Case 2: Concatenation

Assume that r_{n+1} is the expression $r_i \cdot r_j$. Let $Q = (X, Y, r_{n+1})$. $L(r_{n+1}) = L(r_i)L(r_j)$.

The Datalog program $P_{r_{n+1}}$ produced is:

$$r_{n+1}(X, Y, \bar{F_i}, \bar{F_j}) \text{ :- } r_i(X, T, \bar{Z_i}), r_j(T, Y, \bar{Z_j}).$$

Consider fact $f_a = r_{n+1}(c_1, c_2, \bar{c_3})$, derived from some facts $f_{o1} = e_m(c_1, t_1, \bar{b_1})$ and $f_{o2} = e_n(t_1, c_2, \bar{b_2})$ in F_G. From the inductive hypothesis, we know that there is there is a path p_i from c_1 to t_1 in G where $\lambda(p) \in L(\theta_i(r_i))$, where θ_i is an answer substitution. Similarly, there is a path p_j from t_1 to c_2 where $\lambda(p) \in L(\theta_j(r_j))$. So there is a path p in G from c_1 to c_2. Since θ_i and θ_j must agree on common variables, $\theta = \theta_i \cup \theta_j$ is an answer substitution. Hence p satisfies $\theta(r_{n+1})$.

Conversely, we show that if there is a path in G satisfying r_{n+1}, then there is a corresponding r_{n+1} fact derived by program $P_{r_{n+1}}$.

Let θ be an answer substitution such that there is a path p from c_1 to c_2 in G where $\lambda(p) \in L(\theta(r_{n+1}))$, i.e., p satisfies $\theta(r_{n+1})$. Since $r_{n+1} = r_i . r_j$, there must be a path p_i from c_1 to t_1 satisfying $\theta_i(r_i)$, and p_j from t_1 to c_2 satisfying $\theta(r_j)$, where θ_i and θ_j are θ restricted to the variables appearing in r_i and r_j , that is $\bar{Z_i}$ and $\bar{Z_j}$, respectively. By the inductive hypothesis, $r_i(c_1, t_1, \theta(\bar{Z_i}))$ is derived by P_{r_i}. Similarly, $r_j(t_1, c_1, \theta(\bar{Z_j}))$ is derived by P_{r_j}. Hence, $r_{n+1}(c_1, c_2, \theta(\bar{F_i}), \theta(\bar{F_j}))$ is derived by $P_{r_{n+1}}$.

[3] Some of the terms in $\bar{c_3}$ may be unbound variables, if $\bar{Z_i}$ and $\bar{Z_j}$ contain different variables.

Induction Case 3: Kleene Closure

Assume that r_{n+1} is the expression r_i*. Let $Q = (X, Y, r_{n+1})$. $L(r_{n+1}) = \{\epsilon \cup L(r_i) \cup L(r_i)L(r_i) \cup \ldots\}$. We shall refer to the number of r_i substrings in each string in $L(r_n)$ as the *expression length*, with the ϵ-string having length 0.

The Datalog program $P_{r_{n+1}}$ produced for Q is:

$$r_{n+1}(X, X, \bar{F}_i) \text{ :- } node(X).$$

$$r_{n+1}(X, Y, \bar{F}_i) \text{ :- } r_i(X, T, \bar{Z}_i), r_{n+1}(T, Y, \bar{F}_i).$$

We will first prove that for a fact f in $P_{r_{n+1}}(F_G)$, generated after m iterations of $P_{r_{n+1}}$, there is a path p in G satisfying $\theta(r_{n+1})$ by a further induction on the number of times r_i is "iterated" and the number of times the recursive rule is iterated.

Basis: Suppose that $m = 0$. We need to show that for fact f_0 in F_G, there is a path p_0 in G satisfying $\theta(r_{n+1})$, where θ is an answer substitution..

Since $m = 0$, there are no iterations of the recursive rule. Therefore, fact $f_0 = r_{n+1}(c_1, c_2, \bar{c}_3)$ can only be derived from a fact $node(c_1)$ in the first rule of $P_{r_{n+1}}$, where $c_1 = c_2$. From Definition 2.14, there is a node labelled c_1 in G and hence a path p_0 in G from node c_1 to itself denoted by tuple (c_1, c_1, ϵ).

Inductive Hypothesis: Assume that there is a path p_m in G from c_1 to c_2 satisfying $\theta(r_{n+1})$ for fact $f_m = r_{n+1}(c_1, c_2, \bar{c}_3)$ in $P_{r_{n+1}}(F_G)$, generated after m iterations of $P_{r_{n+1}}$. We will show that there is a path p_{m+1} in G for fact $f_{m+1} = r_{n+1}(c_1, c_2, \bar{c}_3)$ in $P_{r_{n+1}}(F_G)$, generated after $m + 1$ iterations of $P_{r_{n+1}}$.

Fact $f_{m+1} = r_{n+1}(c_1, c_2, \bar{c}_3)$ is derived in the recursive rule from some facts $r_i(c_1, t_1, \bar{c}_3)$ and $r_{n+1}(t_1, c_2, \bar{c}_3)$. Using similar reasoning to the first part of induction case 2 (concatenation), we deduce from our main inductive hypothesis that there is a path p_i from c_1 to t_1 in G where $\lambda(p_i) \in L(\theta_i(r_i))$, where θ_i is an answer substitution. From our inductive hypothesis for this case, there is a path p_m in G from t_1 to c_2 where $\lambda(p_m) \in L(\theta_j(r_{n+1}))$. Hence, there is a path p_{m+1} in G from c_1 to c_2. Since θ_i and θ_j agree on T and are otherwise defined for the same variables $\bar{F}_i$ and are otherwise defined for the same variables $\bar{F}_i$, $\theta = \theta_i \cup \theta_j$ is also an answer substitution. Hence p_{m+1} satisfies $\theta(r_{n+1})$, i.e., $\lambda(p_{m+1}) \in L(\theta(r_{n+1}))$.

Conversely, we show that if there is a path p of length m in G satisfying $\theta(r_{n+1})$, then there is a fact r_{n+1} derived by program $P_{r_{n+1}}$ generated after m iterations. Again, this is done

by a further induction on the number of times r_i is "iterated" and the number of times the recursive rule is iterated.

Basis: Suppose that $m = 0$. We need to show that for path p_0 in G satisfying $\theta(r_{n+1})$, there is a fact f_0 in $P_{r_{n+1}}(F_G)$. Path p_0 is derived from some ϵ-path in G, where $c_1 = c_2$. For this node c_1, there is a corresponding fact node(c_1, c_1) in F_G by Definition 2.14. From the first rule in $P_{r_{n+1}}$, we can derive a fact $r_{n+1}(c_1, c_1, \bar{c_3})$, where $\bar{c_3}$ is a tuple of unbound edge variables.

Inductive Hypothesis: Assume that there is a fact $f_m = r_{n+1}(c_1, c_2, \bar{c_3})$ in $P_{r_{n+1}}(F_G)$, for path p_m from c_1 to c_2 in G of length m satisfying $\theta(r_{n+1})$, where θ is an answer substitution. We will show that there is a fact $f_{m+1} = r_{n+1}(c_1, c_2, \bar{c_3})$ in $P_{r_{n+1}}(F_G)$, generated after $m+1$ iterations of $P_{r_{n+1}}$, for path p_{m+1} from c_1 to c_2 in G.

Let θ be an answer substitution such that there is a path p of $m+1$ iterations of r_i from c_1 to c_2 in G where $\lambda(p) \in L(\theta(r_{n+1}))$, i.e., p satisfies $\theta(r_{n+1})$. Using similar reasoning to the second part of induction case 2 (concatenation), we can divide this path p into 2 sub-paths in G, namely p_1, from c_1 to t_1 where p_1 satisfies $\theta_j(r_j)$, and p_m, from t_1 to c_2 where p_m satisfies $\theta_j(r_{n+1})$. Both θ_i and θ_j are restricted to the the variables appearing in r_i and r_{n+1}, that is $\bar{Z}_i$.[4]

From the main inductive hypothesis, we can derive a fact $r_i(c_1, t_1, \bar{c_3})$. From our inductive hypothesis for this case, we derive a fact $r_{n+1}(t_1, c_2, \bar{c_3})$. Hence, using program $P_{r_{n+1}}$, we can derive a fact $r_{n+1}(c_1, c_2, \bar{c_3})$.

□

Theorem 3.1 does not consider the derivation of answer graphs using the distinguished edge. Only paths in the database graph G that satisfy an answer substitution θ of the query regular expression are considered. We can extend the result to a query Q_D with a distinguished edge D to derive an answer graph by simply applying substitution θ to the components of the D, as well as adding a "top-level rule" t_D to program P_r to generate a new program P_D. The head of t_D contains a predicate $d(X, Y, \bar{Z})$ generated from the distinguished edge $d(\bar{Z})$ for the query, from X to Y. The body of t_D contains a predicate $r_{n+1}(X, Y, \bar{F})$, where $\bar{F}$ is a tuple of all variables in expression r_{n+1}.

The set of answer facts produced corresponds to the edges generated by the graph semantics, since these edges also have an edge label similar to the distinguished edge label, and bindings

[4] If $m = 0$, p_1 is from c_1 to c_2 and p_2 is from c_2 to itself.

derived from the substitutions for the query. All unbound variables in the facts would also be unbound in the edges, since there would be no binding for these variables in the answer substitution.

Corollary 3.1 Let Q_D be a monotone GraphLog query with a single edge from X to Y labelled with regular expression r, and a distinguished edge D with label $d(\bar{Z})$, where $\bar{F}$ is the tuple of variables in r. $P_D(F_G)$ corresponds to $Q_D(G)$.

Proof: Every fact $f_a = d(c_1, c_2, \bar{c_3})$ in $P_D(F_G)$ is derived from some fact $f_o = r_{n+1}(c_1, c_2, \bar{c_4})$, using the top-level rule, where $\bar{c_3}$ are the bindings for $\bar{Z}$ and $\bar{c_4}$ for $\bar{F}$. From Theorem 3.1, we know that there is some path p_{n+1} in G satisfying $\theta(r_{n+1})$, where θ is the answer substitution such that $\theta(X) = c_1$, $\theta(Y) = c_2$ and $\theta(\bar{F}) = \bar{c_4}$. Since we apply substitution θ to distinguished edge D, from X to Y, we can derive an edge $e_a = (c_1, c_2, d(\bar{c_3}))$. For variables common to $\bar{F}$ and $\bar{Z}$, $\bar{c_3}$ and $\bar{c_4}$ agree, and $\bar{c_3}$ leaves other variables in $\bar{Z}$ unbound.

Conversely, edge $e_a = (c_1, c_2, d(\bar{c_3}))$ in $Q_D(G)$ is derived by applying a substitution θ to the distinguished edge, where θ is an answer substitution for $\bar{F}$, such that there is a path p_{n+1} in G from c_1 to c_2 satisfying $\theta(r_{n+1})$. Again, by Theorem 3.1, there is a fact $f_o = r_{n+1}(c_1, c_2, \theta(\bar{F}))$. We can derive fact $f_a = d(c_1, c_2, \theta(\bar{Z}))$ in $P_D(F_G)$ using our top-level rule. Since the same distinguished edge variables are used to derive e_a and f_a, fact f_a corresponds to edge e_a.

□

It is straight-forward to extend this result to GraphLog queries with multiple edges, by a further inductive proof based on the previous proof. This would involve decomposing the multiple-edge query into a series of single-edge queries, by adding a distinguished edge to every edge in the query.

3.5 Summary

We have introduced an alternative semantics for GraphLog, the graph semantics. These semantics are based on an interpretation of the meaning of GraphLog queries that is independent from the choice of translation.

We have also shown how an application of these semantics to certain queries results in a different set of results to those produced by the RE-translation. A modification to the RE-translation has been suggested to ensure that the expected set of results is produced. We

have proved that the semantics are equivalent for the case of single edge GraphLog queries. A description of the approach required to extend this result to multiple-edge queries was also given.

Possibilities exist for the optimization of various queries so that the separation of node and edge domains by the explicit definition of predicates such as "node" would not be required. This would mean that the domain integrity of the variables would be preserved, and the overhead of defining and incorporating these predicates would be eliminated.

Chapter 4

Query Translation and Optimization

This chapter describes the evolution of the factoring translation presented in [VW95] into a more general translation that may be used on visual queries with multiple edges. In addition, an optimizing technique referred to here as *variable constraining* is built into this general translation scheme.

We also show how the amount of intermediate computation may sometimes be reduced, while maintaining the query semantics. This results in further optimization of the query translated to Datalog. Such optimizations, which are not performed by the GraphLog/Hy$^+$ system, result in significant performance improvements in the evaluation of multiple-edge queries.

4.1 Overview

As mentioned previously, the usefulness of the NFA-translation is limited by the restrictive class of GraphLog queries that can be translated.

Example 4.1 Consider the query shown in Figure 4.1. This query describes a request by a fussy traveller who requires details of connected flights from all cities X1 to some city(ct) on some airline U. In addition, the traveller wishes to visit all towns X3 that are indirectly or directly reachable by trains, which are run by the same company U, from city(ct). The

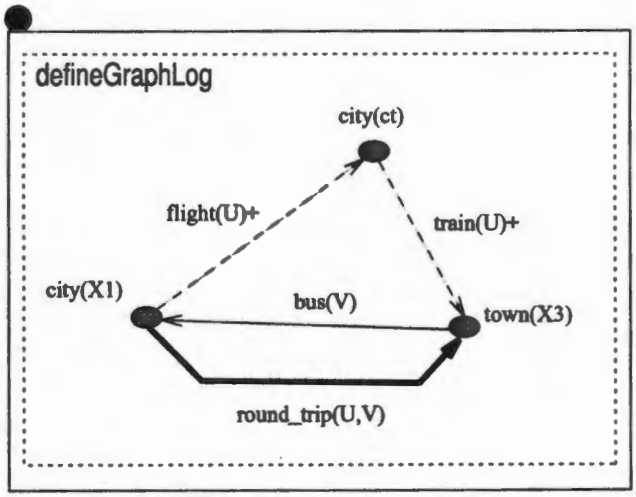

Figure 4.1: Round trip travel with stop in **city(ct)**

towns **X3** must also have a single bus trip, run by some company **V**, which returns to the original city **X1**.

■

The query in the example above demonstrates a practical application to the travel domain, which may be used in planning an itinerary where certain constraints need to be enforced. It also shows the power of the visual query formalism, which requires a comparatively lengthy text description to express all the implicit requirements and constraints contained in a fairly simple GraphLog query.

Importantly, there is no way to translate this query with the NFA-translation, as originally defined, since the restricted query language only allows for single-edge queries. Queries that have any intermediate node constraints that contribute towards the answer (like **city(ct)** or **town(X3)** in the example above) cannot be expressed using a single edge, because intermediate node values are an implicit part of the evaluation in the NFA-translation. Even if we assume that the nodes are restricted to variables, this query still cannot be translated by the NFA-translation. This is because the distinguished edge is not attached to the ends of the query. There is no way of obtaining node information which is an implicit part of a single-edge query.

The new algorithm, in the following section, generalizes the NFA-translation and eliminates both limitations mentioned in the previous paragraph. This algorithm will be referred to as

the M-NFA translation. As mentioned before, the NFA-translation has the useful property that if constants are present in the nodes of a visual query, the translation to Datalog constructs a factored program. The M-NFA translation preserves the factoring property for subqueries that contain node constants. This results in a significant improvement in performance over the RE-translation for certain multiple-edge queries which cannot be translated by the NFA-translation. This will be demonstrated in Chapter 5.

Our approach will be to translate a multiple-edge query as a series of subqueries, by constructing an ordering of the edges. We then construct a rule for the distinguished edge that includes predicates for all the subqueries in the rule body.

There are also multiple-edge queries where factoring is not possible, or a factoring translation is only possible for a part of the query. Where such a situation exists, we describe an optimizing technique, referred to as *variable constraining* for propagating the bindings of node variables between edges in a query. This allows sets of values to be passed forward in the query to potentially restrict the amount of subsequent evaluation. In addition, edge variables may also be passed to constrain the evaluation even further. This optimization is incorporated into the M-NFA translation.

The following example serves to provide a preview of the translation technique. It is intended to describe the whole process which may then be related to the following translation stages.

Example 4.2 Consider the query in Figure 4.1 again. We generate an ordering of the edges by performing a depth-first search starting at the node labelled with a constant:

e_1: (city(ct), town(X3), train(U)+).

e_2: (town(X3), city(X1), bus(V)).

e_3: (city(X1), city(ct), flight(U)+).

For each edge in the ordering, we find variables which are also present in an earlier edge in the ordering. We identify these variables as "constrained" and also record the most recent edge where each variable was present in parentheses.

e_1: Constrained: none

e_2: Constrained: X3 (e_1)

e_3: Constrained: X1 (e_2), U (e_1)

The translation produced for this query follows.

We first consider the first tuple in the ordering. This tuple has a constant city(ct) in the source node, and the resulting subquery translation is factored:

```
t_1_0(city(ct)).
t_1_1_U(Y,U) :- t_1_0(X), train(X,Y,U).
t_1_1_U(Y,U) :- t_1_1_U(X,U), train(X,Y,U).
```

We now create a rule for this subquery.

```
tie_1(town(X3),U) :- t_1_1_U(town(X3),U).
```

In this case, the terms in the head of this rule are the same as the terms in the body of the rule. However, this is not true for the general case.

Since variables on this edge are required for constraining future edges in the ordering, we create the following constraining rules.

```
proj_1_U(U) :- tie_1(town(X3),U).
proj_1_X3(X3) :- tie_1(town(X3),U).
```

We translate the second and third edges in the ordering in a similar way.

```
t_2_0(X3,X3) :- proj_1_X3(X3).
t_2_1_V(X,Y,V) :- t_2_0(X,T), bus(T,Y,V).
tie_2(town(X3),city(X1),V) :- t_2_1_V(town(X3),city(X1),V).
proj_2_X1(X1) :- tie_2(town(X3),city(X1),V).
```

```
t_3_0(city(ct)).
t_3_1_U(Y,U) :- t_3_0(X), flight(Y,X,U), proj_1_U(U).
t_3_1_U(Y,U) :- t_3_1_U(X), flight(Y,X,U).
tie_3(city(X1),U) :- t_3_q_U(city(X1),U), proj_2_X1(X1).
```

There are some interesting points concerning the four rules generated for the third edge in the ordering. The first point revolves around the use of variables in the second and third rules, whereas the visual query contains structured terms in the nodes. This is because the use of structured terms in the rules would alter the meaning of the query, and would restrict any (implicit) intermediate nodes in a closure computation to have a similar structure to the nodes in the query.

The second point involves the subgoal `proj_2_X1(X1)`, where the variable `X1` is a node variable. Since the other adjacent node contains a constant, we opt to generate a factored subquery, and only use `X1` to constrain the final stage of the computation in the fourth rule.

However, since the variable `U` is not a node variable, it can be used to restrict the values of all other occurrences of `U`. Since edge variables which are structured variable terms are included in the body of rules, all edge variables can be restricted. Therefore, we add the subgoal `proj_1_U(U)` to the body of all rules, corresponding to the third edge, where variable `U` appears.

We then generate a "top-level" rule to join the results of the various subqueries.

```
round_trip(city(X1),town(X3),U,V) :-
        tie_1(town(X3),U),
        tie_2(town(X3),city(X1),V),
        tie_3(city(X1),U).
```

■

We assume initially that we are given a query, comprising one define query and one filter query with a single distinguished edge.

The define query may have multiple non-distinguished edges, in addition to a single distinguished edge. Each non-distinguished edge may be labelled with an arbitrary regular expression (which may contain named variables and ground terms). The filter query simply refers to the distinguished edge of the define query.

Each node in the query should either have a single, unique variable as a label or a constant (not necessarily unique) as a label. Although unlabelled nodes are permitted for queries in Hy$^+$, in this chapter, we consider only query graphs and db-graphs in which all nodes are labelled. For the purposes of the following description, adjacent nodes in the define query may not have ground terms in both of them. This is connected to the problem of overbound queries studied in [MP91]. However, such queries are permitted in the implementation, where one ground term is replaced with a distinct variable during translation, which is existentially quantified in a subgoal.

It will be assumed that all algorithms operate on a GraphLog query format subject to the restrictions above.

