RULE SUPPORTING IN PROLOG

by

O. Shmueli, S. Tsur, and H. Zfira

Technical Report #337

September 1984

1 Computer Science Dept. Technion-IIT, Haifa, Israel
2 IBM Scientific Center, Technion City, Haifa, Israel
3 Computer Science Dept. Technion-IIT, Haifa, Israel.
RULE SUPPORT IN PROLOG

Oded Shmueli¹
Shalom Tsur²
Hana Zfira³

ABSTRACT

In this paper we propose the idea of rule support - a method which improves the performance of goal-proof processes in PROLOG by augmenting the rules in the rule-base with additional support knowledge. This method is especially appropriate when a PROLOG program contains a large internal database.

We discuss the idea in general and show two variants of its implementation: a priori and dynamic support. For a priori support, the supporting knowledge is collected for all feasible cases prior to using the rule. For dynamic support, the supporting knowledge is accumulated incrementally as the rule is used.

¹ Computer Science Department, Technion - Israel Institute of Technology, Haifa, Israel.
² IBM Scientific Center, Technion, Haifa, Israel.
³ Computer Science Department, Technion - Israel Institute of Technology, Haifa, Israel.
CONTENTS

1. Introduction .................................................................................................................. 1
2. Basic Concepts .............................................................................................................. 4
  2.1. Classification of PROLOG Rules ............................................................................. 4
  2.2. Graph Representation of Rules .............................................................................. 5
  2.3. Support Structure ................................................................................................. 7
3. Constructing a Support Structure ................................................................................ 8
  3.1. Transformation into a Serviceable Procedure ...................................................... 8
  3.2. Creation of a Support Structure for Serviceable Rules ....................................... 9
    3.2.1. Creation of a Rule Schema .......................................................................... 9
    3.2.2. Qual Tree ................................................................................................. 11
4. A Priori Rule Support ................................................................................................... 13
  4.1. Populating the Support Structure ....................................................................... 13
  4.2. Changes in the Underlying Rules ....................................................................... 14
  4.3. A Service Program ......................................................................................... 15
5. Dynamic Rule Support ................................................................................................ 16
  5.1. The Basic Mechanism ..................................................................................... 16
  5.2. The Spectrum of Support Mechanisms .............................................................. 18
  5.3. Handling Templates ....................................................................................... 18
6. Conclusion ................................................................................................................... 20
References ...................................................................................................................... 21
1. INTRODUCTION

One of the most promising developments of logic programming appears to be the application of the method to problems in the database area. The main attraction of the method is in its ability to address the problems of data organization and the operations on data in terms of first order logic. In database terms this means that the problems of data modeling and query languages are treated in a uniform way in which no formal distinction between data objects and the operations on these objects is made. An additional advantage of this method is that the formalism enables the specification of powerful queries e.g., recursive queries which are often beyond the limits of expression of traditional query languages.

Whereas it is clear that logic programming is a potent tool of expression in the database area, other aspects, notably those that are performance related, must be examined. Many traditional database problems such as query optimization (see e.g., [Ull]) have their counterparts in the context of logic programming. These problems have received some attention [War, KY, Ull1, JCV, Kel] but more needs to be done.

In this paper we address one of these problems. We have chosen to call it "rule support" and, in concept, it is akin to the view maintenance problem of database systems. We describe the problem in terms of PROLOG [CM1] and assume some familiarity on behalf of the reader with this language. Although we have chosen a specific language to discuss the problem, it is by no means restricted by this particular context and should be regarded as a general problem of logic programming and databases.

Many applications e.g., expert systems, maintain a knowledge base - a large, almost static set of axioms and rules which is used by the inference mechanism to prove given hypotheses (or goals) which are submitted to it. Proofs are executed as a rule-directed, depth first search through a solution space and terminate when data can be found to satisfy all goals and their subgoals in the search tree. This process is sequential and often requires backtracking during the process in order to satisfy a subgoal with a new candidate for a solution.

During the proof process a significant amount of additional knowledge is created which is immediately discarded and hence, is nonexistent when the same proof is attempted at a future time. The idea of rule support is to save this knowledge in suitable structures so as to shorten the required time when the same proof is repeated. We propose thus to trade time for space in these proof processes.

Consider the following trivial example:

\[
girl(\text{mary}).
girl(\text{sue}).
boy(\text{john}).
boy(\text{jack}).
\]

\[
pair(\text{X}, \text{Y}) :- boy(\text{X}), girl(\text{Y}).
\]

To generate all possible pairs of girls and boys we specify the goal:

\[
pair(\text{X}, \text{Y}), \text{write}(\text{X,Y}), \text{nl}, \text{fail}.
\]

during which the fact base is scanned over and again to produce the set of possible results. Ordinarily, these results are lost and must be reproduced when the
query is specified again. Instead, we could augment the fact base and retain these results in the form of:

```
pair(john, mary).
pair(john, sue).
pair(jack, mary).
pair(jack, sue).
```

We say that in this form the rule 'pair(X, Y)' is supported. We call the 'boy' and 'girl' facts base facts and the retained 'pair' information derived facts. Intuitively, a rule is supportable if in all its appearances it is either immediately resolvable or all of its right hand terms are supported. Precise definitions will be given in the sequel.