The major steps in visual query translation are:

1. Perform a traversal of the query graph to generate an evaluation order for the query, creating an edge table that reflects this ordering.
2. Analysis of the edge table to determine whether variable constraining between edges is possible.
3. Translation of each edge in the edge table into a logic sub-query, in the order determined. Where possible, constant factoring is performed. Additionally, the variable constraining between edges, mentioned above, is used where applicable.
4. Generate rules for each distinguished edge in the query to provide answers to the query. This will involve the grouping of variables instantiated in each edge contained in the edge table. These rules will be known as the *distinguished rules.*

These steps are described in each of the following four sections.

In addition to the main optimization techniques that are built into the translation described, there are also a number of other optimizations that are applicable to special cases. These further optimizations are considered, and examples provided, in Section 4.6.

4.2 Query Graph Traversal

A depth-first search of the query graph, with edge directions ignored, is performed. This is used to generate a total ordering L of edges, so that information may be passed to edges that will only be evaluated later in the order.

Where there is a choice in edge selection, those edges with sink nodes which contain constant values are added to L before those nodes that contain variables. In addition, edges with sink nodes of higher degree are chosen first. We refer to these as the *constant heuristic* and the *degree heuristic*, respectively. The variable passing optimization technique in the following stage attempts to constrain the size of intermediate relations in query evaluation. These heuristics are appealing in the absence of other information, because they generally allow the largest reduction in the size of the intermediate relations.

Consider an edge e, from node X_b to some node X_d with edge regular expression exp, in the query graph. If e is traversed on iteration i of the depth-first search, then $L[i] = e(X_b, X_d, exp)$. This will give an unambiguous evaluation order for which a translation can be performed.

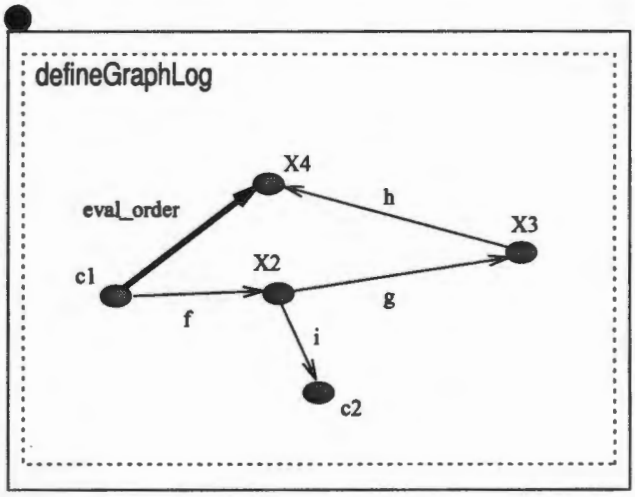

Figure 4.2: Query **eval_order**

Example 4.3 Consider the query **eval_order** in Figure 4.2. We generate an ordering of the edges in this query as follows:

e_1: (c1, X2, f)

e_2: (X2, c2, i)

e_3: (X2, X3, g)

e_4: (X3, X4, h)

Initially, we have to choose a node at which to start our traversal. We have a choice of constants **c1** or **c2**, using the constant heuristic. After traversing the first edge labelled **f**, and creating tuple e_1, we are presented with a choice of edges. We again use the constant heuristic to choose the edge labelled **i** and add tuple e_2 to the ordering. This demonstrates the precedence of the constant heuristic over the degree heuristic.

∎

4.3 Generation of Constraining Information

The term *constraining* refers to the restriction of the set of values to which a variable may be bound. In a multiple-edge query, the edges in the query have node variables in common,

and may have edge variables in common. Imagine that we decompose a query Q with n edges into n subqueries by duplicating shared nodes. Suppose that variable U labelled the distinguished edge or an adjacent node. The semantics of Q require that the set of U values present in the answer to Q must be the intersection of all the sets of U values computed for the edges also containing U in their labels. Since we evaluate the edges in a sequential fashion, we know that values computed for the common variables on a particular edge must also be present on all the other edges in order to be present in the final answer. We can therefore use these values to restrict the amount of subsequent computation for the other edges.

The technique therefore determines variable bindings between edges in the evaluation order generated in the previous step. The *passed variables* are those variables on the current edge whose bound values are used to restrict the domain of a variable on a future edge in the ordering. The *constrained variables* are those variables on the current edge which are restricted by a set of values bound on an edge earlier in the evaluation ordering. *Variable constraining* refers to the creation and use of passed variables to constrain (syntactically identical) variables later in the ordering.

This suggests a complementary relationship. One edge's passed variables are one or more future edge's constrained variables. *Constraining rules* are rules which isolate passed variables for a particular edge, so they may be used to constrain other variables.

The algorithm in Figure 4.3 is used to compute the set of variables that may be used to constrain an edge i, (the constrained variables), as well as the set of variables that an edge j passes to edges later in the ordering L, (the passed variables). These sets are denoted CV_i and PV_j, respectively, in the algorithm. A variable constrained on a particular edge e is a variable passed by another edge which precedes e in L. This information is computed for every edge in the graph.

Since each tuple e_i in the list L constructed above corresponds to an edge and its pair of adjacent nodes in the query graph, the terms "an edge e_i" and "the tuple e_i" will be used interchangeably.

Intuitively, the algorithm proceeds from the final edge in the ordering. Each variable present on the edge or its adjacent nodes is considered. In the inner loop, the algorithm runs backwards through the previous edges to determine whether the variable being considered is constrained by a variable in one of the previous edges.

If such a variable is found, the necessary information is saved in sets CV_i and PV_j for both edges. Note that the algorithm only finds the most recent instance of a variable in the

```
for every edge e_i in L {
    create an empty set of constrained variables, CV_i
    create an empty set of passed variables, PV_i
}
let n be the number of tuples in L.
let i = n.
while (i > 0) {
        for edge e_i in L
        for each variable V in e_i {
            let j = i - 1;
            let var_found = false;
            while (j >= 0) and not (var_found){
                    if variable V is present in edge e_j {
                      add V to PV_j;
                      let var_found = true;
                    }
                    else
                      let j = j - 1;
            }
            if (var_found){
                    add V to CV_i.
            }
        }
        let i = i - 1;
}
```

Figure 4.3: Algorithm to find constraining information

ordering. This ensures that the smallest set of values is used to constrain an edge. At this point, the inner loop terminates and algorithm repeats, until the first edge in the ordering is encountered. This method ensures that all possible constraining information in the ordering is computed.

No constraining information is computed for the first edge in the ordering since, clearly, it cannot be constrained by any other edge. Also, no variables are passed by the final edge since this information is not relevant to any other edge.

As with most optimization techniques, specific situations may arise, where performing a depth-first search and using the heuristics do not result in the best possible ordering for constraining. Generally though, the method described will result in a better ordering than a non-sequential ordering. One of the ways of measuring this is by calculating the number of edges, as well as the number of variables, that are constrained.

Example 4.4 Consider the query shown in Figure 4.2 on page 72. We use the algorithm in Figure 4.3 to find the following constraining information for the ordering generated in Example 4.3:

e_1: Constrained: none

e_2: Constrained: X2 (e_1)

e_3: Constrained: X2 (e_2)

e_4: Constrained: X3 (e_3)

We see that the total number of constrained variables here is 3, and the number of constrained edges is also 3.

Suppose we were to choose a ordering which is not connected instead:

e_1: (c1, X2, f)

e_2: (X3, X4, h)

e_3: (X2, c2, i)

e_4: (X2, X3, g)

By applying the algorithm in Figure 4.3, we obtain the following:

e_1: Constrained: none

e_2: Constrained: none

e_3: Constrained: X2 (e_1)

e_4: Constrained: X2 (e_3), X3 (e_2)

Here, we see that the total number of constrained variables is also 3, but in this case, the number of constrained edges is 2.

Since we are attempting to restrict as many edges as soon in the ordering as possible, the first connected ordering generated by the depth-first search is preferable.

∎

The previous example illustrates how a connected search, such as depth-first search, ensures that the maximum number of edges are constrained, since every edge in the query (except the first) is constrained. This may be compared to an unconnected ordering, where although the total number of variables that are constrained may be the same, fewer edges are constrained. This is achieved simply by first choosing edges which do not have a common node with any other edge already in the ordering. If this is the case, then the nodes of these edge cannot be constrained.

It is important to note that the method of constraining the source or sink variables of subsequent edges in a query is only effective when the size of the relation containing the constraining variable is less than the size of the relation containing the constrained variable. Therefore, the effectiveness of the variable constraining optimization is data-dependent, with the assumption that the overhead of constraining the variables is compensated for by the reduction of subsequent computation. A similar idea for the optimization of bound query variables in Datalog programs is described in [KRS90], This is discussed in Example 4.5.

Also, if the edge being considered contains some type of closure operator, the payoff from an initial reduction of the base relation will generally significantly reduce the amount of computation subsequently required for the closure. We are assuming that the databases under consideration are very large, and the set of values bound to a passed variable is a substantially smaller subset of the total set of values in its domain.

Example 4.5 Consider the following rules for deriving the ancestor relation:

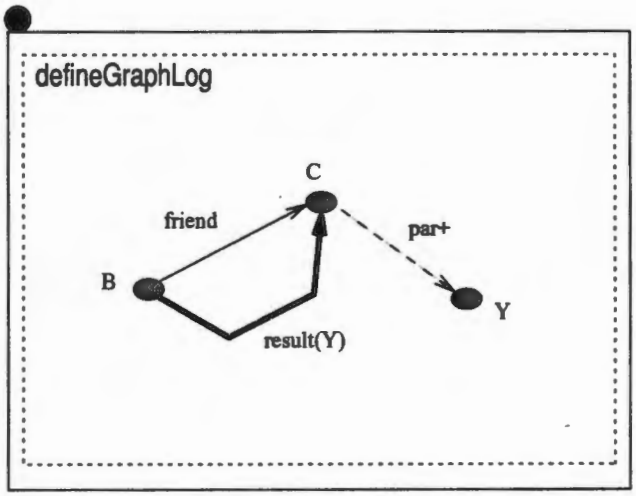

Figure 4.4: Ancestors of friends

```
anc(X,Y) :- par(X,Y).
anc(X,Y) :- par(X,Z), anc(Z,Y).
```

Suppose there is some base relation **friend** with arity 2, and that the query is:

```
?friend(B,C), anc(C,Y).
```

This is equivalent to the context-rewritten program P in Figure 2.9 on page 28 produced by the transformation described in [KRS90].

Suppose we translated the visual query in Figure 4.4 using the M-NFA translation. The Datalog query Q produced would be:

```
t_0_0_(X,X).
t_0_1_(X,Y) :- t_0_0_(X,T), friend(T,Y).
tie_0(B,C) :- t_0_1_(B,C).
proj0_C(C) :- tie_0(B,C).

t_1_0_(C,C) :- proj0_C(C).
t_1_1_(X,Y) :- t_1_0_(X,T), par(T,Y).
t_1_1_(X,Y) :- t_1_1_(X,T), par(T,Y).
```

```
tie_1(C,Y) :- t_1_1_(C,Y).

result(B,C,Y) :- tie_0(B,C), tie_1(C,Y).
```

Although the Datalog code generated by the M-NFA translation is syntactically different to the context-rewritten program, a comparison suggests that the method used to optimize each program is similar. If we examine the fifth rule in Q, we notice that the set of C values is used in the first step of the computation of ancestors in the following two rules. This is almost identical to the addition of a **friend** subgoal to the first rule of P, which also uses the "context" information represented by variable C as the first step in the recursive ancestor computation.

Since the context-rewritten example P assumes that **friend** is an EDB predicate, the first four rules in P are eliminated by adding a **friend** subgoal to the final rule. The M-NFA translation generates the additional rules as part of a more general method required to translate all GraphLog queries, although such special-case rule reduction may be used where possible.

■

4.4 Edge Translation

After generating constraining information, the next step is to translate each edge $e_i = (X_b, X_d, R)$ in L into Datalog. The outline of the algorithm is as follows:

1. Construct an NFA M which accepts the language $L(R)$.
2. If X_b is not constrained and X_d is constrained, or X_d contains a ground term, we reverse the automaton M and perform the inversion of each term labelling a transition in M. Although reversing the automaton may result in multiple initial states, the translation is not affected. This now means that X_b is now X_d and vice-versa for following steps in the translation.[1]
3. Perform the NFA-TRANSFORM algorithm to generate program P (see Section 4.4.1 below).

[1] Alternatively, an equivalent translation which effectively does the automaton reversal in an implicit way may be performed. For conciseness, this translation is not described here, although it is used in the implementation.

4. Perform a bottom-up propagation of edge variables in P, and call the resulting program Q (see Section 4.4.2 below)

5. Add rules to generate bindings for all variables to be passed to the distinguished edge, called *edge module rules*, to Q (see Section 4.4.3).

6. Add rules to isolate all the passed variables required by other edges, called *project rules*, to Q (see Section 4.4.4).

4.4.1 The NFA-TRANSFORM Algorithm

Input : edge $e_i(X_b, X_d, R)$, NFA $M = (S, \Sigma, \delta, s_0, F)$, constrained variables CV_i.

1. (a) If source node label X_b is a variable term, generate a rule with head $t\_i\_s(X_b, X_b)$, where s is the initial state of the automaton. For each variable C in X_b that is also contained in CV_i, add a subgoal $proj\_m\_C(C)$ where m is the edge number where C was passed from.

   (b) Otherwise, generate a fact $t\_i\_s(X_b)$, where X_b is a ground term.

   To avoid naming conflicts, we assume that predicate names beginning with t are distinct from any of the predicate names contained in any query edge label.

2. (a) If source node label X_b is a variable term, then for each transition τ in M from p to q labelled with $e(\bar{Z})$, where $\bar{Z}$ may be a sequence of terms, generate a Datalog rule as follows:

   $$t\_i\_q(X, Y) \text{ :- } t\_i\_p(X, T), e(T, Y, \bar{Z}).$$

   If, instead, τ is labelled with $-e(\bar{Z})$, generate:

   $$t\_i\_q(X, Y) \text{ :- } e(Y, T, \bar{Z}), t\_i\_p(X, T).$$

   For each variable C contained in $\bar{Z}$ that is an element of constraining set CV_i, add the subgoal $proj\_m\_C(C)$, where m is the edge number where C was passed from.

   (b) Otherwise (X_b is ground), generate a modified version of the rules created in step 2(a), by translating as described, and then removing each occurrence of variable X from each IDB predicate in every rule.

4.4.2 Edge Variable Propagation

From P generate a new program Q by performing a bottom-up propagation of the edge variables.

1. Add all rules generated in step (1) of the NFA-TRANSFORM algorithm to Q.
2. If source node label X_b is a variable term, the propagation is as follows. Otherwise, we propagate as described, but remove each occurrence of variable X in all IDB predicates.

   (a) For each rule in P containing a t_i_s subgoal, of the form:

   $$t\_i\_p(X,Y) \text{ :- } t\_i\_s(X,T), e(T,Y,\bar{Z}), \mathcal{G}_1,\ldots,\mathcal{G}_n.$$

   where $\mathcal{G}_1,\ldots,\mathcal{G}_n$ refers tc all $proj_j\_V_k(V_k)$ subgoals in the body, add the rule

   $$t\_i\_p_{\bar{U}}(X,Y,\bar{U}) \text{ :- } t\_i\_s(X,T), e(T,Y,\bar{Z}), \mathcal{G}_1,\ldots,\mathcal{G}_n.$$

   to Q, where $\bar{U}$ comprises all named free variables (without bound terms and anonymous variables) in $\bar{Z}$.

   (b) Repeat the following process until no syntactically new rule is added to Q. If there is a rule with head $t\_i\_p_{\bar{U}}(Y,\bar{U})$ in Q and a rule of the form

   $$t\_i\_r(Y) \text{ :- } t\_i\_p(T), e(T,Y,\bar{Z}), \mathcal{G}_1,\ldots,\mathcal{G}_n.$$

   in P, then add the rule

   $$t\_i\_r_{\bar{V}}(Y,\bar{V}) \text{ :- } t\_i\_p_{\bar{U}}(T,\bar{U}), e(T,Y,\bar{Z}), \mathcal{G}_1,\ldots,\mathcal{G}_n.$$

   to Q, where $\bar{V}$ is a sequence of named free variables which includes all of those in $\bar{Z}$ and $\bar{U}$ such that the ordering of variables is consistent throughout the rules.

Example 4.6 For the first edge in the in the ordering generated in Example 4.2 on page 68, the NFA-TRANSFORM algorithm produces the following program P:

```
t_1_0(city(ct)).
t_1_1(Y) :- t_1_0(X), train(X,Y,U).
t_1_1(Y) :- t_1_1(X), train(X,Y,U).
```

Edge variable propagation results in a program Q:

```
t_1_0(city(ct)).
t_1_1_U(Y,U) :- t_1_0(X), train(X,Y,U).
t_1_1_U(Y,U) :- t_1_1_U(X,U), train(X,Y,U).
```

■

4.4.3 Generating Edge Module Rules

Edge module rules are created to collect all variables in the heads of rules which are generated from accept states, and contribute to the answer for this edge sub-query. This includes node and edge variables (which are propagated into the rule heads) for the edge. This is comparable to step 4 of the standard NFA-translation and is required so that the sub-query for this edge is represented by a single predicate containing all the variables.

1. (a) If the source node contains a variable term, we add the following rule:

   $$tie\_i(X_b, X_d, \bar{V}) \text{ :- } t\_i\_f(X_b, X_d, \bar{Z}).$$

   to Q, for every rule r in Q that is derived from a transition τ leading to a final state f, where $\bar{V}$ contains all the variables that appear in the edge regular expression R, and $\bar{Z}$ contains all the edge variables contained in the head of r.

   (b) Otherwise, we add the rule and then remove each occurrence of variable X_b.

2. For each variable C in X_d that is also contained in CV_i, add a subgoal $proj\_m\_C(C)$, where m is the edge number where C was passed from.

Note that $\bar{V}$ and $\bar{Z}$ are not necessarily the same, as there are instances where the predicate in the head of the rule contains variables which are not present in the body predicate. This is demonstrated in the following example. This problem is related to the safety of queries and is discussed in Section 4.7.

Example 4.7 Consider the query in Figure 4.5. We translate this query using the edge translation rules as follows:

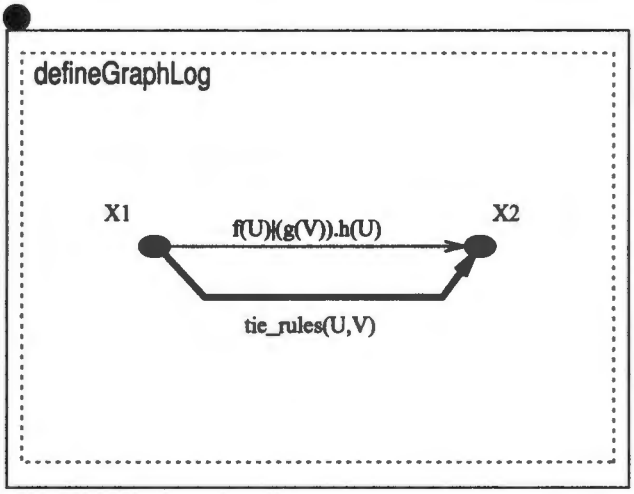

Figure 4.5: Query tie_rules

```
t_1_0(X1,X1).
t_1_1(X,Y) :- t_1_0(X,T), f(T,Y,U).
t_1_1(X,Y) :- t_1_0(X,T), g(T,Y,V).
t_1_2(X,Y) :- t_1_1(X,T), h(T,Y,U).
```

By performing an edge variable propagation, we obtain the following program.

```
t_1_0(X1,X1).
t_1_1_U(X,Y,U) :- t_1_0(X,T), f(T,Y,U).
t_1_1_V(X,Y,V) :- t_1_0(X,T), g(T,Y,V).
t_1_2_U(X,Y,U) :- t_1_1_U(X,T,U), h(T,Y,U).
t_1_2_U_V(X,Y,U,V) :- t_1_1_V(X,T,V), h(T,Y,U).
```

We notice that the rules which would compute the required answers (with head predicates t_1_2_U and t_1_2_U_V) have differing arities, namely 3 and 4. This is due to a problem involving the safety of the query, which is further discussed in Section 4.7. However, these predicates would have to be part of a rule joining all the subgoals associated with every edge in the ordering.