So far it was implicitly assumed that the knowledge base is static i.e. the information contained in it never changes. In practice this is of course seldom true, although as mentioned, in many cases changes are infrequent. We must therefore consider the potential effect of changes in the knowledge base on the supported rules. In this respect we must address the following questions:

1. Which rules can be supported?
2. If a rule is supportable - how do we support it?
3. For which of the supportable rules does it pay off to lend the support?

Another aspect of the problem is the time of support. Rules can be supported a priori - they are evaluated before their actual use in a proof and only updated when changes occur. We will refer to this method as a priori support. Alternatively, support can be generated for rules during their actual use in proofs. The resulting support will be built incrementally and no distinction will be made between the buildup phase and the update phase. We call this method dynamic support. Some hybrid methods are also possible.

In this paper we present versions of the a priori support method and the dynamic support method. We assume for the time being that the user, who knows the most about the problem to be solved, will indicate to the system which of the rules he wants supported. Obviously, an operational analysis which accounts for such factors as the frequency of use of the various rules, the available memory space and other statistics would be most useful in this respect. Such an analysis, which would be within the realm of the third question that we have posed, is still an open problem.

In supporting PROLOG rules the concept of tree database [GS2] (also called acyclic database [BFMMUY]) is utilized. Acyclicity is a property of the database schema which has wide implications in query processing, dependency theory and schema design (see [SI] for references). (We assume familiarity with relational databases at the level of [ULI]). It has been shown [BC, BG, GS1, Yan] that certain queries which imply acyclic databases, called tree queries, appear easier to process than queries which imply cyclic databases (called cyclic queries); and that the crux of query processing is constructing a tree (actually an embedded tree) [GS3, GST].

In [SI] a mechanism was proposed for maintaining results of queries over time; it consists of maintaining an acyclic database (which is derived from the the original database and the query) together with information that may be useful for future maintenance following additions and deletions of tuples. If the derived database is cyclic, then it is made acyclic by adding relations. The added relations are called templates. During additions (or deletions) base and template relations undergo modifications to reflect changes to base relations. Changes
Towards the database root where relations are viewed as tree nodes.

In this paper the query support mechanism is modified, extended and adapted for supporting PROLOG rules. Relational databases are "built into" PROLOG, one can think about tuples as base facts, relations as a collection of base facts for the same rules and queries as PROLOG procedures. Hence, the query maintenance mechanism lends itself (with certain modifications) to the support of PROLOG rules. The paper is organized as follows. Section 2 defines basic concepts and relates these to the more traditional concepts of relational databases. Section 3 discusses the transformation of the original PROLOG program and the construction of the support structure. Sections 4 and 5 discuss two variants of the idea which we have called "a-priori support" and "dynamic support". We conclude this work in section 6 with some suggestions for further research.
2. BASIC CONCEPTS

2.1. Classification of PROLOG rules

A PROLOG procedure is a collection of PROLOG rules all having the same head. The right hand (r.h.) side of each rule consists of a conjunction of one or more terms. A base fact is any rule in the rule base without a right hand (or, by convention, having 'true' as its right hand term), and no uninstantiated variables. We consider facts containing uninstantiated variables as rules, for example the fact \( a(X) \) is treated as the rule \( a(X) :- X > -\infty \). By comparison, a base fact is a tuple in the relational sense. We will use the term base relation for a collection of base facts all having the same axiom name.

From now on we will distinguish between "rules" and "facts", and assume that there is no rule having the same name as a base relation. Also, rules are assumed to have no constant elements within the rule head. Observe that the rule \( a(X,3,1) :- X < 80 \) is equivalent to the rule \( a(X,Y,Z) :- X<80, Y=3, Z=1 \). As mentioned in the introduction, occasionally a rule is replaced by a collection of evaluated instances. We call these instances derived facts. In many cases we do not distinguish between base facts and derived facts and we simply call them facts.

**Definition - Supportable**

A rule is supportable if it is equivalent to a finite number of facts. A procedure is supportable if all its rules are supportable; a term is supportable if it corresponds to a supportable procedure.

**Definition - Immediate Terms**

Terms that are not unified (in the PROLOG sense) are called immediate terms, e.g. ! (cut), fail, \( X \Theta C \), \( X \Theta Y \), where \( X,Y \) are variables, \( C \) is a constant and \( \Theta \) is a comparison operator.

**Definition - Elementary Rule**

1. Any base fact is an elementary rule.
2. A rule is elementary if all its right hand terms are elementary.
3. No other rule is elementary.

The definition of elementary rules covers those cases in which no immediate terms are included in the rules. The more general case includes immediate terms which we now define.

**Definition - Semi Elementary Rule.**

1. Any elementary rule is semi elementary.
2. A rule is semi elementary if all its r.h. terms are semi elementary or they are of any of the following formats:
   - \( X = C \) where \( X \) is a variable and \( C \) is a constant.
   - \( X \Theta C \) where variable \( X \) appears in any of the semi elementary terms of the rule and \( C \) is a constant.
   - \( X \Theta Y \) where variables \( X \) and \( Y \) appear in semi elementary terms of the rule.