One solution is to create multiple distinguished rules for each distinguished edge, with all the combinations of predicates of differing arities. However, the extra overhead involved

would make this solution unworkable. Instead, we create multiple rules at the edge module level containing all the query variables as follows:

```
tie_1(X1,X2,U,V) :- t_1_2_U(X1,X2,U).
tie_1(X1,X2,U,V) :- t_1_2_U_V(X1,X2,U,V).
```

This means that only a single subgoal is required for each edge in the distinguished rule.

∎

Our inclusion of **proj** subgoals in the body of edge module rules allows us to constrain variables in the sink node in addition to those in the source node, when possible. Although not reducing the amount of computation for this edge, it allows a possible further reduction in the number of values used to constrain a future edge. The overhead of extra joins of subgoals of arity 1 is felt to justify the pay-off possibilities. In the event that this is not the case, selective or total elimination of variable constraining is possible. This is discussed further in Chapter 5.

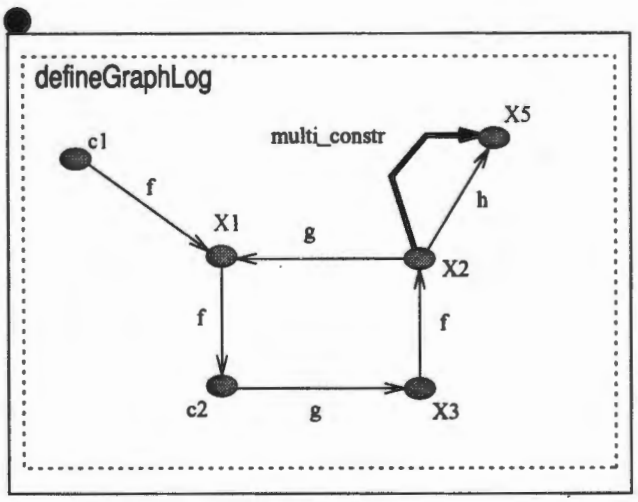

Figure 4.6: Query multi_constr

Example 4.8 Suppose we translate the query in Figure 4.6.

We first generate an ordering as follows:

e_1: (c1, X1, f)

e_2: (X1, c2, f)

e_3: (c2, X3, g)

e_4: (X3, X2, f)

e_5: (X2, X1, g)

e_6: (X2, X5, h)

We now find the constraining information:

e_1: Constrained: none

e_2: Constrained: X1 (e_1)

e_3: Constrained: none

e_4: Constrained: X3 (e_3)

e_5: Constrained: X1 (e_2), X2 (e_4)

e_6: Constrained: X2 (e_5)

Looking at the constraining information, we observe that edge e_5 has both its source and sink node variables constrained. We also notice that e_5 also passes information to edge e_6. If we were to just use just one of these variables to constrain the edge, then the amount of information passed could increase substantially, and so dramatically reduce the amount of constraining on edge e_6.

It is with this situation in mind that we add an extra `proj` subgoal to the edge module rules to allow both node variables to be constrained.

■

Notice that the head of every edge module rule for a particular edge is identical. Therefore, when we refer to the head of a set of edge module rules for an edge, we refer to a single predicate.

4.4.4 Generating Project Rules

Project rules are created to project out variables from the edge module rules. These variables are used to constrain an edge which is later in the ordering. The reason that variables are separated instead of using the head of the edge module (which contains all the variables) is to avoid computing potentially expensive joins with a predicate of much greater arity, sometimes repeatedly in a closure. By separating the variables, the joins only involve a predicate of arity 1, although this requires greater space cost, as well as the cost of the projection.

For each variable C contained in PV_i, add a rule

$$proj\_i\_C(C) \text{ :- } tie\_i(\bar{W}).$$

to Q, where $tie\_i(\bar{W})$ corresponds to the head of the edge module rules for e_i.

4.5 Adding Rules for the Distinguished Edge

The distinguished edge d should be translated as a distinguished rule D_r, which has a head predicate which includes all distinguished query variables, and has the same name as the distinguished predicate. For each edge i in the ordering, add a single subgoal $tie\_i(\bar{W})$, corresponding to the head of the edge module rules for i, to the body of D_r.

This rule creation may also be optimized by eliminating subgoals which do not contribute towards the query result. This is demonstrated in Example 4.11 on page 88. The following example is an extension of the standard "common ancestor" query, applied to the object-oriented domain.

Example 4.9 Consider the query graph in Figure 4.7. The edge ordering L that results from the traversal of the graph is:

e_1 (X1, function(destroyer), calls+)

e_2 (X1, X2, calls+)

e_3 (X3, X2, classmember)

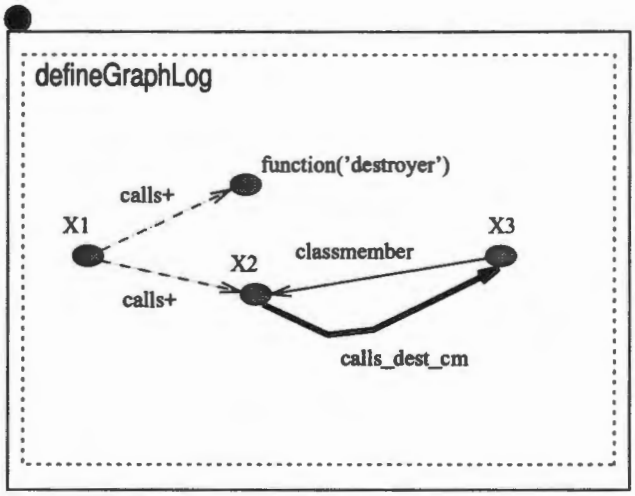

Figure 4.7: Query calls_dest_cm

This order is constructed using the heuristics suggested in Section 4.2. The logic program that corresponds to this query, using the M-NFA translation is:

```
t_1_0(function("destroyer")).
t_1_1(X) :- t_1_0(Y), calls(X,Y).
t_1_1(X) :- t_1_1(Y), calls(X,Y).
tie_1(X1) :- t_1_1(X1).
proj1_X1(X1) :- tie_1(X1).

t_2_0(X1,X1) :- proj1_X1(X1).
t_2_1(X,Y) :- t_2_0(X,T), calls(T,Y).
t_2_1(X,Y) :- t_2_1(X,T), calls(T,Y).
tie_2(X1,X2) :- t_2_1(X1,X2).
proj2_X2(X2) :- tie_2(X1,X2).

t_3_0(X2,X2) :- proj2_X2(X2).
t_3_1(X,Y) :- t_3_0(X,T), classmember(Y,T).
tie_3(X2,X3) :- t_3_1(X2,X3).

calls_dest_cm(X2,X3) :- tie_1(X1), tie_2(X1,X2), tie_3(X2,X3).
```

The first two rule blocks in the above logic program correspond to the common_anc program presented in [VW95]. Certain redundant relations are computed in this translation, to ensure the generality of the translation.

∎

4.6 Additional Optimization Possibilities

As mentioned earlier, there are various possibilities of optimizing queries in addition to the mainstream techniques that are part of the translation. These are discussed in the following subsections. Although they are not currently implemented, their effect on performance is considered in Section 5.5.4.

4.6.1 Existential Subqueries

The following example describes how the different edges of a query may be optimized by different evaluation methods.

Example 4.10 Consider the query in Figure 4.8. We want to avoid computing the entire set of X1 values for f(X1,c), in other words, a selection on the whole relation. If there is a closure on the first edge, there could be an even larger amount of computation. What we want is a single (existential) X1 value, that satisifies the first edge, and then to compute the entire set of X2 values with g(c,X2).

Therefore, the first edge could just be evaluated for a single value rather than a set of values. i.e. Find the first answer that satisfies the first (f) edge and then compute the second edge (g) bottom-up.

∎

If the edge being considered contains a node with a constant and any variables on the edge are not part of the distinguished edge or present in the set of passed variables, then this query qualifies as an existential query and may be handled as such.

4.6.2 Redundant Subgoal and Variable Elimination

Subgoals in the distinguished rule may sometimes be eliminated. This is due to variable constraining, where in certain circumstances, a subset of the subqueries generated for each edge may contain all the results required in the distinguished edge.

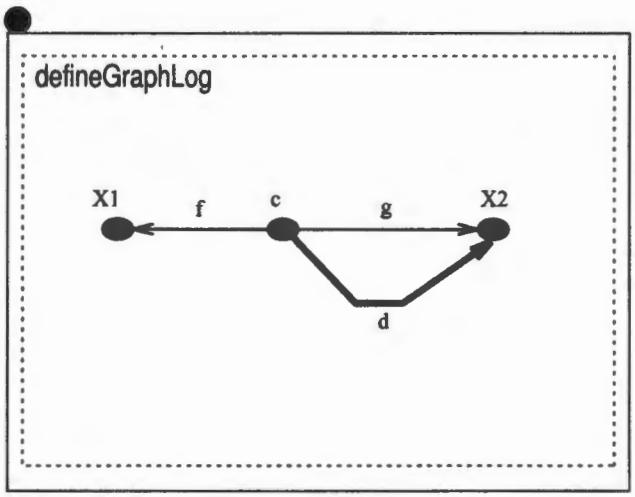

Figure 4.8: Existential subquery

The difference between this optimization and the existential query optimization, is that this optimization results in a query that has fewer subgoals and/or variables, whereas existential optimization recommends a different evaluation method (top-down evaluation) to be used.

The following example shows how such elimination may be performed.

Example 4.11 Consider the query in Figure 4.9.

This is translated as:

```
t_0_0(X1,X1).
t_0_1(X,Y) :- t_0_0(X,Z), f(Z,Y).
tie_0(X1,X2) :- t_0_0(X1,X2).
proj_0_X2(X2) :- tie_0(X1,X2).

t_1_0(X2,X2) :- proj_0_X2(X2).
t_1_1(X,Y) :- t_1_0(X,Z), g(Z,Y).
tie_1(X2,X3) :- t_1_1(X2,X3).

d(X2,X3) :- tie_0(X1,X2), tie_1(X2,X3).
```

However, all `X2` values that are used in the distinguished rule have been passed to the rules that generate values for `tie_1(X2,X3)`. This means that the range of X2 values in `tie_1`

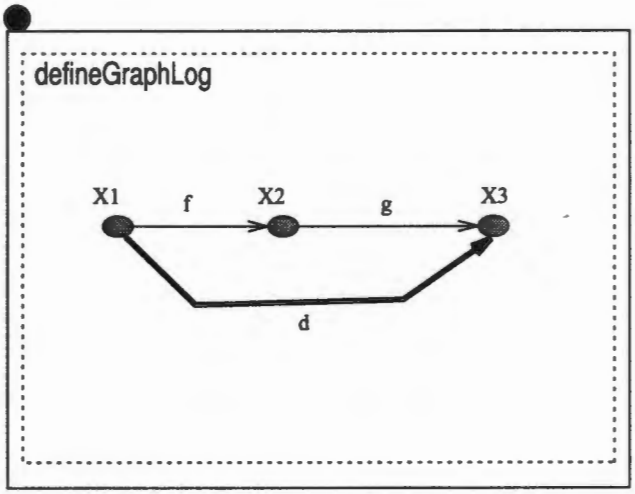

Figure 4.9: Redundant Subgoal Elimination

is no larger than the set of X2 values in tie_0. Since the X1 variable is not present in the head of the distinguished rule, the first subgoal is redundant and may be eliminated without affecting the result.

■

In a similar way to factoring, queries that only contain variable terms in nodes may result in some of those variables being eliminated in the rules produced by the translation, if the variables are not used anywhere else. This contrasts with the factoring performed by the NFA-translation, where the arity of literals is reduced only if a constant is present in a node in the query. This reduction in arity can result in a substantial improvement in the performance of various queries, which will be shown in Chapter 5. This is demonstrated in the example below.

Example 4.12 Consider the Datalog query in Example 4.11. We can further improve the efficiency of the translation by eliminating unused variable X1 from the first 4 rules of the resultant Datalog program as follows:

```
t_0_0(X1).
t_0_1(Y) :- t_0_0(Z), f(Z,Y).
tie_0(X2) :- t_0_0(X2).
```

```
proj_0_X2(X2) :- tie_0(X2).

t_1_0(X2,X2) :- proj_0_X2(X2).
t_1_1(X,Y) :- t_1_0(X,Z), g(Z,Y).
tie_1(X2,X3) :- t_1_1(X2,X3).

d(X2,X3) :- tie_1(X2,X3).
```

This is similar to the existential optimization described in Section 4.6.1, but here we need to compute a set of X2 values, as compared to a single existential value. Therefore, the set of X2 values may be computed using a standard bottom-up computation method, whereas a different evaluation technique is required for existential subquery optimization.

■

4.7 Safety

As discussed in Section 2.3.3, Datalog rules which contain variables in the rule head which are not present in any predicate in the rule body are unsafe. These variables are considered to be unbound, and range over an infinite set of values. In Chapter 3, we have seen that the RE-translation produces Datalog queries that are not safe and considered the semantic implications of this. In this section, we consider the safety of the M-NFA translation and compare this to the safety of the RE-translation. We find that there are occasions where both the RE-translation and the M-NFA translation generate Datalog rules which are not safe.

However, there are queries for which the RE-translation generates unsafe rules, whereas the M-NFA translation does not generate unsafe rules, although both programs are safe. This is due to the method by which the Datalog program is constructed, as compared to the semantic problem mentioned above. As a result, every path in any database graph that satisfes such a query instantiates all the variables. The answer does not contain unbound variables, although the program generated by the RE-translation contains unsafe rules.

Example 4.13 We re-examine the query in Figure 4.5 on page 82. The program in Example 4.7, generated by performing the M-NFA translation on this query, contained unsafe rules.

If we performed the RE-translation on this query, we would also obtain a program containing unsafe rules:

```
tie_rules(X1,X2,U,V) :- alter(X1,T1,U,V), h(T1,X2,U).
alter(X,Y,U,V) :- f(X,Y,U).
alter(X,Y,U,V) :- g(X,Y,V).
```

This is an example of a semantic safety problem, since it is possible to traverse a path in a database graph (say, an f edge followed by an h edge) which satisifes the query, but has not instantiated variable V. As a result, there are valid answers where V is unbound. ∎

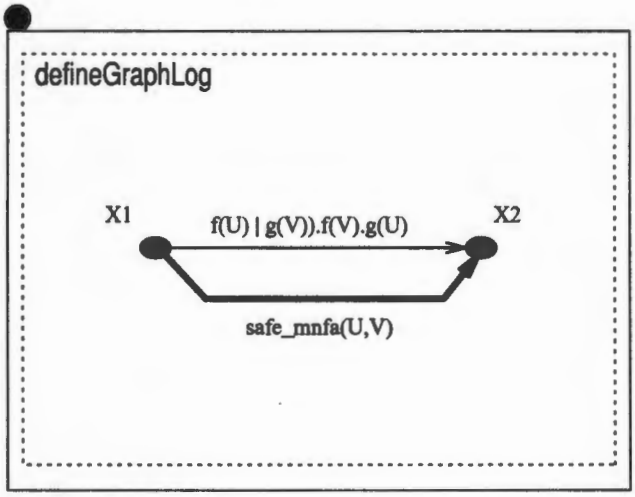

Figure 4.10: Query **safe_mnfa**

Example 4.14 The query in Figure 4.10 is translated to a Datalog program P_1 by the M-NFA translation as follows:

```
t_0_0(X,X).
t_0_1_U(X1,X2,U) :- t_0_0_(X1,YY), f(YY,X2,U).
t_0_2_V(X1,X2,V) :- t_0_0(X1,YY), g(YY,X2,V).
t_0_3_U_V(X1,X2,U,V) :- t_0_1_U(X1,YY,U), f(YY,X2,V).
t_0_4_U_V(X1,X2,U,V) :- t_0_3_U_V(X1,YY,U,V) , g(YY,X2,U).
t_0_3_V(X1,X2,V) :- t_0_2_V(X1,YY,V), f(YY,X2,V).
```

```
t_0_4_U_V(X1,X2,U,V) :- t_0_3_V(X1,YY,V), g(YY,X2,U).
tie_0(X1,X2,U,V) :- t_0_4_U_V(X1,X2,U,V).
safe_mnfa(X1,X2,U,V) :- tie_0(X1,X2,U,V).
```

The RE-translation produces the following program P_2:

```
safe_mnfa(X1,X2,U,V) :- alter(X1,T1,U,V), f(T1,T2,V), g(T2,X2,U).
alter(X,Y,U,V) :- f(X,Y,U).
alter(X,Y,U,V) :- g(X,Y,V).
```

Although both programs result in bindings for variables U and V, it is clear that all rules produced by the M-NFA translation are safe, which is not the case with the RE-translation. This is due to the method of variable propagation used by the M-NFA translation to distribute variables throughout the program. Although this has the effect of generating more rules, as is evident in this example, it also ensures that in certain cases, unsafe rules are not produced.

The results of both queries are the same, though, since P_2 is safe in the sense that both U and V are both bound in the answer. Nevertheless, if a back-end which was incapable of evaluating programs with unsafe rules was used, no answers would be produced for P_2.

■

We introduce a safety rule for the M-NFA translation which may be applied to the NFA constructed for each edge to determine whether the subquery, generated by the translation for the edge, is safe.

> **Safety Rule:** Every path from a start state to an accept state in the NFA must instantiate all the variables on the query edge.

Unsafe rules have posed a problem for logic database systems in the past, but this has largely been overcome. As mentioned earlier, CORAL handles unbound variables as part of its capability of dealing with non-ground facts. Therefore, there is no longer any danger that a translation of a visual query that produces unsafe rules will not be evaluated. This also means that the answers to unsafe Datalog queries contain unbound variables, indicated by CORAL with variables beginning with an underscore symbol. The user may be presented with these unbound variables as part of the answer, so a suitable explanation regarding

their meaning will have to be provided. The notion of a database graph also needs to be extended, since the composition of queries may result in queries being evaluated on a database graph which contains unbound variables. This extension has been described after the definition of a database graph, Definition 2.6 on page 12.

4.8 Translating Equivalent Query Forms

One of the aspects of query formulation that has not been discussed is that equivalent queries may be written using a variety of syntactically different queries, particularly when multiple edges are used. This is related to the expressive power of GraphLog. However, this may have consequences for the efficiency with which queries are evaluated. The ideal translation technique should be general enough to translate all GraphLog queries, but identify and perform the maximum amount of special case optimization, so that the most efficient Datalog query would be produced.

In this section, we examine queries that are semantically identical but differ in form (i.e. syntactically). We consider the overhead of evaluating multiple-edge queries with the M-NFA translation, which may be rewritten as a single-edge query and translated with the NFA-translation. Since the RE-translation does not attempt to perform any optimization as part of the translation, different query forms do not have as significant an impact.