---

*We assume that prior to processing, all PROLOG rules have been normalized to conjunctions of right hand terms.*
(3) No other rule is semi elementary.

As will be shown in section 3, semi elementary rules are supportable. Note that the immediate term types included in the definition of a semi elementary rule are those that can be applied to the set of facts when the support structure is created. This is not always the case. Consider the following example:

\[
p(X, Y) \leftarrow X > B, q1(X, Z), Z > 9.
q1(X, Y) \leftarrow X > 10, q3(Y).
\]

Rule \( q1 \) is not semi elementary - variable \( X \) does not appear in any of the semi elementary terms of the rule. Therefore the constraint \( X > B \) of rule \( p \) does not correspond with any of the allowable types of a semi elementary rule and consequently, rule \( p \) is not semi elementary.

In certain cases the information contained in the top level of the support structure is sufficient to service the terms that can be unified with the rule but special provisions need to be taken when the support structure is used. The set of these constraints will be called loose constraints.

**Definition - Loose constraints**

An immediate term is a loose constraint in exactly the following cases:

1. It is an inequality which involves a variable not appearing in any other semi elementary term of the rule.
2. It is the PROLOG axiom \( 'fail' \).
3. It is the PROLOG axiom \( '!' \) (cut).

**Definition - Serviceable Rules and Procedures (Domain of Support).**

A rule is serviceable if each r.h. term is either semi elementary or loose. A procedure is serviceable if all its rules are serviceable. This defines the domain of support.

### 2.2. Graph Representation of Rules

In this subsection we will relate the above concepts with known relational database concepts. Consider an elementary PROLOG rule possessing a right hand side in the PROLOG subset of interest. The rule has the form:

\[
q(X_1, ..., X_n) \leftarrow \bigwedge_{i=1}^{m} p_i(Y_{1i}, ..., Y_{ni}).
\]

where \( (X_1, ..., X_n) \) is the set of target variables and the terms on the right hand side are called the qualification. The \( X_i 's \) are variables and the \( Y_{ki} 's \) may be \( X_i 's \), constants or variables.

We need to encode the way in which supportable terms interact in rules. With each variable in the rule we associate a unique new constant - the attribute for this variable. Each term is associated with a term schema which is composed of a unique constant corresponding to the term's functor which we call a relation name, and the list of attributes appearing in the terms. The collection of term schemas for a rule together with the term schema containing the attributes corresponding to the target variables (called the target schema), is called a rule schema.

Note that a term (rule) schema corresponds to a relation (database) schema in relational databases. In fact, we shall use them as relation schemas for relations (in the database sense) used in the support structure.
for a rule is an undirected graph whose nodes are in one-to-one correspondence with the term schemas of the qualification and the target schema, such that for each attribute $A$, the subgraph induced by the nodes whose corresponding terms contain $A$ is connected (this is the original qual graph definition for relational databases [BG]). A rule is a tree rule if some qual graph for it is a tree; otherwise it is a cyclic rule.

example:

Consider the rule:

$q(A,B,X) :- p(A,B,C,X), s(A,C), t(A,B,D), r(B,D).$

The rule schema is:

$(q, [a,b,x]), (p, [a,b,c,x]), (s, [a,c]), (t, [a,b,d]), (r, [b,d]),$ 

This is a tree rule as depicted by the following qual graph:

```
abx----abcx----ac
       |   |
       abd----bd
```

example:

Consider the rule:

$q(A,X) :- p(A,B), r(A,C), s(B,C,D), t(B,D).$

This is a cyclic rule, in fact the only qual graph for it is the following cyclic graph:

```
ax----ab----ac
       /   \
      bcd----bd
```

The following simple procedure, discovered independently by [Gra] and [YO], recognizes tree rules. The procedure applies the following two steps until neither is applicable. For tree rules it also constructs $T$, the edge set of a qual tree for the rule (T is initially empty).

Step 1: Delete any attribute which appears in exactly one term schema or only in the target schema.

Step 2: Find term schema $\tau_1$ such that either the attributes appearing in it are a subset of the target attributes, or there is another term schema $\tau_2$ such that all the attributes in $\tau_1$ also appear in $\tau_2$. Delete $\tau_1$ from the rule schema and add the edge $(\tau_1, \tau_2)$ to $T$.

It can be shown that the original rule is a tree rule iff upon termination of the above procedure the rule has an empty right hand side. If the input rule is in fact

---

6We use traditional graph theory notation.

6Actually $T$ defines a directed graph, $T$ is the edge set of a qual tree if the edges are made undirected.
a tree rule then T is a qual tree for the rule with the target at its root.

2.3. Support Structure

The support structure for a rule contains the following information:

1. A set of facts which replaces the rule and is used in the resolution of any term that can be unified with the head of the rule.
2. An update structure which is used to modify the set of facts when a change occurs in any of the underlying rules.
3. A service program which is used to access the serviced rule.

The update structure is hierarchical, induced by the qual trees defined previously. Changes in any of the underlying rules may affect the validity of their associated fact-sets. These changes may in turn propagate upwards and affect the facts of rules at higher levels. Whereas updates may propagate upwards in the support structure, we require that term resolution is performed from the top level of the structure i.e., we will not have to descend to its lower levels for this purpose. It may be necessary to transform the original PROLOG program in order to satisfy this requirement.
3. CONSTRUCTING A SUPPORT STRUCTURE

3.1. Transformation into a Serviceable Procedure

Ordinarily, all semi elementary (s.e.) rules are serviceable. In certain cases a procedure 'p' which is not serviceable due to a non serviceable rule may be transformed into an equivalent serviceable procedure. This transformation may affect other procedures which take part, directly or indirectly, in the definition of the procedure we wish to service. The transformation succeeds precisely when the definition of 'p' is not recursive; i.e. there is no sequence of procedures p=p_1, ... , p_n=p_{n+1}, n>1, such that for i=1...n-1 procedures p_i has a rule with a r.h. term referencing procedure p_{i+1}.

The transformation uses the following operations:

1. partition and duplicate (pad)

   Let the rules for a procedure 'g' appear in the program in the order g_1, ..., g_n. The sequence g_1, ..., g_n is partitioned into m subsequences, called groups, G_1, ..., G_m, in such a way that each group is made either of rules which are all s.e. or of rules which are all non s.e. The rules in each group are uniquely renamed to form a new procedure; let g'_{1}, ..., g'_{m} be the names of these new procedures. Next, each program rule containing a reference to procedure 'g' is duplicated m times; in the i'th copy g is replaced by g'_{i}.

2. duplicate and subsume (das)

   Let the rules for a procedure 'f' appear in the order f_1, ..., f_n. Each program rule containing a reference to procedure 'f' is duplicated n times; the i'th copy incorporates rule f_i. This is performed by substituting the r.h side of f_i with appropriate variables renamed, for the call to 'f'.

   The transformation is done "bottom up" reflecting the directed acyclic graph nature of the s.e. definition. The pad operation partitions a procedure into s.e. and non s.e. procedures; the duplication ensures that all these procedures will be used in a derivation. The das operation is useful in treating non s.e. procedures. Essentially, rules for such procedures are percolated "upwards" to rules referencing them. The following example illustrates these operations.

Consider the program fragment:

(1) p(X,Y) :- a(X,Z), b(Z,Y), Y > 90, g(K,L,M).
(2) b(X,Y) :- c(Y,Z), f(Z), X < 100.
(3) c(X,Y) :- d(X,Y), !, fail.
(4) c(X,Y) :- h(X,Y), !, fail.
(5) c(X,Y) :- e(X,Y), e1(Y).
(6) c(X,Y) :- h(X,Z).
(7) g(K,L,M) :- h(K,L), i(L,M), j(M).
(8) g(K,L,M) :- h(K,L), j(M).

Terms that do not appear in the left hand of any rule correspond to base relations. In order to service procedure 'p' an equivalent serviceable procedure is generated in three steps:
Step 1:

Apply pad to procedure 'c'.

(3') \texttt{c1(X,Y) :- d(X,Y), !, fail.}

(4') \texttt{c1(X,Y) :- h(X,Y), !, fail.}

(5') \texttt{c2(X,Y) :- e(X,Y), e1(Y).}

(6') \texttt{c2(X,Y) :- h(X,Z).}

(2') \texttt{b(X,Y) :- c1(Y,Z), f(Z), X < 100.}

(2'') \texttt{b(X,Y) :- c2(Y,Z), f(Z), X < 100.}

Step 2:

Apply das to procedure 'c1'.

(2'.1) \texttt{b(X,Y) :- d(Y,Z), !, fail, f(Z), X < 100.}

(2'.2) \texttt{b(X,Y) :- h(Y,Z), !, fail, f(Z), X < 100.}

(Actually, the part after 'fail' is irrelevant; also, note that now procedure c1 is no longer referenced and hence may be discarded.)

Step 3:

Apply das to rule 'b'.

(1') \texttt{p(X,Y) :- a(X,Z), d(Y,M), !, fail.}

(1'') \texttt{p(X,Y) :- a(X,Z), h(Y,M), !, fail.}

(1''') \texttt{p(X,Y) :- a(X,Z), c2(Y,N), f(N), Z < 100, Y > 90, g(K,L,M).}

3.2. Creation of a Support Structure for Serviceable Rules

We distinguish between two steps in the creation process:

(1) Creation of a rule schema.

(2) Creation of a qual tree for the schema.

3.2.1. Creation of a Rule Schema

In order to encode its term schema, for each supportable term we add the clause

\[
\text{relation(schema_name,relation_name,[a_1,a_2,...,a_n]).}
\]

The 'schema_name' indicates the rule schema, relation_name is the name of the term schema and the a_i's are the attributes corresponding to the variables in the term (see section 2.2). Supportable terms which are not base relations are associated with distinct rule schemas. Each rule schema gets a unique 'schema_name'. For an immediate term a special axiom is added to the rule base
indicating the extra work necessary in later steps. The format used for such an axiom is:

\[
\text{constraint}(\text{schema}_1, \text{schema}_2, \text{functor}_name, [a_1, a_2, ..., a_4]).
\]

The \text{\textit{r}}_i's indicate the relation names on which the constraint operates;\text{\textit{functor}}_name is the operator name (e.g. \text{\texttt{<}}, \text{\texttt{...}}), and the \text{\textit{a}}_i's are the attributes associated with the relations.