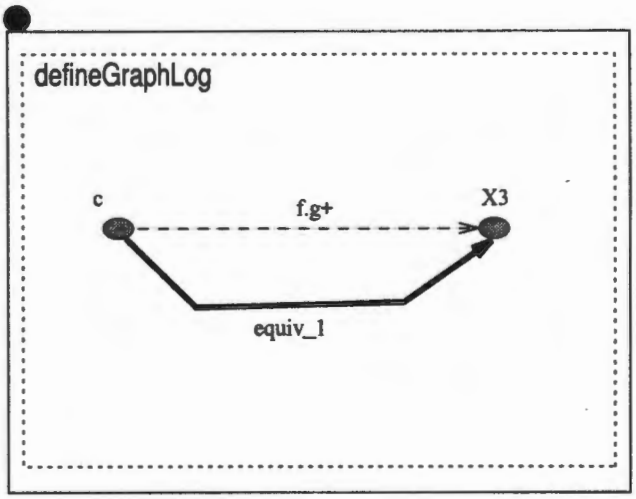

Figure 4.11: Query **equiv_1**

For query **equiv_1** in Figure 4.11, the NFA-translation would produce program P_1:

```
t_1_0(c).
t_1_1(Y) :- t_1_0(T), f(T,Y).

t_1_2(Y) :- t_1_1(T), g(T,Y).
t_1_2(Y) :- t_1_2(T), g(T,Y).

equiv_1(c,X3) :- t_1_2(X3).
```

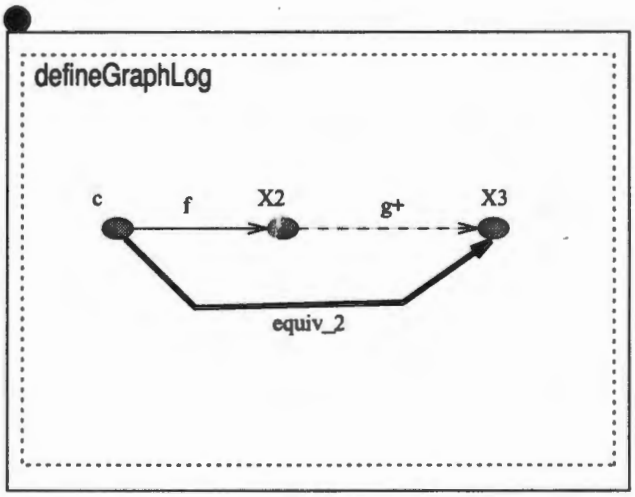

Figure 4.12: Query **equiv_2**

For the query **equiv_2** in Figure 4.12, the M-NFA translation would produce a program P_2:

```
t_1_0(c).
t_1_1(X2) :- t_1_0(T), f(T,X2).
tie_1(X2) :- t_1_1(X2).
proj1_X2(X2) :- t_1_1(X2).

t_2_0(X2,X2) :- proj1_X2(X2).
t_2_1(X,Y) :- t_2_0(X,T), g(T,Y).
t_2_1(X,Y) :- t_2_1(X,T), g(T,Y).
tie_2(X2,X3) :- t_2_1(X2,X3).

equiv_2(c,X3) :- tie_1(X2), tie_2(X2,X3).
```

Clearly, though semantically identical, the first visual query is syntactically more concise than the other. Program P_1 is also more concise than program P_2. We see that the arity of some of the predicates is greater in the second program, and there is a "top-level" rule required in the second program which is not present in the first program.

The M-NFA translation attempts to combine the benefits of the factoring NFA-translation with those of the RE-translation; specifically the restriction of the number of variables required to be propagated throughout the query. Variable propagation can result in an exponential number of rules in the worst case depending on the number of variables in the query, as well as resulting in predicates with very large arities. We translate the query by dividing it up into subqueries to restrict this exponential increase to the number of variables in the subquery. The cost of this, which both the M-NFA translation and the RE-translation incur for multiple-edge queries, is the top-level join which is also required.

One of the potential solutions to this problem of losing efficiency with generality is to do a set of distinguished edge-based optimizations. This would involve examining the position and variables of the distinguished edge and using this information to direct the translation. This would therefore eliminate the redundant information computed by the more general translation.

However, if user did use the most concise form possible (i.e. query **equiv_1** instead of query **equiv_2**), the M-NFA translation will generate a translation that is essentially as efficient as the standard NFA-translation. This is because the NFA-TRANSFORM algorithm is mostly identical to the NFA-translation, and no variable constraining for single edge queries. Hence, the NFA-translation is a special case of the M-NFA translation. Ideally, translation of the most concise form should be automatic, especially if user expertise is to be reduced.

4.9 Discussion

The NFA-translation produces a potentially exponential number of rules with the variable propagation technique, as well as an increase in the arity of predicates. This means that a possible alternative of translating the entire multiple-edge query as a single automaton, created by combining the automaton for each of the edges, is not feasible. This would result in a large increase in the number of rules that would be generated by the variable propagation technique.

With this problem in mind, the M-NFA translation attempts to combine the advantages of the RE-translation and the NFA-translation, in order to generate factored programs

where possible, but also to reduce the number of rules and arity of predicates by translating according to the structure of the multiple-edge query, and joining all subqueries in the distinguished rule.

Although introduced within the context of the M-NFA translation, the variable constraining technique is not specific to the NFA-translation method. In fact, the variable constraining technique may also be applied to multiple edge queries translated with the RE-translation. This would require a relatively simple change to the RE-translation and could result in substantial performance improvements over a non-optimized translation.

One of the consequences of using the variable constraining technique is that the evaluation of edges must proceed in a serial order. This effectively limits the benefits if the query was to be evaluated in parallel. However, if parallel evaluation is desired, the translation may be trivially modified to prevent any variable passing occurring, therefore eliminating inter-dependencies between edges in the query.

We have adopted the approach of building factoring and other optimizations into the translation, as compared to rewriting the translated program. As motivated in Chapter 1, this approach ensures that the vagaries of a particular query optimizer do not prohibit an efficient translation of visual queries. Queries may then be executed on a number of deductive database back-ends without an execution time increase as a result of less efficient back-end optimization.

Chapter 5

System Details and Performance

The application of theory in the form of a working system to solve the complex real-world problems that we are constantly presented with is a good indicator of its usefulness. This system should also be applicable to any domain that may be visualized as a directed graph, and there are no restrictions on the data set used.

The interactive value of a visual querying system is determined by the speed at which queries are evaluated. In addition, when better performance is a motivation for a new translation method, it is important to be able to quantify the differences, particularly with a set of real, rather than contrived, data. This is how the typical user of Hy^+ would expect the system to perform when evaluating queries.

In this light, a system for the translation of visual queries has been implemented in conjunction with the theoretical work that has been presented in the previous chapters. In Section 5.1, details of this system will be discussed.

The application of Hy^+ to various domains, including an object-oriented class library, networking and distributed computing, has been described [CH93] [Con94] [CHM93]. In this tradition, a logic database has been created by filtering a real-world Geographical Information System (GIS) database containing details of geographical information and features. Various aspects of this database will be presented, in addition to a few example queries in Section 5.2. We also describe queries based on an object-oriented class library database in Section 5.3 [VW95], as well as a memory overlaying application in Section 5.4 [CMVW95]. These queries have been used to evalute efficiency of the alternative translations in previous work, and are included here for comparison.

Finally, Section 5.5 describes performance results obtained by evaluating the alternative

translations of various queries with the CORAL system. These results are intended to give the reader an idea of the real-world speed-up that can be obtained by using the M-NFA translation on a reasonably large dataset, using typical queries.

5.1 The EVOQ Prototype

The EVOQ translator incorporates the main ideas introduced in this thesis. It performs the M-NFA translation on GraphLog queries, including those with multiple edges, and produces Datalog queries as output. It also incorporates features that are available in the Hy^+ system, as well as extensions specific to CORAL, including negation, aggregation and arithmetic operators.

5.1.1 Implementation

EVOQ has been implemented in C++, and comprises approximately 12000 lines of code. It has been compiled and tested on the Sun Sparc, Silicon Graphics and Linux platforms.

The method of translation implemented follows the stages of the M-NFA translation described in the previous chapter quite closely, and is displayed in Figure 5.1. This corresponds to the following process:

1. Ordering and Constraining
   (a) Depth-first search of query graph.
   (b) Edge table creation.
   (c) Constraining information generation.
2. Translation of each edge in the edge tables
   (a) Lexical analysis and parsing of edge regular expression.
   (b) Construction of automaton.
   (c) Translation including constraining (NFA-TRANSFORM).
   (d) Edge variable propagation.
   (e) Creation of edge module rules and passed variables rules.
3. Generation of rule for distinguished edge.

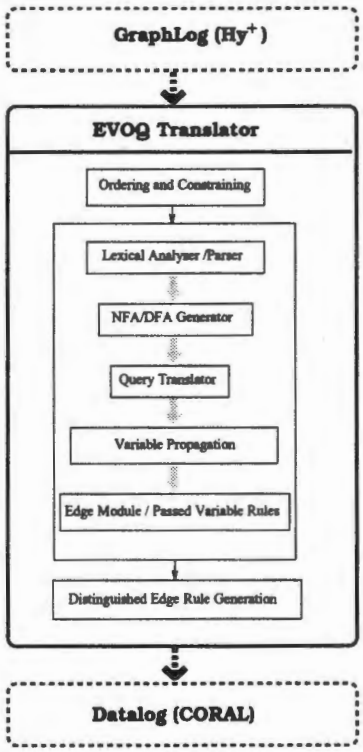

Figure 5.1: Components of EVOQ system

EVOQ translates GraphLog visual queries using a textual representation in the form of *GXF* files, which are the native Hy$^+$ format, or in an alternative format specific to EVOQ known as *Graph Data Format* (or GDF). The GDF representation of the define query in Figure 2.2 on page 9 is included in Appendix A. These files are parsed in EVOQ (the lexical analysis and parsing module) which allows GraphLog to be extended and translated independently of Hy$^+$. Some of these extensions are described in Section 5.1.2.

In order to allow maximum flexibility, and to be able to determine the effectiveness of the optimization methods, EVOQ includes command-line switches to allow the options to be toggled. For example, factoring or variable constraining may be turned off. It is also possible to prevent constraining of edge variables while permitting node variable constraining. In addition, a number of other options exist to allow tracing of the translation stages for informational or debugging purposes.

Another option allows the query to be translated by constructing a deterministic finite state automaton (DFA), instead of an NFA. This allows both translations to be compared,

in terms of the size of output, and to determine any difference in performance. The NFA-based translation is preferred, since the number of states of a DFA can be exponential in the number of states of an NFA. The algorithms for the construction of these automata are described in [ASU86] and [AU92].

Whereas the M-NFA translation requires an explicit automaton reversal in certain instances, the implementation performs an implicit automaton reversal, using a modified translation method. This eliminates one stage of the translation, and queries are translated faster.

5.1.2 Query Language Extensions

As indicated earlier, the EVOQ translator implements features of GraphLog such as negation and aggregation, amongst others. These extensions have not been formally included as part of the translation in the previous chapter, since they are beyond the scope of this thesis. However, in order to increase the range and expressive power of visual queries available to the user, a number of features have been included in the implementation. In Chapter 2, we mentioned that CORAL includes many extensions to Datalog, so the evaluation of queries using negation and aggregation presents no problem.

CORAL allows grouping by multiset and contains built-in aggregation functions, including **avg**, **count**, **max**, **min** and **sum**. In addition, the use of negation which is sufficient for GraphLog is supported. Therefore, the translation of GraphLog queries containing aggregation and negation operators to CORAL is relatively simple. An example of such a query is displayed in Figure 2.3 on page 13.

EVOQ allows arithmetic expressions and other constraints to be included on query edges. These constraints are expressed by adding a suitable expression enclosed in braces ({}) after a literal to the edge regular expression. This information is added to the base rule generated by the translation for each literal. The user may express constraints on the initial value of a variable, which then constrains all the subsequent values of the variable. The set of constraints that are possible are determined by the operations available in the database system.

Example 5.1 Consider the query in Figure 5.2. This demonstrates how arithmetic constraints are used to restrict the edge variables. This query produces the following Datalog program when translated with the M-NFA translation:

```
t_1_0(X1,X1).
```

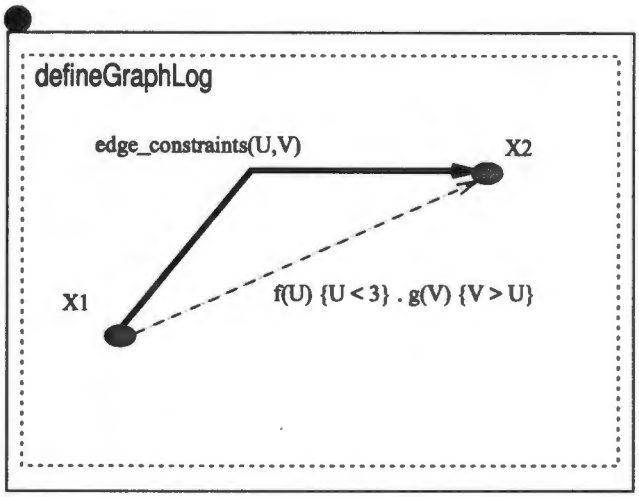

Figure 5.2: Query **edge_constraints**

```
t_1_1_U(X,Y,U) :- t_1_0(X,T), f(T,Y,U), U < 3.
t_1_2_U_V(X,Y,U,V) :- t_1_1_U(X,T,U), g(T,Y,V), V > U.
tie_1(X1,X2,U,V) :- t_1_2_U_V(X1,X2,U,V).

edge_constraints(X1,X2,U,V) :- tie_1(X1,X2,U,V).
```

We can see that the variable U is restricted to values less than 3 in the second rule. The third rule enforces the constraint that, in any answer tuple, the value for V must be greater than the U value.[1]

■

Another feature that is available is the ability to add multiple distinguished edges to a query. This does not add any power to the query, but saves the user having to reformulate and recompute the query in order to determine values for a second distinguished edge. This is especially useful if another query is then composed using both the distinguished edges.

We also allow the possibility of a distinguished edge that has one or both of its adjacent nodes remaining unlabelled, although the nodes adjacent to all other edges in the query

[1]This extension is not part of GraphLog and not incorporated in Hy$^+$. It is only available for queries parsed and translated by EVOQ. Since Hy$^+$ parses only GraphLog queries before saving them to *GXF*, queries using this extension cannot be expressed in Hy$^+$. Instead, a *GXF* text file is created manually and then translated by EVOQ. Therefore, the evaluation of the visual query in Figure 5.2 is only possible in Hy$^+$ if the parser was extended.

do contain labels. This allows queries where we are not interested in node values to be expressed. As a result, we may have a distinguished edge that is only attached to the rest of the query at one node, or alternatively "floats" independently of the rest of the query.

5.2 GIS Visualization and Querying

The Arc/Info Geographical Information System (GIS) has been developed by the ESRI (Environmental Systems Research Institute). Arc/Info works with spatial datasets, defined in terms of arcs and nodes. In this respect, the problem domain proved very suitable for visualization and analysis using Hy^+ and GraphLog. Assistance was obtained from the University of Cape Town's GIS department in obtaining local geographical data. This data was generated in a text format from the Arc/Info database and then filtered into a suitable format for CORAL using Unix text manipulation utilities such as awk [AKW88]. Some of these awk scripts are presented in Appendix B. Since one of the important objectives of GraphLog is its application to diverse problem domains, we describe the GIS domain in the same spirit.

The most basic element in Arc/Info is a node, defined in terms of its co-ordinates, namely longitude and latitude. A particular co-ordinate system is used for local data, where latitude is measured relative to the equator and longitude measured relative to the 23 degrees east longitude line. Nodes are connected by arcs. Both nodes and arcs may have attributes attached to them, for example, a city or road name. Polygons are defined in terms of a series of arcs enclosing an area and attributes attached to them, for example, the annual rainfall or the number of hospitals in the area.

Data is generated using a geographical projection where a curved surface is mapped onto a plane, and longitude and latitude become planar co-ordinates measured in meters. The various projections optimize particular parameters, for example correct distances for navigation, area for property valuations or shape for human perception in atlases. The projection used for our data is known as the Albers projection.

The GIS database that was used contains data related to a number of categories including rainfall, road systems, rivers, provinces and more. One of the problems was to unify these heterogeneous data sets using different arc and node/vertex info into a format suitable for use in Datalog evaluation. Since our investigation was mainly to determine the feasibility of using GraphLog queries on such data, we restricted our data set to a few categories. These included route, town and provincial information. The route data consists of road arcs

between points on the map, with each arc having a route name and a unique identification number within that route. A series of adjacent arcs sharing the same route name constitutes a route. The town data contains details of more than 2000 local towns, and their co-ordinates. A visualization of a section of the GIS database is shown in Figure 5.3.

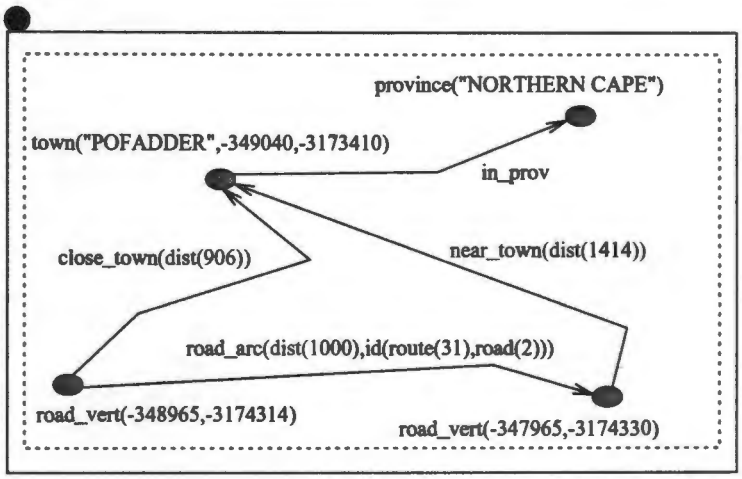

Figure 5.3: GIS database

Awk scripts were written to create two datasets; one to find towns within a user-specified distance (in meters) of a road arc (**near_town**), and another to find the closest road for every town (**close_town**). The third dataset contained details of towns and the provinces to which they belong (**in_prov**). The entire GIS Datalog database contains 19808 facts, of which 11743 are **road_arc** facts, 5422 are **near_town** facts and 2643 are **in_prov** facts.

The following examples demonstrate how this data may be queried in GraphLog. Such queries may be useful to travellers, urban planners and freight companies, amongst others.

Example 5.2 The query in Figure 5.4 finds all points on the map that can be reached from a particular point, using only two different routes U and V in sequence. This involves first finding the closure of all roads on route U, followed by a closure of all roads on route V. The distinguished edge **two_roads(U,V)** determines the route names as well the final points reached.

∎

Example 5.3 The query in Figure 5.5 finds all towns X2 that may be reached using a single route U, starting from some town 'Vredendal'. This query uses nested transitive closure

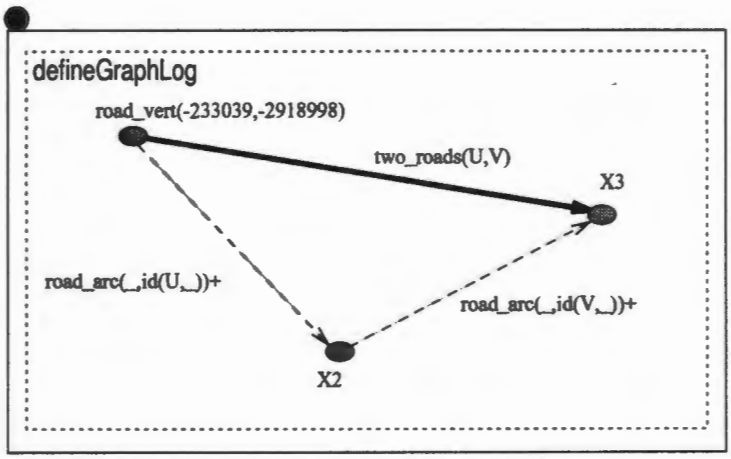

Figure 5.4: Query **two_roads**

operators (+). This permits an intermediate town to be reached using some sequence of road arcs (the inner closure) and then continuing along a different (disjoint) path on the same route U to reach another town (the outer closure). If the outer closure was omitted, this query would have a different meaning, since the path followed after reaching an intermediate town would have to be connected to the path used to reach the intermediate node.

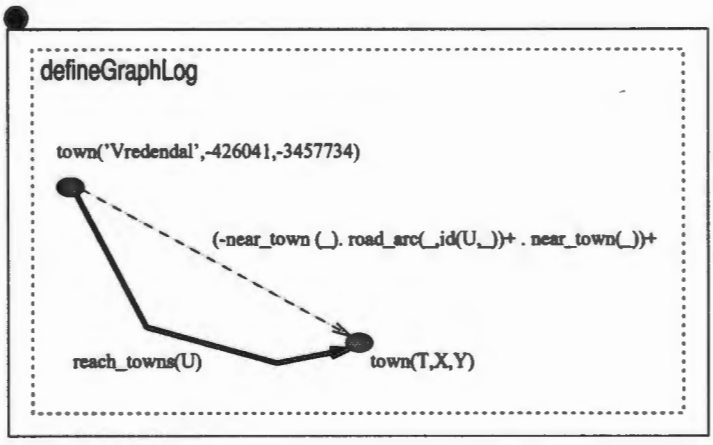

Figure 5.5: Query **reach_towns**

Example 5.4 Suppose that we replace the variable U in the **reach_towns** query in Figure 5.5 with the anonymous variable '_'. This changes the meaning of the query to find all towns reachable from town **'Vredendal'** using any route. This query shall be known as the **many_roads** query and the distinguished edge is from node **town('Vredendal')** to node **X2** with label **many_roads**.

∎

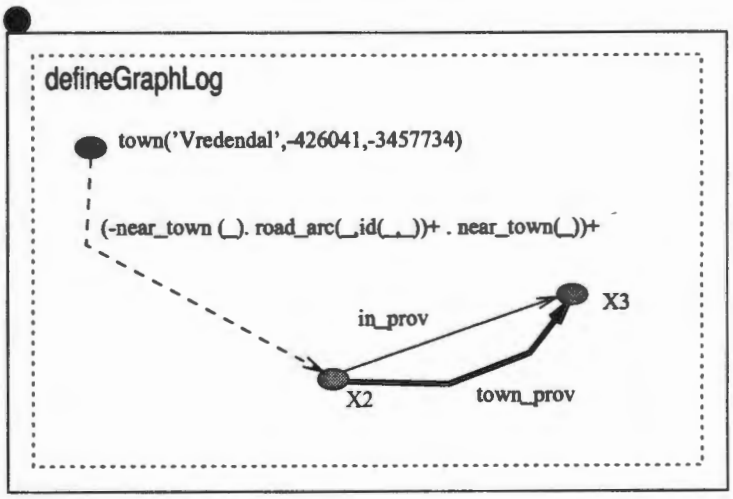

Figure 5.6: Query **roads_prov**

Example 5.5 The query in Figure 5.6 further extends the query defined in Example 5.4 to also determine the province to which each town that is found belongs.

∎

5.3 Class Library Examples

In this section, we describe a few example queries on a database containing details of relationships in the National Institutes of Health (NIH) public domain C++ class library. Most of the queries were introduced in [VW95], and are included here so that the performance speed-ups obtained in that paper may be compared to our results, including the M-NFA translation in Section 5.5. Since the environments and hardware under which these tests were performed were different, only relative comparisons are relevant.

The NIH database comprises 9124 facts, of which 2406 are **calls** facts, 2755 are **contains** facts, 3846 are **ref** facts, 72 are **subclass** facts, and 45 are **friend** facts. The **calls** facts include details of functions which are called by other functions. The **ref** facts contain details of variables that the functions reference. The **contains** facts describe the member functions of each class. The **subclass** facts describe classes and their subclasses. The **friend** facts detail the friend declarations between classes and classes or functions.

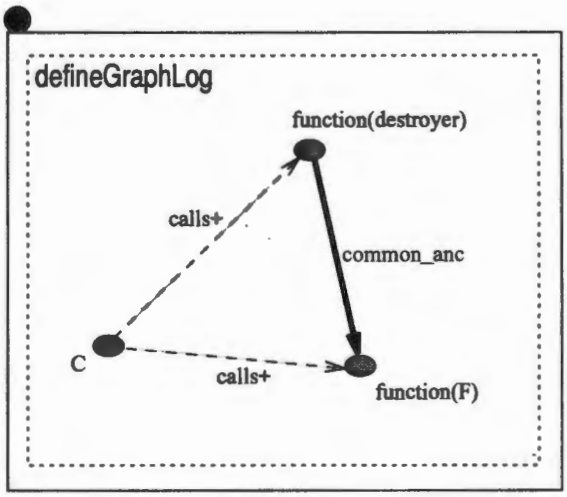

Figure 5.7: Query common_anc

Example 5.6 Consider the query in Figure 5.7. This query is a form of "common ancestor" query applied to function calls, and is similar to the query described in Example 2.1 on page 9. Here we are interested in all functions F that are called (indirectly or directly) by a function C, which also calls (indirectly or directly) the function **destroyer**. In order for the NFA-translation to apply to this query, it has to be rewritten to an equivalent single-edge query, with a regular expression calls+.-calls+. However, this rewriting is not necessary for the M-NFA translation, which can produce a (sometimes less efficient) Datalog program without having to rewrite the query as a single-edge query. Note that if the M-NFA translation is applied to the rewritten single-edge query, it would be as efficient as the NFA-translation. This is examined in Section 5.5.

■

Example 5.7 Suppose that we replace the constant function(destroyer) in Figure 5.7 with another constant function(schedule). This query is now identical to the query in Figure 2.2

on page 9. We refer to this query as the **calls_sched** query, where the distinguished edge has a label **calls_sched**.

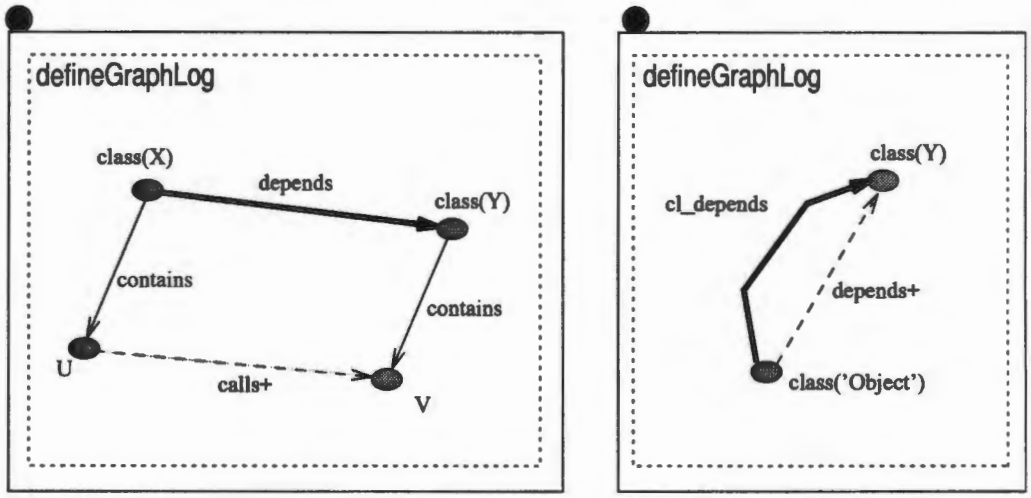

Figure 5.8: Query **cl_depends**

Example 5.8 The define queries in Figure 5.8 demonstrate how queries may be composed on existing queries. Building query hierarchies has the effect of reducing the amount of complexity required in the construction of each individual query. It also has the benefit of reuse, which may involve a comprehensive library of queries in each domain.

The first define query defines a relationship **depends** between classes X and Y, where X contains some function U which calls (directly or indirectly) some function V, contained in class Y.

The second define query defines a relationship **cl_depends**, which represents all classes Y which directly or indirectly depend on some class **'Object'**. This makes use of the **depends** distinguished edge defined in the first define query. The filter query displays the **cl_depends** edges as a directed graph, which the user may then query further.

Example 5.9 Query **cl_depends** in Figure 5.8 may be rewritten as an equivalent single-edge query, with a regular expression **(contains.calls+.-contains)+** from node **class('Object')**

to node class(Y). We will refer to this query as **se_depends**. The rewriting is essential for the NFA-translation to be applicable. However, there are performance benefits for both the M-NFA translation and the RE-translation if the single-edge form is used. This will be examined further in Section 5.5. Note that if we were to restrict variables U or V to only being functions, by renaming them function(U) and function(V) respectively, we could not translate this query using the NFA-translation. This has been discussed in Section 4.1.

■

5.4 Overlay Module Partitioning

The following queries are used to demonstrate the application of visual queries towards performance tuning during software development. These queries are described in [Con94] and [CMVW95]. Again, they are presented here for similar reasons to the class library examples in the previous section.

Data produced for a DOS application by a linker is queried with the intention of determining the best way of partitioning code into overlay modules which allow a reduction in memory requirements. These overlay modules are loaded independently into memory as the program executes, and may potentially overwrite other modules. This allows memory to be reused. However, the cost of such overlaying may be a great reduction in performance if the partitioning is not done carefully, and modules need to be repeatedly loaded. This problem is similar to the problem of virtual memory management on a multi-user operating system.

The overlay database comprises 5726 facts, of which 995 are `section_function` facts, 4593 are `calls` facts, 31 are `section_area` facts and 107 are `area_section` facts.

Example 5.10 The **overwrites_f** query is displayed in Figure 5.9. The meaning of the query is that a code section S1 overwrites some code section S2 if there are two distinct code sections, SP1 and SP2, at the top level of the same area A in memory, such that SP1 (resp. SP2) is an ancestor in the section-area tree of S1 (resp. S2) or is S1 (resp. S2) itself.

■

Example 5.11 Suppose that we modify query **overwrites_f** by replacing memory area variable A with a constant **deskArea**. This query will be known as **overwrites_b**, which is also the distinguished edge label.

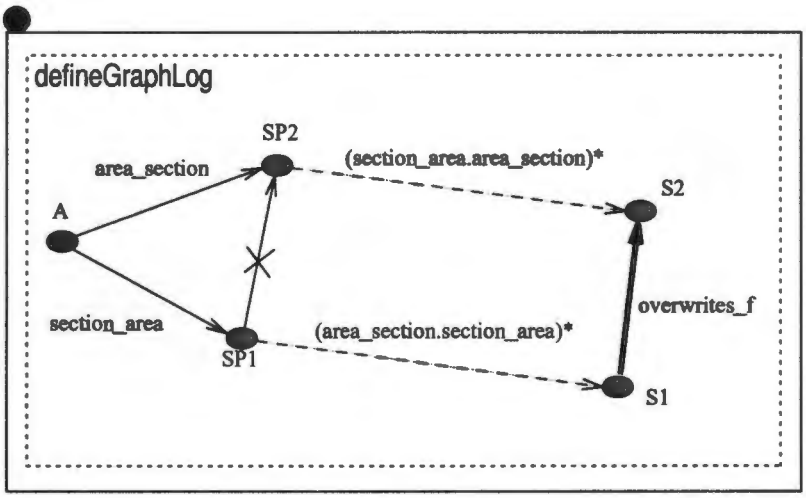

Figure 5.9: Query **overwrites**

Example 5.12 The **smashable_f** query is displayed in Figure 5.10. Here, we say that code section S2 is a smashable caller of function F1 if S2 contains a function F3 that calls F1, and S2 is not an ancestor in the section-area tree of the section S1 that contains F1.

Example 5.13 We modify the **smashable_f** query in Figure 5.10 to find all smashable callers where variable S2 is replaced with a constant **initSect**. We shall refer to this query as **smashable_b**, which will also be the label on the distinguished edge.

5.5 Performance Results

It is important to be able to quantify the differences in evaluation speed possible on "real-world" queries when using alternative translations and different optimization techniques. Therefore, a number of performance comparisons using the same CORAL back-end were done. Databases on which these tests were performed include the GIS database, as well as the NIH C++ class library and the overlay databases.

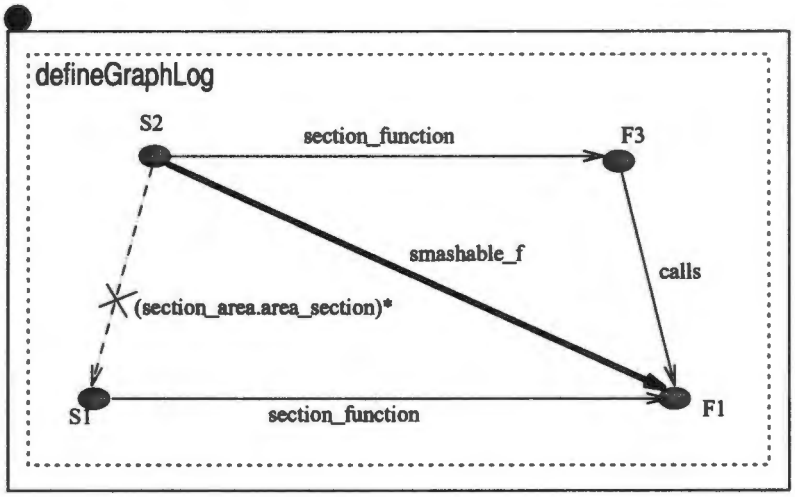

Figure 5.10: Query **smashable_caller**

These comparisons are intended to provide insight into the potential speed-up (or slow-down) that can be obtained when queries are translated with the RE-translation or the M-NFA translation. In addition, they are intended to demonstrate the consequences of using translation options available for the M-NFA translation such as not using variable constraining, as well as combining CORAL optimizations with factoring where feasible.

A similar set of comparisons was performed in [VW95] and [CMVW95], comparing the NFA-translation to the RE-translation for queries that could be written as single-edge queries. Here, we extend these comparisons to include queries that may only be expressed as multiple-edge queries, and cannot be translated by the NFA-translation. We also consider the consequences of evaluating a multiple-edge query instead of evaluating the query rewritten as an equivalent single-edge query. This demonstrates the potential for a further optimization of a subclass of multiple-edge queries which may be rewritten as single-edge queries.[2]

5.5.1 Presentation of Results

In this section, we discuss results obtained by performing the M-NFA translation, as well as the RE-translation, on the queries described in the previous three sections and evaluating the

[2]The results for the queries in [VW95] and [CMVW95] are recomputed, since a different environment and version of CORAL (CORAL v1.2) is used for our comparisons.

Table of GIS Results							
A	B	C	D	E	F	G	H
Query	RE	RE	MNFA	MNFA	MNFA	MNFA	MNFA
	no rw	context/ magic rw	factor constr no rw	factor no constr no rw	factor constr rw	factor no constr rw	factor single edge
gis/q1 two_roads	?	45.9 (m)	11.6	?	22.9	11.9	(10.9)
gis/q2 reach_towns	?	16.5 (m+sr)	11.2	n/a	n/a	11.2	11.2
gis/q3 many_roads	?	? (m+sr)	20.2	n/a	n/a	n/a	20.2
gis/q4 roads_prov	?	? (c+sr)	26.1	59.2	?	26.6	n/a

Table 5.1: GIS Results

Datalog program with CORAL. The Datalog code that has been generated for the queries is presented in Appendix C. In the previous chapter, we described how certain queries may be further optimized by variable and subgoal elimination, among other possibilities. Since these were not described as part of the general translation, we cover them in the following section. In order to avoid a potentially overwhelming table of results, we divide the queries into three tables. The first table contains results for queries performed on the GIS database, the second table contains results for queries that use the NIH database, and third table presents results for queries that use the overlay database. The entire set of results are presented in a single table in Appendix D. In all cases, the times reported are in seconds and are averaged over 10 runs on a lightly loaded Sun Sparc1+, with 45 MB of virtual memory, including 28 MB of physical memory.

Consider the results for the GIS queries in Table 5.1. We next describe the meaning of each column and the various symbols. The tables for the NIH and overlay database results are in an identical format, and the same description applies.

The use of a "?" symbol in any column indicates that the query terminates improperly, either because it does not terminate within 15 minutes of processing by CORAL, or terminates without generating any results after exhausting free store. The 15 minute limit was established as the maximum reasonable time that the user should expect to wait, particularly for interactive querying. In most cases, this limit represents an order of magnitude difference

compared to alternative translations for the same query which do terminate properly.

An "n/a" label in a column indicates that variable constraining is not possible, and usually applies to single-edge queries translated using the M-NFA translation. This is to avoid a misleading result suggesting that variable constraining has no effect, whereas no variable constraining occurs in a single-edge query.

Column A indicates the database name followed by the query name in the first row of each table entry, and the query label (derived from the distinguished edge) in the second row.

Column B contains results that are obtained by evaluating the Datalog query generated by the RE-translation without performing any CORAL optimization. The large size of the databases used means that most of these results are unavailable due to improper termination.

Column C can be considered as the "best-case" RE-translation result. The major rewriting techniques available in CORAL are attempted, namely context rewriting and supplementary magic sets and only the better result is retained. The letters "c" or "m" in parentheses after the result indicate whether context rewriting or supplementary magic sets was faster. In certain instances, depending on the binding patterns of Datalog query patterns, context rewriting is not possible. In this case, CORAL defaults to supplementary magic sets. The letter indicating the rewriting method takes this into account. CORAL does not always choose the best subgoal ordering during evaluation. As a result, the Datalog program has been manually rewritten to improve the ordering of the subgoals. The letters "sr" after a result indicate that subgoal re-ordering was also attempted, where possible. This problem is also discussed in [VW95].

Columns D–G contain results produced by using the M-NFA translation, with factoring where possible. For columns D and E, the CORAL `@no_rewriting.` annotation was used, so that no rewriting was performed by CORAL. However, columns D and E differ in that variable constraining was switched on for queries in column D and off for queries in column E. Columns F and G show the results of performing the supplementary magic rewriting using the `@sup_magic.` annotation in CORAL. Here, column F differs from column G in that variable constraining is not performed by the M-NFA translation for queries in column G.

Column H contains the results of single-edge queries that have been evaluated using the M-NFA translation. These values can be regarded as the same as those produced by the NFA-translation, since the NFA-translation only operates on single-edge queries. Some of the results are generated by modifying the source query by rewriting it as a single-

Table of NIH Performance Results							
A	B	C	D	E	F	G	H
Query	RE	RE	MNFA	MNFA	MNFA	MNFA	MNFA
	no rw	context/ magic rw	factor constr no rw	factor no constr no rw	factor constr rw	factor no constr rw	factor single edge
nih/q1 common_anc	?	105 (m) 63.8 (c+sr)	79.8	104	622	82.3	(2.92)
nih/q2 calls_sched	?	90.1 (m) 10.1 (c+sr)	0.85	85.7	2.89	2.29	(0.52)
nih/q3 cl_depends	?	?	?	?	?	83.4	(3.57)
nih/q4 se_depends	?	63.9 (m+sr) ? (c+sr)	3.57	n/a	n/a	n/a	3.57

Table 5.2: NIH Results

edge query. This process has to be performed manually, and is not guaranteed to be possible for all queries. However, these values are reported to compare the overhead of the M-NFA translation translating a query in a multiple-edge form, against evaluating the equivalent single-edge query. Where such rewriting or any other modification is necessary, the results are displayed in parentheses.

5.5.2 Discussion of Results

There are a number of interesting points that have emerged from the test results. In the following, we discuss the results obtained for all the test queries, and identify any trends or noteworthy aspects of these results.

If we consider optimization choices for the RE-translation alone, we find that it is difficult to obtain a fully general rule. One heuristic might be always to attempt context rewriting, since CORAL defaults to supplementary magic sets when this is not possible. However, our results in Table 5.2 indicate that for query **se_depends** (Example 5.9) on the NIH class library, this would not be a wise choice. In fact, the context rewritten program does not terminate after 15 minutes, whereas the program rewritten using supplementary magic sets does terminate in a significantly faster 64.9 seconds. It has been shown [MP91] that context rewriting can result in an order of magnitude slowdown compared to magic sets.

Table of Overlay Results							
A	B	C	D	E	F	G	H
Query	RE	RE	MNFA	MNFA	MNFA	MNFA	MNFA
	no rw	context/ magic rw	factor constr no rw	factor no constr no rw	factor constr rw	factor no constr rw	factor single edge
overlay/q1 overwrites_f	25.3	22.4 (m) 22.7 (c)	21.3	22.4	?	25.6	(19.8)
overlay/q2 overwrites_b	8.24	3.56 (m) 2.83 (c)	2.47	6.38	3.41	2.92	(2.24)
overlay/q3 smashable_f	14.6	89.5 (m)	32.5	64.7	n/a	n/a	(11.1)
overlay/q4 smashable_b	1.68	2.32	2.12	39.4	n/a	n/a	(0.73)

Table 5.3: Overlay Results

Another heuristic that one might be tempted to use would be always to select a CORAL optimization method (i.e. either supplementary magic or context rewriting) for the standard query generated by the RE-translation. In light of the results for **smashable_f** query (Figure 5.10) on the overlay database, we observe that this is not always successful (Table 5.3). In fact, the rewritten query (C) is more than 6 times slower than the standard query (B) without any CORAL optimization.

It appears that some combination of the suggested heuristics needs to be used to ensure efficient evaluation. Most of the CORAL rewriting methods are most effective if there are bound values in the query. Our results agree with this. Therefore, we should use a CORAL optimization method whenever queries contain bound values, and no rewriting method otherwise.

We now consider the results obtained for the M-NFA translation. Suppose that we compare the results of queries which use variable constraining (D), to those that do not (E). We see that very large speedups can be obtained when using constraining, particularly for the **calls_sched** query (Example 5.7) on the NIH database The results for the least effective variable constraining (query **overwrites_f** in Table 5.3) suggest that our assertion in Chapter 4, that the overhead of performing variable constraining was not comparable to the potential pay-off, appears to be correct.