\text{example:}

\begin{verbatim}
query(A,B) :- r1(A,B), r2(A,C), r3(D,B), r4(D,E), r5(D,F,'sf'), E > 100.
question(A) :- r1(B,A).
\end{verbatim}

\begin{verbatim}
r1(5,8), r1(5,15), r1(9,17), r1(19,55), r1(13,55).
r2(1,6), r2(3,1), r2(7,8), r2(9,1), r2(18,8).
r3(33,15), r3(47,8), r3(55,8), r3(99,17), r3(90,55).
r4(1,101), r4(3,90), r4(47,95), r4(55,600), r4(90,400), r4(99,500).
r5(33,jones,sf), r5(47,white, sf), r5(55,brown, sf), r5(90,smith, sf).
r5(99,white, ny).
\end{verbatim}

Variables begin with capital letters. Each \text{\textit{r}}_i is a base relation.

The term schemas for the rule 'query' are:

\textit{relation} name \hspace{1cm} attributes
\begin{align*}
\text{\textit{r}}_1 & \quad (a_1, a_2) \\
\text{\textit{r}}_2 & \quad (a_1, a_3) \\
\text{\textit{r}}_3 & \quad (a_4, a_2) \\
\text{\textit{r}}_4 & \quad (a_4, a_3) \\
\text{\textit{r}}_5 & \quad (a_4, a_6, a_7).
\end{align*}

\text{constraint:} a_7 = 'sf'.
\text{constraint:} a_5 \geq 100.
\text{target attributes:} (a_1, a_2).
\text{target name:} query.

This information is represented in PROLOG as:

\begin{verbatim}
relation(s1,r1,[a1,a2]).
relation(s1,r2,[a1,a3]).
relation(s1,r3,[a4,a2]).
relation(s1,r4,[a4,a5]).
relation(s1,r5,[a4,a6,a7])
target(s1,query,[a1,a2]).
\end{verbatim}
constraint(s1,[r4],'>',[a5,100]).
constraint(s1,[r5],'=',[a7,'sf']).

Similarly, rule 'question' is represented in PROLOG as:
relation(s2,r1,[a1,a2]).
target(s2,question,[a2]).

In order to service a procedure a rule_schema is created for each rule in the procedure. All these rule schemas have the same 'target_name'.

3.2.2. Qual Tree

For each rule schema a corresponding qual tree is created. The qual tree is required for updating the fact sets when the underlying rules change. The algorithm for building a tree is as follows:

1. If the rule is cyclic then transform it to an acyclic one by adding new terms called template terms.
2. Generate a qual tree (e.g. by using the algorithm in section 2.2).
3. Check inequality constraints: each inequality associated with two relations is treated as follows:
   [3.1] If the two relations are identical (e.g. [r1,r1]), then this constraint is treated as a selector on the relation r1.
   [3.2] If there already exists an edge between the two relations in the tree, or this inequality can be derived from existing tree edges and constraints then we are done.
   [3.3] Otherwise add an extra edge to the tree. If this results in a cyclic rule schema then add template terms to remove the cycle.

If there are more than one schema with the same 'target_name', an extra node called a union node is created. This node will be the parent node of all the sub trees having the same target_name. A union node is not created for the rules of the procedure we wish to service.

We keep the information about the tree as follows:

\[\text{edge}(s_n,r_i,[r_f,\ldots,r_k]).\]

Where \(r_f,\ldots,r_k\) are the children of \(r_i\) in the tree, in rule schema \(s_n\). We differentiate between ordinary terms and template terms by adding the clause:

\[\text{template}(\text{schema	extunderscore name},\text{relation	extunderscore name}).\]

\textbf{Example:}

The tree for the rule scheme 'query' is:

```
query
 /|
 r1 / \r
r2 r3 / \r
```

Technion - Computer Science Department - Technical Report CS0337 - 1984
r4  r5

In PROLOG this tree is kept as:

\begin{verbatim}
edge(s1,query,[r1]).
edge(s1,r1,[r2,r3]).
edge(s1,r3,[r4,r5]).
root(s1,query).
\end{verbatim}
in PROLOG this tree is kept as:

edge(s1, query, [r1]).
edge(s1, r1, [r2, r3]).
edge(s1, r3, [r4, r5]).
root(s1, query).
Only those facts that satisfy the constraints are inserted into the structure (see e.g. \( r_4, r_5 \) in the example). Facts that belong to a leaf relation are marked as good. A fact that belongs to a node is represented in a child of that node if each of its components have either the same value (for the same attributes) in that node and in some fact in the child node, or, in the case of constraints, have values that satisfy the constraints. Facts that belong to an internal node are checked for representation in every child of that node. If so - the fact is added to the node as a good one. Else, the fact is marked as a bad one and its marks list is appended with those children that do not represent the fact.