If we compare the variable constrained queries (D) to queries that use supplementary magic

sets in conjunction with factoring (F), we notice that the results are generally similar with one glaring exception, namely the **cl_depends query** on the NIH database. In this case, the M-NFA translation with variable constraining does not terminate, whereas the query using supplementary magic computes results in 83 seconds. However, an examination of the actual query structure explains this discrepancy. The **cl_depends** query (Figure 5.8) is the only query that consists of multiple define queries where one query is composed on the result of the other. As a result, the variable constraining algorithm does not consider the **class('Object)'** binding which is passed downwards to the **depends** query, but considers each query in isolation. This is recognized by the magic set optimization and in conjunction with factoring, the query does terminate.

This suggests a possibility for further research into extending the optimization methods to analyse queries and determine whether optimization across queries that have common edges is possible, and the best way of translating these queries. This is not a trivial problem, particularly when closure queries are involved.

It appears that the method of variable constraining, while not as general as magic sets, is comparable where such constraining is performed. Variable constraining, which is much simpler in concept, mimics the sideways information passing of magic sets so that queries translated with the M-NFA translation are similar in performance to those where magic sets is used instead. This would suggest that major performance differences are the result of factoring. Another important point is that for multiple-edge queries, some form of optimization between edges is essential to obtain noteworthy speedups. If one was to choose between variable constraining and supplementary magic sets, in conjunction with factoring, the generality of the magic method would favour it. However, this should be viewed in the context of variable constraining being a comparatively simple alternative, which is a built-in part of the translation.

The results of the attempt to combine both variable constraining and supplementary magic sets (F) indicate that this results in a conflict. No answers are obtained for any of the queries that take more than a few seconds to compute for each of the methods in isolation. What is evident is that both methods should not be used together.

The column containing results for queries which are written as single-edge queries and then translated with the M-NFA translation (H) suggests that the sometimes huge speedups that can be obtained by factoring an entire query may justify such rewriting. For example, results for the queries in Table 5.2 indicate decreases in evaluation time for a query rewritten using a single edge by a factor of approximately 20 over all other translation methods. This

speed-up is possible because of the presence of a constant in a node of each query which results in the factoring translation of the entire query. This reduces the arity of every IDB predicate in the Datalog program that is generated. When translated as a multiple-edge query, the program produced by the M-NFA translation only contains factored IDB predicates for edges with an adjacent constant node. Therefore, all the IDB predicates in the program are not necessarily factored, unlike the translation of the single-edge query. The rewriting methods used by CORAL are either not applicable to certain queries or do not result in a reduction in arity of IDB predicates.

Although such structure-dependent rewriting should not be the responsibility of the user, particularly in a system where user expertise is to be minimized, the opportunity for such query rewriting at the visual level exists. This may be compared to subgoal re-ordering, for example, where it is far more difficult for the user to correct the problem from within Hy^+, even if the user recognizes the problem. Nevertheless, the reduction of multiple-edge queries into equivalent minimal queries should be part of the translation in an ideal situation.

Finally, we compare the best results obtained for the M-NFA translation with those of the RE-translation for the general case (which excludes the single-edge rewritten queries). The best results for the RE-translation and the M-NFA translation are selected from the preceding tables in this section and displayed in Table 5.4. Column B contains the number of answers for each query. The RE-translation results are in column C, and the M-NFA translation results are in column D.

The speedup figure (E) reveals that for certain queries, the M-NFA translation is an order of magnitude faster, whereas for other queries, the performance results are similar. However, there are results where the RE-translation is faster, and the reasons for this are described in [CMVW95]. In fact, for query **roads_prov** (which cannot be rewritten as a single-edge query for the NFA-translation), the only translation method that generates results without improper termination is the M-NFA translation. None of the CORAL rewriting techniques is effective in this case. This is a compelling reason why the M-NFA translation should be available as a translation alternative.

However we do need to examine why the RE-translation (with no rewriting) is faster than the M-NFA translation. For example, results for query **smashable_f** in Table 5.3 demonstrate this situation. One of the major requirements for optimization in the M-NFA translation is the presence of node constants in the visual query, so that factoring is possible. We note that query **smashable_f** in Figure 5.10 does not contain any such node constants.

Another reason why the RE-translation is sometimes more efficient is revealed by examining

Best RE vs Best MNFA Comparison				
A	B	C	D	E
Query	No. Answers	Best RE	Best MNFA	**Speed-up (C/D)**
gis/q1 two_roads	189	45.9	11.6	3.97
gis/q2 reach_towns	11	16.5	11.2	1.53
gis/q3 many_roads	384	?	20.2	?
gis/q4 roads_prov	415	?	26.1	?
nih/q1 common_anc	507	63.8	79.8	0.80
nih/q2 calls_sched	76	10.1	0.85	11.9
nih/q3 cl_depends	32	?	83.4	?
nih/q4 se_depends	32	63.9	3.5	18.2
overlay/q1 overwrites_f	6684	22.4	21.3	1.05
overlay/q2 overwrites_b	352	2.83	2.47	1.15
overlay/q3 smashable_f	2001	14.6	32.5	0.45
overlay/q4 smashable_b	95	1.68	2.12	0.79

Table 5.4: RE vs MNFA Results

the method of translation. The M-NFA translation generates rules from transitions in an NFA. Since each transition is labelled with a single EDB predicate name, each Datalog rule generated by the M-NFA translation contains one EDB predicate. On the other hand, the RE-translation generates a Datalog program recursively according to the structure of the regular expression. As a result, it is possible to have rules containing more than one EDB predicate (for example, concatenation) provided no CORAL rewriting occurs. Subsequent computation would occur on the join of these EDB predicates, which might be relatively small. This would give the RE-translation a performance advantage, which factoring and other rewriting cannot compensate for, irrespective of the presence of constants.

Of course, the opposite could occur, in that the size of the joins could be much larger, and the RE-translation would incur a severe performance penalty. This depends on the particular dataset being used and the nature of the query. Therefore, the converse is also possible where the M-NFA translation is faster than the RE-translation for queries which do not contain constants. The results for query overwrites_f in Table 5.3 demonstrate this. These possibilities are discussed in [CMVW95].

5.5.3 Summary

Our results might suggest the M-NFA translation should generally be used instead of the RE-translation. Such an inference should be viewed with caution due to the complexity of finding a general solution. As discussed in the previous section, it has been shown that the NFA-translation (and hence the M-NFA translation) can perform much worse than the standard RE-translation (without any optimization) for certain queries [CMVW95]. This paper demonstrates how the presence of constants in a query does not guarantee that the NFA-translation is superior. Instead, the choice of translation is guided by the "shape" of the database graph, which essentially involves the sizes of joins computed by the different translations, referred to as the "join selectivity" of the relations.

This result is of concern with respect to an ideal querying system where the system attempts to choose the best translation method and optimization possibility. This is because the required information related to the shape of the graph can only be fully determined by a potentially expensive data analysis. In [CMVW95], cost functions which approximate the size of joins are proposed to assist with the choice of translation. These functions are used as a heuristic to indicate which method should be used. However, this is based on approximations and cannot guarantee the correct choice of translation method.

Although counter-examples have been revealed that illustrate the difficulty of choosing

between translation methods and optimization techniques, it is important not to lose sight of the benefits of optimization. For certain queries, evaluation would not be possible without using a rewriting method. Generally, it appears that the worst case situation for a rewritten program is rarely an order of magnitude slowdown, whereas significant speed-ups can be obtained for most queries. If a query is to be executed several times on a similar dataset, it would make sense to try the various alternatives since the time saving could be substantial. This motivates the use of switches to direct the translation method.

Our results for the M-NFA translation suggest that where possible, the visual query should be expressed using a regular expression on a single edge. If this is not possible, the best option is to perform the M-NFA translation with either variable constraining or magic sets. The hypothesis that the M-NFA translation outperforms the RE-translation for queries containing constants is confirmed by most of our test cases which we consider to be representative. Similarly, for queries containing no constants, the RE-translation is usually preferred to the M-NFA translation. However, despite the presence or absence of constants in nodes, examples have been constructed to show that each translation can outperform the other.

5.5.4 Results for further optimization

In this section, we present some results for certain queries that have been further optimized by the methods introduced in Section 4.6. This involves the elimination of redundant variables, as well as subgoals in the rule derived for the distinguished edge. This has not been incorporated into the implementation, and the Datalog queries are produced by a manual translation. In a sense, these results represent a "best-case" application of the M-NFA translation method, where the overhead (introduced by the general method) is removed from the special cases. This is not possible for every query, so only the queries where improvements were obtained are displayed in Table 5.5.

The most significant speed-up is obtained for query common_anc (Figure 5.7). Since this query is similar to query calls_sched (described in Example 5.7), we consider the reasons for the comparatively large speed-up. Most of the benefits for query calls_sched result from constraining, using variable C. Therefore, the effect of variable and subgoal elimination does not substantially reduce the evaluation time. For query common_anc, the set of constraining values is much larger, and there is a much smaller amount of constraining. The variable elimination in the Datalog program has a far more significant impact here. The reduction in arity due to variable elimination results in a factored program, which is substantially

MNFA vs Optimized MNFA Results				
A	B	C	D	E
Query	MNFA	Opt. MNFA	Optimization	Speed-up (B/C)
gis/q4 roads_prov	26.1	19.9	subgoal elim.	1.3
nih/q1 common_anc	79.8	3.9	variable elim. subgoal elim.	20.5
nih/q2 calls_sched	0.85	0.34	variable elim. subgoal elim.	2.5

Table 5.5: Additional Optimization Results

faster. This is further improved by subgoal elimination. The performance of this query is comparable to the results obtained for translating the equivalent single-edge common_anc query, displayed in column H of Table 5.2.

Chapter 6

Conclusion

A number of issues related to the evaluation of visual queries in GraphLog have been examined in this thesis. These include:

- The introduction of graph semantics as an alternative way of formalizing the meaning of queries based on an intuitive understanding.
- The description of the M-NFA translation, extending the factoring NFA-translation which is an alternative to the RE-translation used in Hy$^+$.
- The efficient evaluation of GraphLog queries by incorporating optimization techniques into the translation of these queries to Datalog.

In Chapter 1, we introduced the fields of visual queries and logic database systems and described the value of attempting to integrate these fields in a system such as Hy$^+$. The need for clear and well-defined semantics, as well as an efficient way of evaluating visual queries, was motivated.

In Chapter 2, we described the necessary background to the issues of query semantics and translation. These issues require an understanding of GraphLog, Datalog and optimization techniques for Datalog evaluation. We also provided details of the RE-translation which is the standard method of translation, as well as the alternative NFA-translation method.

Graph semantics were defined in Chapter 3. This involved formalizing the intuitive understanding of GraphLog queries as "finding paths in graphs". We showed that for certain queries involving the Kleene closure operator, the Datalog semantics of the RE-translation

do not correspond to the graph semantics. Modifications to the RE-translation were proposed to resolve the differences and a proof of the equivalence of this semantics to the graph semantics was given.

In Chapter 4, we described the M-NFA translation which extended the NFA-translation presented in Chapter 2. The expressive power of queries that may be translated with the NFA-translation is increased by permitting the translation of multiple-edge queries, while preserving the factoring property. We also introduced an optimizing technique called variable constraining for passing information between edges in a multiple-edge query.

The evaluation of "real life" queries was a primary motivation for the development of GraphLog. In Chapter 5, we discussed the implementation of the EVOQ system that performs the M-NFA translation. In addition, the performance benefits of translating queries with the M-NFA translation in practical environments were explored. Queries on databases from diverse real-world domains were described, and the performance results presented.

One of our focuses in this thesis was the evaluation of visual queries. This raises the issue of defining the meaning of such queries, so that any answers generated would correspond to the natural interpretation of a query. We introduced graph semantics, defined in terms of finding paths in graphs, which contrasts with the original semantics of GraphLog defined by a translation to Datalog. The graph semantics successfully combines the natural interpretation with a formal definition of semantics for GraphLog.

Another focus involved searching for more efficient methods of evaluating visual queries, particularly those involving query constants and multiple edges. This resulted in the development of the M-NFA translation which permits the translation of a more general class of queries than the NFA-translation. The principles of factoring and variable constraining for efficient evaluation were incorporated into the M-NFA translation. We verified the efficiency of our method by comparing performance results for the M-NFA translation against the standard RE-translation. It was found that the variable constraining algorithm for visual queries with multiple edges is comparable to the magic rewriting method used by CORAL. In addition, the factoring translation of edges containing constants also results in significant performance benefits.

6.1 Future Work

The issues that have been explored in this thesis reveal possibilities for further investigation and refinement. These include:

- Use of additional information to choose translation method.

  Information such as cost estimates could be used to decide if the M-NFA translation or the RE-translation should be used for more efficient evaluation of a particular query. This has been explored in [CMVW95]

- Further optimization of queries during translation.

  The additional optimization possibilities discussed in Section 4.6 need to be investigated further to determine the best way of including them into the M-NFA translation. This would require an attempt to preserve the declarative nature of queries while using the best possible translation. In other words, the user should ideally not have to be knowledgeable about Datalog or optimization techniques. This is one of the reasons why optimization built into the translation is desirable.

- The extension of optimizations to include filter queries.

  The method used for translating filter queries in Hy^+ is not very efficient, and all filter queries are translated to define queries. This offers great potential for developing more efficient methods to optimize the translation of filter queries. In fact, this possibility is suggested in [CMV94] where the "naive" translation of filter queries is described.

- Extension of variable constraining to the RE-translation.

  As mentioned in Section 4.9, the variable constraining optimization described for multiple-edge queries is not specific to the M-NFA translation. It could be implemented as part of the RE-translation without much effort.

- The inclusion of negation, aggregation and path summarization.

  As mentioned previously, each of these areas is a substantial field of research. Although features are included to support these in the implementation, it would be valuable to consider these more carefully within the context of efficient evaluation. These areas have been examined from a GraphLog perspective in [Con89] and [CM92], but there is scope for further research.

- The translation of blobs in hygraphs.

  While only comprising a small part of the RE-translation, blobs are very useful visual abstractions. A number of intuitive containment relationships may be expressed visually using them. A general implementation of a translation system for GraphLog should be capable of handling queries containing blobs.

- Extending the graph semantics to include all GraphLog visual formalisms.

  The graph semantics should be generalized to define the meaning of all Hy$^+$ queries particularly for queries that involve blobs and filter queries. Since the semantics of Datalog extensions such as negation and aggregation are not standard across logic database systems, it would also be useful to define a fixed semantics for these features, again attempting to preserve the intuitive understanding of queries where possible.

- Database and visual query system administration.

  Although some of the topics regarding query evaluation are largely discussed from a theoretical perspective, the context of this work with regard to a "real-world" visualization and query evaluation system should not be forgotten. Such a system, which Hy$^+$ aims to be, would be used to store large volumes of visual data, in addition to a library of commonly used queries in the particular enterprise. The maintenance of this system would be the task of an administrator, whose duties would include updating data and deciding on the best optimization methods for different queries, in conjunction with the translations that are possible.

  The user would then be in a position to interact with the system by using and extending queries which had been pre-defined, with little concern about choosing appropriate translations or optimization details.

- Higher-order extensions to the visual query system (HiLog)

  There are potentially useful queries that cannot be expressed in Datalog but can be expressed using HiLog [CKW89], which is an extension of the predicate logic used in Datalog. HiLog has a higher-order syntax which permits the use of arbitrary terms in positions where predicates, functions and atoms occur in Datalog. This is a generalization of the use of structured or compound terms in literals. Schema-level queries may therfore be posed on a database.

  The extension of GraphLog to permit such potentially useful queries is feasible. However, the translation of these queries would need to be carefully examined, particularly concerning efficiency, since the generalization of the query structure can have significant performance consequences.

Appendix A

File Formats

A.1 *GXF* Format

The *GXF* format for the query displayed in Figure 2.2 on page 9 is presented below:

```
; nodes: 7
; edges: 4
; blobs: 2

(GRAPH
    (ID "anish@juliet: 18 July 1995 2:34:51 pm" )
    (BOUNDS
        (RECTANGLE
            (XY 0 0 )
            (XY 1 1 )))
    (LAYOUTS
        (LAYOUT
            (NAME "circle" )
            (ALGORITHM "circle" ))
        (LAYOUT
            (NAME "random" )
            (ALGORITHM "random" ))
        (LAYOUT
            (NAME "grid" )
            (ALGORITHM "grid" "minsep" "20" "20" "20" ))
        (LAYOUT
            (NAME "spring" )
            (ALGORITHM "spring" ))
        (LAYOUT
            (NAME "row" )
            (ALGORITHM "1d" "x" )))
    (NODES
```

```
(NODE
    (ID "1" )
    (LABEL "function(schedule)" )
    (POINT
        (XY 0.685956 0.297173 ))
    (DISTINGUISHED )
    (BLOBS_REGION ))
(NODE
    (ID "2" )
    (LABEL "function(F)" )
    (POINT
        (XY 0.782529 0.673712 ))
    (DISTINGUISHED )
    (BLOBS_REGION ))
(NODE
    (ID "3" )
    (LABEL "function(F)" )
    (POINT
        (XY 0.38898 0.647494 ))
    (DISTINGUISHED )
    (BLOBS_REGION ))
(NODE
    (ID "4" )
    (LABEL "function(schedule)" )
    (POINT
        (XY 0.285085 0.285545 ))
    (DISTINGUISHED )
    (BLOBS_REGION ))
(NODE
    (ID "5" )
    (LABEL "" )
    (POINT
        (XY 0.62212 0.0929487 ))
    (BLOBS_REGION )
    (BOUNDS
        (RECTANGLE
            (XY 0.62212 0.0929487 )
            (XY 0.921659 0.817308 ))))
(NODE
    (ID "6" )
    (LABEL "" )
    (POINT
        (XY 0.0542083 0.0929487 ))
    (BLOBS_REGION )
    (BOUNDS
        (RECTANGLE
            (XY 0.0542083 0.0929487 )
            (XY 0.547296 0.830128 ))))
(NODE
    (ID "7" )
    (LABEL "C" )
    (POINT
        (XY 0.110278 0.591043 ))
```

```
        (BLOBS_REGION )))
    (BLOBS
        (BLOB
            (FROM "6" )
            (TO "7" )
            (TO "3" )
            (TO "4" )
            (LABEL "defineGraphLog" )
            (NODES_REGION )
            (BOUNDS
                (RECTANGLE
                    (XY 0.0542083 0.0929487 )
                    (XY 0.547296 0.830128 ))))
        (BLOB
            (FROM "5" )
            (TO "1" )
            (TO "2" )
            (LABEL "showGraphLog" )
            (NODES_REGION )
            (BOUNDS
                (RECTANGLE
                    (XY 0.62212 0.0929487 )
                    (XY 0.921659 0.817308 )))))
    (EDGES
        (EDGE
            (FROM "7"
                (ATTACH
                    (XY 0.375 0.25 )))
            (TO "4"
                (ATTACH
                    (XY 0.5 0.625 )))
            (LABEL "calls+" )
            (POINTS
                (XY 0.195435 0.352564 )))
        (EDGE
            (FROM "1"
                (ATTACH
                    (XY 0.5 0.375 )))
            (TO "2"
                (ATTACH
                    (XY 0.375 0.625 )))
            (LABEL "calls_sched" )
            (DISTINGUISHED ))
        (EDGE
            (FROM "4"
                (ATTACH
                    (XY 0.125 0.625 )))
            (TO "3"
                (ATTACH
                    (XY 0.75 0.75 )))
            (LABEL "calls_sched" )
            (DISTINGUISHED ))
        (EDGE
```

```
            (FROM "7" )
            (TO "3"
                (ATTACH
                    (XY 0.375 0.625 )))
            (LABEL "calls+" )
            (POINTS
                (XY 0.28388 0.647436 )))))
; nodes: 7
; edges: 4
; blobs: 2
```

A.2 EVOQ GDF Format

The GDF format is used by EVOQ as a textual representation of visual queries, as a simple (though less general) alternative to *GXF*. An example of GDF is displayed below for the define query in Figure 2.2 on page 9:

```
Nodes 3

Node: 0 C
Node: 1 function(F)
Node: 2 function(schedule)

Edges 2

0 2
calls+
0 1
calls+

DistEdges 1

2 1 calls_sched(function(schedule),function(F))
```

Appendix B

Awk Scripts

B.1 Route information

The following script is used to generate information about routes and roads (sections of a route). The geographical area covered may be restricted by specifying co-ordinates.

```
#
# awk program to create datalog facts from a GIS roads database
# (for example file roads.txt)
#
# Usage: nawk -f roadfile.awk [roadfile]
#
# input file is series of vertices separated in records for each route
#
# sample output line is:
#
# road_arc(road_vert(20,30),road_vert(30,28),dist(10),id(route(1),road(2)).
# route_arc(road_vert(10,40),road_vert(30,33),dist(21),id(route(2))).
#
BEGIN {
#   default values for x and y co-ordinates to eliminate
    x_c_default = 0
    x_dir_default = "west"
    y_c_default = 0
    y_dir_default = "south"
    if (ARGC == 1) {
        printf("\nUsage : roadfilt2 [road_file_name] west | east [x-coord]\
        north|south [y-coord] \n")
        printf("\nOtherwise default values will be used :\n")
        printf("\nArea of Interest: %d %s %d %s\n",x_c_default,\
        x_dir_default,y_c_default,y_dir_default)
        exit
    }
```

```
    # values explicitly defined
    if (ARGC == 6) {
        x_dir_default = ARGV[2]
        ARGV[2] = ""
        x_c_default = ARGV[3]
        ARGV[3] = ""
        y_dir_default = ARGV[4]
        ARGV[4] = ""
        y_c_default = ARGV[5]
        ARGV[5] = ""
    }
    printf("\n/* Selective Road/route information (roadfilt2.awk) */\n")
    printf("\n/* Road file : %s   Interest Area : %d %s : %d %s */\n\n"\
    ,ARGV[1],x_c_default,x_dir_default,y_c_default,y_dir_default)
    rec_switch = 0
    road_id = 0
    road_found = 0
    end_of_route_flag = 0
}
# main pattern matching routines
{
# flag once record found that violate constraints
    if (NF == 2 && x_dir_default == "west" && end_of_route_flag == 0) {
        # if value is greater (east of) constraint
        if ($1 > x_c_default){
            end_of_route_flag = 1
        }
    }
    if (NF == 2 && x_dir_default == "east" && end_of_route_flag == 0) {
        # if value is greater (west of) constraint
        if ($1 < x_c_default){
            print $1, x_c_default
            end_of_route_flag = 1
        }
    }
    if (NF == 2 && y_dir_default == "south" && end_of_route_flag == 0) {
        # if value is greater (west of) constraint
        if ($2 > y_c_default){
            end_of_route_flag = 1
        }
    }

    if (NF == 2 && y_dir_default == "north" && end_of_route_flag == 0) {
        # if value is greater (west of) constraint
        if ($2 < y_c_default){
            end_of_route_flag = 1
        }
    }
# find road number
    if (NF == 1 && $1 != "END" && rec_switch == 0) {
        # now on another route
        node_pos  = 0
        start_node = 1
```

```
        rec_switch = 1
    }
# end of roads file
    else if (NF == 1 && $1 == "END" && rec_switch == 0) {
        exit
    }
# end of record
    else if (NF == 1 && $1 == "END" && rec_switch == 1) {
        if (road_found == 1) {
            printf("route_arc(road_vert(%d,%d)",StartX,StartY)
            # Source is used because actually is last dest node
            printf(",road_vert(%d,%d)",SourceX,SourceY)
            dist = sqrt ((SourceX - StartX) ^ 2 + (SourceY - StartY) ^ 2)
            printf(",dist(%d),id(route(%d))).\n",dist,route_id)
            ++route_id
        }
        rec_switch = 0
        # reset road_found flag
        road_found = 0
        # reset end of route flag because end was processed
        end_of_route_flag = 0
    }
# source node/vertex of a new record
    if (NF == 2 && start_node == 1) {
        SourceX = $1
        SourceY = $2
        StartX = $1
        StartY = $2
        road_id = 0
        start_node = 0
    }
    else if (NF == 2 && start_node == 0 && end_of_route_flag == 0) {
        # End node of previous node
        ++road_id
        #  test here to check if road_arc satisfies constraint
        printf("road_arc(road_vert(%d,%d)",SourceX,SourceY)
        printf(",road_vert(%d,%d)",$1,$2)
        dist = sqrt(($1 - SourceX) ^ 2 + ($2 - SourceY) ^ 2)
        printf(",dist(%d),id(route(%d),road(%d))).\n",dist,route_id,road_id)
        # Save Dest Variables  as new Source Variables
        SourceX = $1
        SourceY = $2
        road_found = 1
    }
}
```

B.2 Closest Towns

The following script is used to generate details of each town and its closest road vertex from files containing road and file information. A similar script is used to find towns that are

within a user-specified distance of a road vertex and facts near_town are generated instead.

```
# awk program to filter records which contain town and road information
# (roads.txt) and (towns_p2.txt). The output is the closest road vertex
# for every town in the database
#
# This is from the towns in provinces file as opposed to a straight
# towns file (the fields for the town info are in $7, $11, & $12)
#
# Usage: nawk -f close_town.awk [townfile] [road_file]
#
# Sample format:
#
# close_town(road_vert(816861,-2612953),town("SHINGWEDZI",857505,-2522166),
# dist(99469)).
#
# for debugging, set the debug flag to 1
BEGIN {
# starting town tuple (in case one needs to restart)
    start_town = 1
# debug flag
    debug = 0
    if (ARGC != 3) {
        for (i = 1; i < ARGC; i++) {
            printf"%d %s\n",i,ARGV[i]
        }
        printf("\nUsage : close_town [town_file_name] [road_file_name]\n")
        printf("\nFind the closest vertex to a given town\
        and its distance\n\n")
        exit
    }
    # kill the values in ARGV so awk doesn't think they are input files
    rfilename = ARGV[2]
    ARGV[2] = ""
    printf("\n/* Closest Town near road vertex (close_town.awk) */\n")
    printf("\n/* Road file : %s Town file : %s */\n\n",rfilename,ARGV[1])
# now read in roads array
# just a big array of co-ordinates
    FS = " "
    eof_flag =  getline < rfilename
    vert_count = 1
    while ( eof_flag > 0){
    # ignore everything except for co-ordinate lines
        if (NF == 2){
            vert_arr[vert_count] = $0
            ++vert_count
        }
        eof_flag = getline < rfilename
    }
    close(rfilename)
    printf("/* Read in road file */\n")
    if (debug == 1) {
        for (i = 1; i < vert_count; i++) {
```

```
            print vert_arr[i]
        }
    }
    no_towns = 0
    FS = ","
}
{
    # for each town in the database
    if (NF == 12) {
        ++no_towns
        if (no_towns >= start_town) {
            if (debug == 1)
                printf("Handling town %s\n",$7)
            town_name = $7
            town_x = $11
            town_y = $12
            # set closest vertex as first vertex (just as a start value)
            split(vert_arr[1],vert_line," ")
            vert_x = vert_line["1"]
            vert_y = vert_line["2"]
            close_vert_x = vert_x
            close_vert_y = vert_y
            check_dist = (town_x - vert_x) ^ 2 + (town_y - vert_y) ^ 2
            close_dist = sqrt ( check_dist )
            # now run through vertices checking if any is closer
            for (j = 2; j < vert_count; j++) {
                split(vert_arr[j],vert_line," ")
                vert_x = vert_line["1"]
                vert_y = vert_line["2"]
                tmp_dist = (town_x - vert_x) ^ 2 + (town_y - vert_y) ^ 2
                if (tmp_dist < check_dist) {
                    close_vert_x = vert_x
                    close_vert_y = vert_y
                    close_dist = sqrt(tmp_dist)
                    check_dist = tmp_dist
                }
            }
            # now print closest distance
            printf("Town,%s,X,%d,Y,%d,Close_X,%d,Close_Y,%d,Dist,%d\n"\
            ,town_name,town_x,town_y,close_vert_x,close_vert_y,\
close_dist) > "clout.txt"
        }
    }
}
END {
}
```

B.3 Town and Province Information

The following script produces facts containing information about towns and the provinces that they are part of.

```
# awk program to create town in province datalog facts from GIS database
# (file towns_p2.txt)
#
# Usage: nawk -f provfilt.awk [prov and town file]
#
# input file is series of records with town info and province name
#
# sample output format:
#
# in_prov(town("Bethlehem",123,231),province("Orange Free State")).
BEGIN {
    printf("\n/* Town in Province info (provfilt.awk) */\n\n")
    FS = ","
}
{
    # only process the towns which are in provinces and leave out
    # blank lines
    if (NF > 0 && $10 != ""){
        printf("in_prov(town(\"%s\",%d,%d),province(\"%s\")).\n"\
        ,$7,$11,$12,$10)
    }
}
```