Information about a fact in PROLOG is represented as:

\[
\text{fact}(s_\text{m}, r_\text{i}, [d_1, \ldots, d_n], [r_j, \ldots, r_k])
\]

Where \( r_\text{i} \) denotes the relation to which the fact belongs. \([d_1, \ldots, d_n]\) is the list of values contained in the fact, and \([r_j, \ldots, r_k]\) is the marks list.

An internal node may correspond to a template term. Such a node is populated by joining already populated relations, called generators, whose attributes union spans the template attributes. Two possible ways for populating a template are:

- **Method 1:** Only good facts from the generators are joined.
- **Method 2:** All the facts from generators are joined.

In method 1 the template contains only good facts and is therefore smaller than the template created by method 2. On the other hand, method 1 has a higher overhead when changes occur.

The population of union nodes is different than that of other nodes. A union node is populated by good facts from already populated roots of the rule schemas for which the union node was created. A fact appears in the union node only once, and the marks associated with that fact indicate in which descendent nodes the fact appears as 'good'. This method saves time when looking for information in the union node, the marks help us when changes occur in the descendent nodes.

### 4.2. Changes in the Underlying Rules

When the support structure is populated, the good facts at the root determine the answer for each serviceable term that can be unified with the rule. Changes in the underlying rule base may affect the status of facts at the root: good ones may become bad and vice versa.

We allow changes (insertions and deletions) only in the base facts (see section 2.1). Upon insertion of a fact, bad facts at upper tree levels may turn into good ones. This transformation of bad facts into good ones may propagate up to the root (and to other trees in the support structure if the relation represented by the root appears as an internal node in other trees). Facts that were previously good remain so. Upon deletions, facts that were previously good may now turn into bad ones and, as in the insertion case, the changes may propagate.

Here are the PROLOG procedures skeletons for insertion and deletion:

\[
\text{insert}(\text{Fact}, \text{Relation}) \leftarrow \text{relation}(\text{Schema}, \text{Relation}),
\]

\[
\text{asserta}(\text{fact}(\text{Schema}, \text{Relation}, \text{Fact}, \text{Mark}\_\text{list})),
\]

\[
\text{check\_below}(\text{Fact}, \text{Relation}, \text{Mark}\_\text{list}),
\]

\[
\text{empty}(\text{Mark}\_\text{list}),
\]

\[
\text{check\_insert\_above}(\text{Fact}, \text{Relation}, \text{Schema}),
\]

\[
\text{fail}.
\]

This program illustrates how a fact \text{Fact} can be inserted into relation \text{Relation} in every schema in which the relation appear. The procedure 'check\_below' checks the status of Fact: if it is a good fact then \text{Mark}\_\text{list} will be empty, else \text{Mark}\_\text{list} will contain the marks list. Procedure 'check\_insert\_above' is invoked when \text{Fact} is determined to be good. It propagates the changes (recursively) upwards to the
father of Relation. The 'fail' causes the procedure to continue the process for every rule schema in which relation Relation appears.

(1) delete(Fact,Relation) :- relation(Schema,Relation,_) delete_fact(Schema,Fact,Relation), fail.

(2) delete_fact(Schema,Fact,Relation) :- fact(Schema,Relation,Fact,[[]],!), retract(fact(Schema,Relation,Fact,[[]])), check_delete_above(Fact,Relation,Schema).

(3) delete_fact(Schema,Fact,Relation) :- fact(Schema,Relation,Fact,[[]]),!, retract(fact(Schema,Relation,Fact,[[]])).

The fact Fact is deleted and possible propagation effects are checked for in every rule schema in which relation Relation appears. Rule no. 2 treats the deletion of a good fact which may have propagation effects. Rule no. 3 treats the deletion of a bad fact for which there are no propagation effects.

4.3. A Service Program

If there are no loose constraints in a serviced rule then service is simple. In the previous example:

query(X,Y):-root_fact(s1,query,[X,Y]).

Where 's1' is the rule schema and 'query' is the name of the root structure. Procedure root_fact is straightforward and it is used for accessing structures. We demonstrate the service program in the presence of PROLOG 'cut' and an inequality loose term.

Consider the following example:

a(X,Y,Z) :- b(X,Y), !, c(Y,Z).
a(X,Y,Z) :- b(X,X), c(Z,Z), Y > 50.
a(X,Y,Z) :- c(Y,Z).

Where 'b' and 'c' are base relations. All possible relevant facts are mentioned in the structures. The service program is:

a(X,Y,Z) :- root_fact(s1,a,[X,Y,_]), !, root_fact(s1,a,[X,Y,Z]).
a(X,Y,Z) :- root_fact(s2,a,[X,Z]), Y > 50.
a(X,Y,Z) :- root_fact(s3,a,[Y,Z]).
5. DYNAMIC RULE SUPPORT

5.1. The Basic Mechanism

The idea behind dynamic rule support is to incrementally maintain support structures. The amount of work invested in maintaining the support structures is related to the number of activations of the rule. The dynamic support mechanism relies on the structures constructed in section 3. We illustrate the dynamic rule support concept with a simple example.

Consider the rule:

\[ q(X) :- p(A,B,C,X), c(A,B,D), d(A,C), e(B,D,E). \]

Suppose b,c,d and e are base relations.