Appendix C

Datalog Programs

The Datalog programs used to generate the performance results in Chapter 5 are presented below. These programs are in the same sequence as the queries presented in the tables of Section 5.5, and include both the M-NFA translation and the RE-translation for each query. The code used for timing the query evaluation is only included at the end of the first program. Similar code is used for subsequent programs.

```
module two_roads.
export two_roads(bfff).
@no_rewriting.

t_0_0(road_vert(-233039,-2918998)).
t_0_1_U(Y,U) :- t_0_0(T), road_arc(T,Y,_,id(U,_)).
t_0_1_U(Y,U) :- t_0_1_U(T,U), road_arc(T,Y,_,id(U,_)).
tie_0(X2,U) :- t_0_1_U(X2,U).
proj_0_X2(X2) :- tie_0(X2,U).

t_1_0(X2,X2) :- proj_0_X2(X2).
t_1_1_V(X,Y,V) :- t_1_0(X,T), road_arc(T,Y,_,id(V,_)).
t_1_1_V(X,Y,V) :- t_1_1_V(X,T,V), road_arc(T,Y,_,id(V,_)).
tie_1(X2,X3,V) :- t_1_1_V(X2,X3,V).

two_roads(road_vert(-233039,-2918998),X3,U,V) :- tie_0(X2,U),tie_1(X2,X3,V).

end_module.

/* Timer code */

module timer.
export timer().
query_succeeds :- two_roads(road_vert(-233039,-2918998),X3,U,V).
timer :-
            printf("\nQuery Name : two_roads.P\n"),
```

```
            printf("\nQuery Goal : ?two_roads(X2)\n"),
            reset_timer(),
            query_succeeds,
            display_timer(),
            printf("\n\nEnd of Query\n\n").
end_module.

/*-------------------------------------------------------------------------*/

module two_roads.
export two_roads(bfff).
@no_rewriting.

two_roads(road_vert(-233039,-2918998),X3,U,V) :-
      tc_road_arc(X2,X3,V),
      tc_road_arc(road_vert(-233039,-2918998),X2,U).

tc_road_arc(Xtemp1,Ytemp1,V) :-
      road_arc(Xtemp1,Ytemp1,_, id(V,_)).
tc_road_arc(Xtemp1,Ytemp1,V) :-
      road_arc(Xtemp1,Ztemp1,_, id(V,_)),
      tc_road_arc(Ztemp1,Ytemp1,V).

tc_road_arc(Xtemp1,Ytemp1,U) :-
      road_arc(Xtemp1,Ytemp1,_, id(U,_)).
tc_road_arc(Xtemp1,Ytemp1,U) :-
      road_arc(Xtemp1,Ztemp1,_, id(U,_)),
      tc_road_arc(Ztemp1, Ytemp1, U).

end_module.

/*=========================================================================*/

module reach_towns.
export reach_towns(bff).
@no_rewriting.

t_0_0(town("VREDENDAL",-426041,-3457734)).
t_0_1(Y) :- t_0_0(T),near_town(Y,T,_).
t_0_2_U(Y,U) :- t_0_1(T), road_arc(T,Y,_,id(U,_)).
t_0_2_U(Y,U) :- t_0_1_U(T,U), road_arc(T,Y,_,id(U,_)).
t_0_2_U(Y,U) :- t_0_2_U(T,U), road_arc(T,Y,_,id(U,_)).
t_0_3_U(Y,U) :- t_0_2_U(T,U), near_town(T,Y,_).
t_0_1_U(Y,U) :- t_0_3_U(T,U), near_town(Y,T,_).

reach_towns(town("VREDENDAL",-426041,-3457734),X2,U) :- t_0_3_U(X2,U).

end_module.

/*-------------------------------------------------------------------------*/
```

```
module reach_towns.
export reach_towns(bff).
@no_rewriting.

reach_towns(town("VREDENDAL",-426041,-3457734),Y,U) :-
      tc_compo1(town("VREDENDAL",-426041,-3457734),Y,U).

tc_compo1(Xtemp1,Ytemp1,U) :-
      compo1(Xtemp1,Ytemp1, U).
tc_compo1(Xtemp1,Ytemp1,U) :-
      compo1(Xtemp1,Ztemp1,U),
      tc_compo1(Ztemp1,Ytemp1,U).

compo1(Xtemp1,Ytemp1,U) :-
      near_town(Temp1,Xtemp1,_),
      compo2(Temp1,Ytemp1,U).

compo2(Temp1,Ytemp1,U) :-
      tc_road_arc(Temp1,Temp2,U),
      near_town(Temp2,Ytemp1,_).

tc_road_arc(Xtemp1,Ytemp1,U) :- road_arc(Xtemp1,Ytemp1,_,id(U,_)).
tc_road_arc(Xtemp1,Ytemp1,U) :-
      road_arc(Xtemp1,Ztemp1,_,id(U,_)),
      tc_road_arc(Ztemp1,Ytemp1,U).

end_module.

/*=============================================================================*/

module many_roads.
export many_roads(bf).
@no_rewriting.

t_0_0(town("VREDENDAL",-426041,-3457734)).
t_0_1(Y) :- t_0_0(T), near_town(Y,T,_).
t_0_2(Y) :- t_0_1(T), road_arc(T,Y,_,id(_,_)).
t_0_2(Y) :- t_0_2(T), road_arc(T,Y,_,id(_,_)).
t_0_3(Y) :- t_0_2(T), near_town(T,Y,_).
t_0_1(Y) :- t_0_3(T), near_town(Y,T,_).

many_roads(town("VREDENDAL",-426041,-3457734),X2) :- t_0_3(X2).

end_module.

/*-----------------------------------------------------------------------------*/

module many_roads.
export many_roads(bf).
@no_rewriting.

many_roads(town("VREDENDAL",-426041,-3457734),Y) :-
```

```
      tc_compo1(town("VREDENDAL",-426041,-3457734),Y).

tc_compo1(Xtemp1,Ytemp1) :-
      compo1(Xtemp1,Ytemp1).
tc_compo1(Xtemp1,Ytemp1) :- compo1(Xtemp1,Ztemp1),
      tc_compo1(Ztemp1,Ytemp1).

compo1(Xtemp1,Ytemp1) :- near_town(Temp1,Xtemp1,_), compo2(Temp1,Ytemp1).

compo2(Temp1,Ytemp1) :- tc_road_arc(Temp1,Temp2), near_town(Temp2,Ytemp1,_).

tc_road_arc(Xtemp1,Ytemp1) :-
      road_arc(Xtemp1,Ytemp1,_,id(_,_)).
tc_road_arc(Xtemp1,Ytemp1) :-
      road_arc(Xtemp1,Ztemp1,_,id(_,_)), tc_road_arc(Ztemp1,Ytemp1).

end_module.

/*=============================================================================*/

module roads_prov.
export roads_prov(ff).
@no_rewriting.

t_0_0(town("VREDENDAL",-426041,-3457734)).
t_0_1(Y) :- t_0_0(T), near_town(Y,T,_).
t_0_2(Y) :- t_0_1(T), road_arc(T,Y,_,id(_,_)).
t_0_2(Y) :- t_0_2(T), road_arc(T,Y,_,id(_,_)).
t_0_3(Y) :- t_0_2(T), near_town(T,Y,_).
t_0_1(Y) :- t_0_3(T), near_town(Y,T,_).

tie_0(X2) :- t_0_3_(X2).
proj_0_X2(X2) :- tie_0(X2).

t_1_0(X2,X2) :- proj_0_X2(X2).
t_1_1(X,Y) :- t_1_0(X,T), in_prov(T,Y).
tie_1(X2,X3) :- t_1_1(X2,X3).

roads_prov(X2,X3) :- tie_0(X2), tie_1(X2,X3).

end_module.

/*-----------------------------------------------------------------------------*/

module roads_prov.
export roads_prov(ff).
@no_rewriting.

roads_prov(X2, X3) :-
      tc_compo1(town("Veredendal",-426041,-3457734), X2),
      in_prov(X2, X3).
```

```
tc_compo1(Xtemp1, Ytemp1) :-
      compo1(Xtemp1, Ytemp1).
tc_compo1(Xtemp1, Ytemp1) :-
      compo1(Xtemp1, Ztemp1),
      tc_compo1(Ztemp1, Ytemp1).

compo1(Xtemp1, Ytemp1) :-
      near_town(Temp1, Xtemp1, _),
      compo2(Temp1, Ytemp1).

compo2(Temp1, Ytemp1) :-
      tc_road_arc(Temp1, Temp1),
      near_town(Temp1, Ytemp1, _).

tc_road_arc(Xtemp1, Ytemp1) :-
      road_arc(Xtemp1, Ytemp1, _, id(_, _)).
tc_road_arc(Xtemp1, Ytemp1) :-
      road_arc(Xtemp1, Ztemp1, _, id(_, _)),
      tc_road_arc(Ztemp1, Ytemp1).

end_module.

/*=========================================================================*/

module common_anc.
export common_anc(bf).
@no_rewriting.

t_0_0(function(destroyer)).
t_0_1(Y) :- t_0_0(T), calls(Y,T).
t_0_1(Y) :- t_0_1(T), calls(Y,T).
tie_0(X1) :- t_0_1(X1).
proj_0_X1(X1) :- tie_0(X1).

t_1_0(X1,X1) :- proj_0_X1(X1).
t_1_2(X,Y) :- t_1_0(X,T), calls(T,Y).
t_1_2(X,Y) :- t_1_2(X,T), calls(T,Y).
tie_1(X1,X2) :- t_1_2(X1,X2).

common_anc(function(destroyer),X2) :- tie_0(X1), tie_1(X1,X2).

end_module.

/*-------------------------------------------------------------------------*/

module common_anc.
export common_anc(bf).
@no_rewriting.

common_anc(function(destroyer), function(F)) :-
      tc_calls(C, function(destroyer)),
      tc_calls(C, function(F)).
```

```
tc_calls(Xtemp1, Ytemp1) :-
      calls(Xtemp1, Ytemp1).
tc_calls(Xtemp1, Ytemp1) :-
      calls(Xtemp1, Ztemp1),
      tc_calls(Ztemp1, Ytemp1).

end_module.

/*=============================================================================*/

module calls_sched.
export calls_sched(f).
@no_rewriting.

t_0_0(function(schedule)).
t_0_1(Y) :- t_0_0(T), calls(Y,T).
t_0_1(Y) :- t_0_1(T), calls(Y,T).
tie_0(X1) :- t_0_1(X1).
proj_0_X1(X1) :- tie_0(X1).

t_1_0(X1,X1) :- proj_0_X1(X1).
t_1_2(X,Y) :- t_1_0(X,T), calls(T,Y).
t_1_2(X,Y) :- t_1_2(X,T), calls(T,Y).
tie_1(X1,X2) :- t_1_2(X1,X2).

calls_sched(X2) :- tie_0(X1), tie_1(X1,X2).

end_module.

/*-----------------------------------------------------------------------------*/

module calls_sched.
export calls_sched(bf).
@no_rewriting.

calls_sched(function(schedule), function(F)) :-
      tc_calls(C, function(schedule)),
      tc_calls(C, function(F)).

tc_calls(Xtemp1, Ytemp1) :-
      calls(Xtemp1, Ytemp1).
tc_calls(Xtemp1, Ytemp1) :-
      calls(Xtemp1, Ztemp1),
      tc_calls(Ztemp1, Ytemp1).


end_module.

/*=============================================================================*/
```

```
module depends.
export depends(ff).
@no_rewriting.

t_0_0(class(X),class(X)).
t_0_1(X,Y) :- t_0_0(X,T), contains(T,Y).
tie_0(class(X),U) :- t_0_1(class(X),U).
proj_0_U(U) :- tie_0(class(X),U).

t_1_0(U,U) :- proj_0_U(U).
t_1_1(X,Y) :- t_1_0(X,T), calls(T,Y).
t_1_1(X,Y) :- t_1_1(X,T), calls(T,Y).
tie_1(U,V) :- t_1_1(U,V).
proj_1_V(V) :- tie_1(U,V).

t_2_0(V,V) :- proj_1_V(V).
t_2_1(X,Y) :- t_2_0(X,T), contains(Y,T).
tie_2(V,class(Y)) :- t_2_1(V,class(Y)).

depends(class(X),class(Y)) :-
      tie_0(class(X),U),
      tie_1(U,V),tie_2(V,class(Y)).

end_module.

module cl_depends.
export cl_depends(bf).
@no_rewriting.

t_0_0(class("Object").
t_0_1(Y) :- t_0_0(T), depends(T,Y).
t_0_1(Y) :- t_0_1(T), depends(T,Y).
cl_depends(class("Object"), class(X2)) :- tie_0(class(X2)).

end_module.

/*-------------------------------------------------------------------------*/

module cl_depends.
export cl_depends(bf).
@no_rewriting.

depends(class(X), class(Y)) :-
   contains(class(Y), V),
   contains(class(X), U),
   tc_calls(U, V).

tc_calls(Xtemp1, Ytemp1) :-
      calls(Xtemp1, Ytemp1).
tc_calls(Xtemp1, Ytemp1) :-
      calls(Xtemp1, Ztemp1),
      tc_calls(Ztemp1, Ytemp1).
```

```
cl_depends(class("Object"), class(Y)) :-
      tc_depends(class("Object"), class(Y)).

tc_depends(Xtemp1, Ytemp1) :-
      depends(Xtemp1, Ytemp1).
tc_depends(Xtemp1, Ytemp1) :-
      depends(Xtemp1, Ztemp1),
      tc_depends(Ztemp1, Ytemp1).

end_module.

/*=============================================================================*/

module se_depends.
export se_depends(bf).
@no_rewriting.

t_0_0(class("Object")).
t_0_1(Y) :- t_0_0(T), contains(T,Y).
t_0_2(Y) :- t_0_1(T), calls(T,Y).
t_0_2(Y) :- t_0_2(T), calls(T,Y).
t_0_3(Y) :- t_0_2(T), contains(Y,T).
t_0_1(Y) :- t_0_3(T), contains(Y,T).
tie_0(class(Y)) :- t_0_3(class(Y)).

se_depends(class("Object"),class(Y)) :- tie_0(class(Y)).

end_module.

/*-----------------------------------------------------------------------------*/

module se_depends.
export se_depends(bf).
@no_rewriting.

se_depends(class("Object"),class(Y)) :-
      tc_compo1(class("Object"),class(Y)).

tc_compo1(Xtemp1,Ytemp1) :-
      compo1(Xtemp1,Ytemp1).
tc_compo1(Xtemp1,Ytemp1) :- compo1(Xtemp1,Ztemp1),
      tc_compo1(Ztemp1,Ytemp1).

compo1(Xtemp1,Ytemp1) :- contains(Xtemp1,Temp1), compo2(Temp1,Ytemp1).

compo2(Temp1,Ytemp1) :- tc_calls(Temp1,Temp2), contains(Ytemp1,Temp2).

tc_calls(Xtemp1,Ytemp1) :-
      calls(Xtemp1,Ytemp1).
tc_calls(Xtemp1,Ytemp1) :-
      calls(Xtemp1,Ztemp1), tc_calls(Ztemp1,Ytemp1).

end_module.
```

```
/*=================================================================*/

module overwrites_f.
export overwrites_f(ff).
@no_rewriting.

t_0_0(A,A).
t_0_1(X,Y) :- t_0_0(X,T), area_section(T,Y).
tie_0(A,SP1) :- t_0_1(A,SP1).
proj_0_SP1(SP1) :- tie_0(A,SP1).
proj_0_A(A) :- tie_0(A,SP1).

t_1_0(SP1,SP1) :- proj_0_SP1(SP1).
t_1_1(X,Y) :- t_1_0(X,T), section_area(T,Y).
t_1_2(X,Y) :- t_1_1(X,T), area_section(T,Y).
t_1_1(X,Y) :- t_1_2(X,T), section_area(T,Y).

tie_1(SP1,S1) :- t_1_0(SP1,S1).
tie_1(SP1,S1) :- t_1_2(SP1,S1).

t_2_0(A,A) :- proj_0_A(A).
t_2_1(X,Y) :- t_2_0(X,T), area_section(T,Y).
tie_2(A,SP2) :- t_2_1(A,SP2).
proj_2_SP2(SP2) :- tie_2(A,SP2).

t_3_0(SP2,SP2) :- proj_2_SP2(SP2).
t_3_1(X,Y) :- t_3_0(X,T), section_area(T,Y).
t_3_2(X,Y) :- t_3_1(X,T), area_section(T,Y).
t_3_1(X,Y) :- t_3_2(X,T), section_area(T,Y).

tie_3(SP2,S2) :- t_3_0(SP2,S2).
tie_3(SP2,S2) :- t_3_2(SP2,S2).

overwrites_f(S1, S2) :-
   tie_0(A,SP1),
   tie_1(SP1,S1),
   tie_2(A,SP2),
   tie_3(SP2,S2),
   not SP1 = SP2.

end_module.
```

```
/*-----------------------------------------------------------------*/

module overwrites_f.
export overwrites_f(ff).
@no_rewriting.

overwrites_f(S1,S2) :-
    aux(A,SP1,S1),
    aux(A,SP2,S2),
    not SP1 = SP2.
```

```
aux(A,SP1,S1) :- area_section(A,SP1), kl_compo(SP1,S1).
kl_compo(SP1,S1) :-
      SP1 = S1.
kl_compo(SP1,S1) :-
      compo(SP1,X),
      kl_compo(X,S1).

compo(X,Y) :- section_area(X,Z), area_section(Z,Y).

end_module.

/*=========================================================================*/

module overwrites_f.
export overwrites_f(ff).
@no_rewriting.

t_0_0(deskArea).
t_0_1(Y) :- t_0_0(T), area_section(T,Y).
tie_0(SP1) :- t_0_1(SP1).
proj_0_SP1(SP1) :- tie_0(SP1).

t_1_0(SP1,SP1) :- proj_0_SP1(SP1).
t_1_1(X,Y) :- t_1_0(X,T), section_area(T,Y).
t_1_2(X,Y) :- t_1_1(X,T), area_section(T,Y).
t_1_1(X,Y) :- t_1_2(X,T), section_area(T,Y).
tie_1(SP1,S1) :- t_1_0(SP1,S1).
tie_1(SP1,S1) :- t_1_2(SP1,S1).

t_2_0(deskArea).
t_2_1(Y) :- t_2_0(T), area_section(T,Y).
tie_2(SP2) :- t_2_1(SP2).
proj_2_SP2(SP2) :- tie_0(SP2).

t_3_0(SP2,SP2) :- proj_2_SP2(SP2).
t_3_1(X,Y) :- t_3_0(X,T), section_area(T,Y).
t_3_2(X,Y) :- t_3_1(X,T), area_section(T,Y).
t_3_1(X,Y) :- t_3_2(X,T), section_area(T,Y).

tie_3(SP2,S2) :- t_3_0(SP2,S2).
tie_3(SP2,S2) :- t_3_2(SP2,S2).

overwrites_f(S1, S2) :-
   tie_0(SP1),
   tie_1(SP1,S1),
   tie_2(SP2),
   tie_3(SP2,S2),
   not SP1 = SP2.

end_module.

/*-------------------------------------------------------------------------*/
```

```
module overwrites_b.
export overwrites_b(ff).
@no_rewriting.

overwrites_b(S1,S2) :-
    aux(SP1,S1),
    aux(SP2,S2),
    not SP1 = SP2.

aux(SP1,S1) :- area_section(deskArea,SP1), kl_compo(SP1,S1).
kl_compo(SP1,S1) :-
      SP1 = S1.
kl_compo(SP1,S1) :- compo(SP1,Ztemp1),
      kl_compo(Ztemp1, S1).

compo(X,Y) :- section_area(X,Ztemp1), area_section(Ztemp1,Y).

end_module.

/*=========================================================================*/

module smashable_f.
export smashable_f(ff).
@no_rewriting.

t_0_0(S2,S2).
t_0_1(X,Y) :- t_0_0(X,T), section_function(T,Y).
tie_0(S2,F3) :- t_0_1(S2,F3).
proj_0_S2(S2) :- tie_0(S2,F3).
proj_0_F3(F3) :- tie_0(S2,F3).


t_1_0(F3,F3) :- proj_0_F3(F3).
t_1_1(X,Y) :- t_1_0(X,T), calls(T,Y).
tie_1(F3,F1) :- t_1_1(F3,F1).
proj_1_F1(F1) :- tie_1(F3,F1).

t_2_0(F1,F1) :- proj_1_F1(F1).
t_2_1(X,Y) :- t_2_0(X,T), section_function(Y,T).
tie_2(F1,S1) :- t_2_1(F1,S1).
proj_2_S1(S1) :- tie_2(F1,S1).

t_3_0(S2,S2).
t_3_1(X,Y) :- t_3_0(X,T), section_area(T,Y).
t_3_0(X,Y) :- t_3_1(X,T), area_section(T,Y).
tie_3(S2,S1) :- t_3_0(S2,S1).

smashable_f(S2,F1) :-
   tie_0(S2,F3),
   tie_1(F3,F1),
   tie_2(F1,S1),
   not tie_3(S2,S1).
```

```
end_module.

/*-------------------------------------------------------------------------*/

module smashable_f.
export smashable_f(ff).
@no_rewriting.

smashable_f(S1, F2) :-
   section_function(S1, F1),
   calls(F1, F2),
   section_function(S2, F2),
   not kl_compo1(S1, S2).

kl_compo1(Xtemp1, Ytemp1) :-
      Xtemp1 = Ytemp1.
kl_compo1(Xtemp1, Ytemp1) :-
      compo1(Xtemp1, Ztemp1),
      kl_compo1(Ztemp1, Ytemp1).

compo1(Xtemp1, Ytemp1) :-
   section_area(Xtemp1, Temp1),
   area_section(Temp1, Ytemp1).

end_module.

/*=========================================================================*/

module smashable_b.
export smashable_b(bf).
@no_rewriting.

t_0_0(initSect).
t_0_1(Y) :- t_0_0(T), section_function(T,Y).
tie_0(F3) :- t_0_1(F3).
proj_0_F3(F3) :- tie_0(F3).

t_1_0(F3,F3) :- proj_0_F3(F3).
t_1_1(X,Y) :- t_1_0(X,T), calls(T,Y).
tie_1(F3,F1) :- t_1_1(F3,F1).
proj_1_F1(F1) :- tie_1(F3,F1).

t_2_0(F1,F1) :- proj_1_F1(F1).
t_2_1(X,Y) :- t_2_0(X,T), section_function(Y,T).
tie_2(F1,S1) :- t_2_1(F1,S1).

t_3_0(initSect).
t_3_1(Y) :- t_3_0(T), section_area(T,Y).
t_3_0(Y) :- t_3_1(T), area_section(T,Y).
tie_3(S1) :- t_3_0(S1).

smashable_b(initSect,F1) :-
```

```
   tie_0(F3),
   tie_1(F3,F1),
   tie_2(F1,S1),
   not tie_3(S1).

end_module.

/*-------------------------------------------------------------------------*/

module smashable_b.
export smashable_b(bf).

smashable_b(initSect, F2) :-
      section_function(initSect, F1),
      calls(F1, F2),
      section_function(S2, F2),
      not kl_compo1(initSect, S2).

kl_compo1(Xtemp1, Ytemp1) :-
      compo1(Xtemp1, Ztemp1),
      kl_compo1(Ztemp1, Ytemp1).
kl_compo1(Xtemp1, Ytemp1) :-
      Xtemp1 = Ytemp1.

compo1(Xtemp1, Ytemp1) :-
      section_area(Xtemp1, Temp1),
      area_section(Temp1, Ytemp1).

end_module.

/*=========================================================================*/
```

Appendix D

Complete Table of Results

Query	RE	RE	MNFA	MNFA	MNFA	MNFA	NFA
	no rw	context/ magic rw	factor constr no rw	factor no constr no rw	factor constr rw	factor no constr rw	factor single edge
two_roads	?	45.9 (m)	11.6	?	22.9	11.9	(10.9)
reach_towns	?	16.5 (m+sr)	11.2	n/a	n/a	11.2	11.2
many_roads	?	? (m+sr)	20.2	n/a	n/a	n/a	20.2
roads_prov	?	? (c+sr)	26.1	59.2	?	26.6	n/a
common_anc	?	105 (m) 63.8 (c+sr)	79.8	104	622	82.3	(2.92)
calls_sched	?	90.1 (m) 10 .1 (c+sr)	0.85	85.7	2.89	2.29	(0.52)
cl_depends	?	?	?	?	?	83.4	(3.57)
se_depends	?	63.9 (m+sr) ? (c+sr)	3.57	n/a	n/a	n/a	3.57
overwrites_f	25.3	22.4 (m) 22.7 (c)	21.3	22.4	?	25.6	(19.8)
overwrites_b	8.24	3.56 (m) 2.83 (c)	2.47	6.38	3.41	2.92	(2.24)
smashable_f	14.6	89.5 (m)	32.5	64.7	n/a	n/a	(11.1)
smashable_b	1.68	2.32	2.12	39.4	n/a	n/a	(0.73)

Bibliography

[AKW88] A.V. Aho, B.W. Kernighan, and P.J. Weinberger. *The AWK Programming Language.* Addison-Wesley Publishing Company, 1988.

[ASU86] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley Publishing Company, 1986.

[AU92] A.V. Aho and J.D. Ullman. *Foundations of Computer Science.* Computer Science Press, 1992.

[Baa88] S. Baase. *Computer Algorithms.* Addison-Wesley Publishing Company, 1988.

[Ber73] C. Berge. *Graphs and Hypergraphs.* North-Holland Publishing Company, 1973.

[BMSU86] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic Sets and other strange ways to implement logic programs. In *Proceedings of the Fifth ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 1–15, 1986.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's intoduction to recursive query processing strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 16–52, 1986.

[BR87] C. Beeri and R. Ramakrishnan. On the power of Magic. In *Proceedings of the ACM Symposium on Principles of Database Systems*, pages 269–283, March 1987.

[CDRS86] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the International Conference on Very Large Data Bases*, August 1986.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, 1990.

[CH93] M. P. Consens and M. Hasan. Supporting network management through declaratively specified data visualizations. In *Proceedings of the Third IFIP/IEEE International Symposium on Integrated Network Management*, pages 725–738, April 1993.

[CHM93] M. P. Consens, M. Hasan, and A. O. Mendelzon. Debugging distributed programs by visualizing and querying even traces. In *Proceedings of the Third ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 181–183, May 1993. Extended abstract.

[CKW89] W. Chen, M. Kifer, and D.S. Warren. HiLog as a platform for database language (or why predicate calculus is not enough). In *Proceedings of the Second International Workshop on Database Programming Languages*, pages 315–329, June 1989.

[CM92] M. P. Consens and A. O. Mendelzon. Low complexity aggregation in GraphLog and Datalog. In *Proceedings of the Third International Conference on Database Theory*, 1992.

[CM93] M. P. Consens and A. O. Mendelzon. Hy$^+$: A Hygraph-based query and visualization system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 51–66, 1993.

[CMV94] M. P. Consens, A. O. Mendelzon, and D. Vista. Deductive database support for data visualization. In *Proc. 4th Int. Conf. on Extending Database Technology*, pages 45–58, 1994.

[CMVW95] M. P. Consens, A. O. Mendelzon, D. Vista, and P.T. Wood. Join ordering and constant propagation in datalog programs. In *To appear in the Second International Workshop on Rules in Database Systems*, 1995.

[CMW88] I.F. Cruz, A.O. Mendelzon, and P.T. Wood. G$^+$: Recursive queries without recursion. In Larry Kerschberg, editor, *Proceedings of the Second International Conference on Expert Database Systems*, pages 355–368, 1988.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Con89] M. P. Consens. Real life recursive queries using graphs. Master's thesis, Department of Computer Science, University of Toronto, January 1989.

[Con94] M. P. Consens. *Creating and Filtering Structural Data Visualizations using Hygraph Patterns.* PhD thesis, Department of Computer Science, University of Toronto, February 1994.

[Cru87] I. Cruz. G^+: Recursive queries without recursion. Master's thesis, Department of Computer Science, University of Toronto, 1987.

[Eig93] F. Ch. Eigler. GXF: A graph exchange format. In A. O. Mendelzon, editor, *Declarative Database Visualization : Recent Papers from the Hy$^+$/GraphLog Project*, pages 92–107. Computer Systems Research Institute, University of Toronto, 1993.

[Fuk91] M. Fukar. Translating Graphlog into Prolog. Master's thesis, Department of Computer Science, University of Toronto, July 1991.

[GV89] G. Gardarin and P. Valduriez. *Relational Databases and Knowledge Bases.* Addison-Wesley, 1989.

[Har88] D. Harel. On visual formalisms. *Communications of the ACM*, 31(5):514–530, 1988.

[HU79] J. Hopcroft and J. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.

[KRS90] D. Kemp, R. Ramamohanarao, and Z. Somogyi. Right-, left-, and multi-linear rule transformations that maintain context information. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 380–391, 1990.

[Lev80] L. S. Levy. *Discrete structures of Computer Science.* John Wiley & Sons, 1980.

[Llo87] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1987.

[MP91] I.S. Mumick and H. Pirahesh. Overbound and right-linear queries. In *Proceedings of the Tenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, pages 127–141, 1991.

[Noi93] E. G. Noik. Graphite: A suite of hygraph visualization utilities. In A. O. Mendelzon, editor, *Declarative Database Visualization : Recent Papers from the Hy$^+$/GraphLog Project*, pages 108–126. Computer Systems Research Institute, University of Toronto, 1993.

[NRSU89a] J.F. Naughton, R.Ramakrishnan, Y. Sagiv, and J.D. Ullman. Argument reduction by Factoring. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 173–182, 1989.

[NRSU89b] J.F. Naughton, R.Ramakrishnan, Y. Sagiv, and J.D. Ullman. Efficient evaluation of right-, left-, and combined-linear rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 235–242, 1989.

[NT89] S. Naqvi and S. Tsur. A logical language for data and knowledge bases. In *Principles of Computer Science*. Computer Science Press, New York, 1989.

[Ram88] R. Ramakrishnan. Magic templates: a spellbinding approach to logic programs. In *Proceedings of the Fifth International Symposium on Logic Programming*, pages 140–159, 1988.

[RSS92] R. Ramakrishnan, D. Srivastava, and S. Sudarshan. CORAL: Control, Relations and Logic. In *Proceedings of the Eighteenth International Conference on Very Large Data Bases*, 1992.

[RSSS93] R. Ramakrishnan, D. Srivastava, S. Sudarshan, and P. Seshadri. Implementation of the CORAL deductive database system. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1993.

[RSSS94] R. Ramakrishnan, P. Seshadri, D. Srivastava, and S. Sudarshan. *The CORAL user manual.* Computer Sciences Department, University of Wisconsin-Madison, WI53706, U.S.A., 1994.

[Ull85] J.D. Ullman. Implementation of logical query languages for databases. *ACM Transactions on Database Systems*, 10(3):289–321, 1985.

[Ull88] J.D. Ullman. *Principles of database and knowledge-base systems*, volume 1. Computer Science Press, 1988.

[VRK+90] J. Vaghani, K. Ramamohanarao, D. Kemp, Z. Somogyi, and P. Stuckey. The Aditi deductive database system. In *Proceedings of the NACLP'90 Workshop on Deductive Database Systems*, 1990.

[VW95] D. Vista and P.T. Wood. Efficient evaluation of visual queries using deductive databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 143–161. Kluwer Academic Publishers, Boston, 1995.

[Woo88] P. T. Wood. *Queries on graphs.* PhD thesis, Department of Computer Science, University of Toronto, December 1988.

[Woo90] P. T. Wood. Factoring augmented regular chain programs. In *Proceedings of the Sixteenth International Conference on Very Large Data Bases*, pages 255–263, 1990.