The above rule is a tree rule as depicted by the following qual graph:

\[ x----abcx----ac \]
\[ \sim-\sim-\sim \]
\[ j \]
\[ \simbd----bde \]
\[ \sim-\sim-\sim \]

The support structure will consist of two new relations bb and cc corresponding to the internal tree nodes 'abcx' and 'abd', respectively. Essentially, facts maintained in cc will be "supported" by facts in e and those in bb will be "supported" by relations c,d and e. Initially both bb and cc are empty; as q(X) is activated, facts may migrate from c to cc or from b to bb. When facts are deleted some facts may migrate from cc to c or from bb to b.

Recall that the original rule is transformed to allow for dynamic support. The result of this transformation on q(X) is illustrated below.

1. \[ q(X) :- bb(A,B,C,X). \]
2. \[ q(X) :- b(A,B,C,X), okc(A,B,D), d(A,C), \]
   \[ \quad \text{retract}(b(A,B,C,X)), \text{uniqueasserta}(bb(A,B,C,X)). \]
3. \[ \text{okc}(A,B,D) :- cc(A,B,D). \]
4. \[ \text{okc}(A,B,D) :- c(A,B,D), e(B,D,E), \]
   \[ \quad \text{retract}(c(A,B,D)), \text{uniqueasserta}(cc(A,B,D)). \]

The first rule tries to satisfy q(X) by stepping through all known bb facts which contain all the answers deduced thus far. Observe that no attempt is made to prevent duplicate satisfaction of q(X) (although this could be done with an extra rule). Once all known results in bb are exhausted, a search for new answers commences in rule 2. A fact \((a_1,b_1,c_1,x_1)\) is drawn out of b using \(b(A,B,C,X)\) and okc is called to verify that \((a_1,b_1,c_1,x_1)\) is supported by facts in c and e; if it is, then support from d is checked as well. In case \((a_1,b_1,c_1,x_1)\) is supported by c,d and e this fact is moved from b to bb. The predicate uniqueasserta prevents duplicate facts. Rule 3 indicates that \(\text{okc}(a_1,b_1,d_1)\) is satisfied if it has been previously derived and put into cc. Rule 4 presents another way to satisfy \(\text{okc}(a_1,b_1,d_1)\) by locating a fact \((a_1,b_1,d_1)\) in c and a supporting fact \((b_1,d_1,e_1)\) in e. Again, once a support for a fact \((a_1,b_1,d_1)\) in cc has been derived this fact is moved from c to cc.

The above rules use already derived knowledge instead of deriving facts anew. When the database is stable, i.e. no facts are inserted or deleted, as the rule q(X) is activated more and more knowledge is accumulated in the form of bb and cc facts. Thus new procedure calls q(X) may use already derived knowledge and
when this is exhausted new derivations are tried which may lead to yet more stored knowledge. The problem is that few databases are that stable and hence we must explain how new facts are inserted and how facts are deleted.

Insertions may affect the validity of the current knowledge, this does not happen in our example and hence for the time being assume that insertions are simply performed into the original relations, i.e. b,c,d and e. Deletions may invalidate derived facts, as illustrated in a continuation of our example.

(5) detractx(b(A,B,C,X)) :- retract(bb(A,B,C,X)).
(6) detractx(b(A,B,C,X)) :- retract(b(A,B,C,X)).
(7) detmove(b(A,B,C,X)) :- bb(A,B,C,X), retract(bb(A,B,C,X)), asserta(b(A,B,C,X)).
(8) detractx(c(A,B,D)) :- cc(A,B,D), retract(cc(A,B,D)), not(cc(A,B,D)), detmove(b(A,B,C,X)).
(9) detractx(c(A,B,D)) :- retract(c(A,B,D)).
(10) detmove(c(A,B,D)) :- cc(A,B,D), retract(cc(A,B,D)), asserta(c(A,B,D)), not(cc(A,B,D)), detmove(b(A,B,C,X)).
(11) detractx(e(B,D,E)) :- retract(e(B,D,E)), not(e(B,D,E)), detmove(cc(A,B,D)).
(12) detractx(d(A,C)) :- retract(d(A,C)), not(d(A,C)), detmove(bb(A,B,C,X)).

Deletions are performed using the rule detractx and are reflected in both original and derived relations. Rule 5 states that if a fact (a1,b1,c1,x1) is deleted, the fact is first searched for in bb and it is removed if found. Otherwise, by rule 6, deletion is performed from b. Rule 8 states that deletion of a fact (a1,b1,d1) is first attempted from cc. In case the fact was in cc it is deleted and if no other fact in cc has the same A,B components a fact in bb may be invalidated. This is checked for by rule 7, bb facts which are no longer supported are located and are moved from bb to b. In case (a1,b1,d1) is not found in cc the deletion is carried out from c (rule 9). Deletion of a fact (b1,d1,e1) is performed on e (rule 11). This may invalidate cc facts which are handled in rule 10. Observe that once facts are moved from cc to c in rule 10 and there may be unsupported facts in bb, rule 7 is invoked. Rule 12 handles deletions from d.

The above rules (1-12) illustrate how a simple rule q(X) may be dynamically supported over time using a straightforward program transformation. The transformed rules compare favorably with the ordinary depth first search PROLOG execution. First, a new procedure call on q(X) utilizes knowledge accumulated by previous calls. Second, newly derived knowledge is stored for future use. Third, each deriving step which PROLOG would have ordinarily done, is replaced by a constant number of steps in the transformed program. Other PROLOG derivations are substituted with simple accesses to derived facts.

There is a high cost which may be associated with a single insertion or deletion. For example, a fact deleted out of relation e in our example could affect the migration of many facts from cc to c or from bb to b. However, we may "charge" the cost of this motion to the derivation which has installed that derived fact which is now migrating. As each fact is installed once and migrates back once,
the overall cost is still bounded by a constant times the cost PROLOG would have "normally" invested. In other words, the worst case complexity of a sequence of operations in the transformed program is the same as that of the original program. Of course, substantial savings may be realized when the number of insertions and deletions is relatively small, or when there are many "independent" procedure invocations each starting from "scratch". In essence, the worst case behavior is a manifestation of non-monotonicity which turns the support mechanism into pure overhead.

5.2. The Spectrum of Support Mechanisms

There seems to be a spectrum of possible dynamic support mechanisms starting from the basic mechanism which we have exhibited and extending to the a priori support discussed in Section 4. These mechanisms differ according to the amount of work which is invested in maintenance. Of course, this depends on the environment in which the mechanism is to operate. A priori support is attractive in a monotonic environment in which almost all of the supported facts are expected to be accessed. Dynamic support is attractive when the portion of accessed supported facts is unknown in advance.

Let us denote the dynamic mechanism described previously as Method A. There are other possibilities (Methods B and C below).

Method A:
Only valid facts are maintained in the support structures bb and cc.

Method B:
Facts which were once derived but are no longer valid may also be kept in the support structures (such as bb and cc in the example). The mark mechanism discussed in describing a priori support is used to indicate why certain facts are no longer valid. No attempts are made to remove these marks during inserts. So, some facts marked as 'bad' may actually be valid. The idea behind Method B is to use 'half valid' facts as first candidates for satisfaction. Curiously, the overall cost is still bounded by a constant times the cost PROLOG would have normally invested.

Method C:
Method C is identical to Method B except that when new facts are derived an attempt is made to remove marks which are no longer needed. This method tends to maintain more valid facts at a somewhat higher cost. The overall cost is no longer bounded by a constant times normal PROLOG cost.

Another optimization which may be used in conjunction with any of the above methods is a mechanism called scan continuation. In a rule activation which can not be satisfied with current knowledge, a search for new derived facts is initiated. However, this search may re-scan many facts which have already led nowhere in the past. It would be more efficient to start the search from the latest previous search state. If such a mechanism is not offered by PROLOG (and it is not offered by the system we use) it may be simulated by leaving "road marks". A complication with scan continuation results from insert operations. Such operations may invalidate the continuation mechanism. It can be shown that the scans must be reset to "the beginning" for all ancestors (original relations) of the relation (into which the insert is performed) in the qual tree used to derive the support structure.

5.3. Handling Templates

Basically, templates may be integrated into all the dynamic support methods. Certain options exist concerning what is stored in templates.

Option A:
Only facts which were explicitly derived are kept in templates. No attempt is
made to deduce more template facts.
Option B:
As many template facts as possible are derived and populate the templates. This is equivalent to maintaining the natural join of relations which can generate the template. Obviously, this method has a higher overhead.

In addition, the scan continuation idea may also be applied to generating "next" template facts.
In this paper we have proposed a program transformation method aiming at improving the efficiency of proof processes in rule-based PROLOG programs. The basic idea is trading time for space. In our approach the rule base is augmented with additional support knowledge and, with this additional knowledge, we show how to improve the efficiency of term resolution. As in many knowledge organization methods, the price to be paid for this improved service is the additional overhead in maintaining an up-to-date support structure when changes in the rule base occur.

We have demonstrated the method and have shown two variants of its implementation: the a priori and dynamic methods. From a database aspect, we have shown that the techniques of view maintenance, originally developed in the context of relational databases, readily carry over to the area of logic programming. In the development we have borrowed and adapted other relational database concepts, notably the graph representation of queries. The ease with which we were able to adopt these concepts supports our opening contention that PROLOG is indeed a suitable tool for a unified treatment of problems in the database area.

From the discussion we believe that the ideas presented are indeed feasible. The proposed methods here are among the first program optimization techniques, very much in the vein of the traditional programming languages, in the context of logic programming with applications to databases.

An important issue concerns the semantics of PROLOG. A program produced by our transformation methods does not always exhibit a behavior which is exactly identical to the original program. First, duplicate rules are not supported and secondly, the order in which facts are derived may differ from the source program. It is our contention that these issues are not critical in most database-oriented PROLOG applications.

More work is left to be done. The methods that we have proposed are two isolated cases from a spectrum of "hybrid methods" which combine these ideas in different ways. In this paper we have not touched upon the operational aspects of the methods and the performance gains that are to be expected. We hope to discuss these problems in our future work.
REFERENCES


