DYNAMIC RULE SUPPORT IN
PROLOG
(Extended Abstract)

by

O. Shmueli¹, S. Tsur², H. Zfira¹, R. Ever-Hadani¹,

Technical Report #370
June 1985

¹ Computer Science Dept., Technion-Israel Institute of Technology, Haifa, Israel.
² MCC, Austin, TEXAS, USA
1. INTRODUCTION

The similarities between Prolog and Relational databases have stimulated much research [EDS, Goe, JCV, KY, UU1, War]. Simplistically, Prolog may be regarded as a rich query language operating on a database composed of (ground clauses) facts. When the set of facts becomes large a number of performance problems arise. This paper describes an attempt at improving Prolog’s performance in executing repetitive queries over large fact sets.

The depth-first search mode in which Prolog traverses its search space may lead to some well-known inefficiencies [War, BP]. One problem is backtracking which may take place when it a priori cannot provide new search options. Another source of inefficiency is the redoing of a previous derivation; backtracking may yield bindings for forward processing which have already been addressed (successfully or not).

The above problems are associated with a single derivation. Inefficiencies may also result from work repetition due to multiple activations. Consider a goal \(q(A)\) that has been activated by an occurrence of the term \(q(A)\) within rule \(p\). If \(p\)'s activation backtracks, then \(q\) "supplies" a number of different answers. Once \(p\)'s activation has terminated none of these supplied answers is "remembered". Thus, if rule \(p\) or another rule referencing \(q\) is re-activated, \(q\)'s derivations will be redone, possibly repeating the same derivations. (Of course, if the new activation starts with new and different variable instantiations, or the database has changed, then a new derivation may be necessary.)

In trying to avoid re-deriving facts it is insufficient to simply remember results of previous activations. Facts may have been deleted which could invalidate previous derivations. Therefore, we are faced with a truth maintenance type problem [Boy, MD]. To complicate matters, Prolog contains non-logical operations (e.g. the "cut") and a very flexible variable binding mechanism which adds another layer of complexity to the maintenance problem.

1. Computer Science Department, Technion - Israel Institute of Technology, Haifa, Israel.
2. Computer Science Department, Technion - Israel Institute of Technology, Haifa, Israel.
3. Computer Science Department, Technion - Israel Institute of Technology, Haifa, Israel.
4. Computer Science Department, Technion - Israel Institute of Technology, Haifa, Israel.
In this paper we examine the idea of dynamic information maintenance. The basic philosophy is that the amount of work invested in maintenance should be determined by the amount of information required by proof processes. Another approach, that of continuously maintaining all possible answers was suggested in [STZ].

A mechanism is proposed for collecting and dynamically maintaining the status of derived facts. The maintenance "data structures" are motivated by the concept of acyclicity in relational databases [Ull] and the view maintenance ideas in [Si]. The transformation is guided by user maintenance directives. In order to use this method, the original program need be transformed into a certain form. It should be noted that the semantics of "ordinary" Prolog is altered in the transformation. We claim that this alteration is reasonable when dealing with a large internal database of facts.

There are several options for implementing the mechanism. First, the Prolog interpreter can be modified so that maintained rules are handled efficiently by using low level constructs. Second, the user program can be further transformed so that the maintenance algorithm is expressed in Prolog; transportability is an obvious advantage. Third, certain features are added to Prolog so that option two can be realized more efficiently. Such features may include: pointers, in-place "assertions", specifying fact scan directions as well as start and end points within the fact database, using indexes etc. So, the third approach is a compromise between the efficiency of option one and the transportability of option two. For didactic reasons it is option two which we present in this paper.

The paper is organized thus. Section 2 presents basic terminology. In section 3, using a simple rule as an example, the basic idea of dynamic rule support is exhibited using a specific mechanism. In section 4 the transformation of a user program into a canonical form is sketched. Section 5 details a dynamic support maintenance strategy. Section 6 describes additional mechanisms and sets directions for future research. Some preliminary performance experiments are reported in section 7.

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 = X. By comparison, a base fact is a tuple in the relational sense. We use the term base relation for a collection of base facts all having the same axiom name.

From now on we 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 < 10 is equivalent to the rule a(X,Y,Z) :- X < 80, Y = 3, Z = 1. Occasionally a rule is replaced by a collection of evaluated instances. These instances are called derived facts. In many cases we do not distinguish between base and derived facts, but simply refer to them facts.

We assume that prior to processing, all Prolog rules have been normalized to conjunctions of right hand terms.
The examples in this paper use a backtracking limiting operator called \( \mathit{braces} \), denoted \('[[' \]. The meaning of this operator is as follows. The left bracket \(' \) always succeeds. When the right bracket \(' \) is reached during backtracking then backtracking continues at the goal immediately to the left of the left parenthesis \(' \). The braces operator can be nested.

**Example:**

\[
q := \{x, [y], z\}, \ w, \ [t, u], \ v.
\]

is equivalent to the following sequence of rules:

\[
q := q_1, \ w, \ q_2, \ v.
\]

\[
q_1 := x, \ q_3, \ z, \ !.
\]

\[
q_2 := t, \ u, \ !.
\]

\[
q_3 := y, !.
\]

### 2.2. Defining Correct Support

Let \( p(X_1, \ldots, X_n) \) be a Prolog\(^1\) procedure, and let \( Y_1, \ldots, Y_n \) be either instantiated or uninstantiated variables. The set \( S \) returned by \( p \) with parameters \( Y_1, \ldots, Y_n \) is the set of facts installed by:

\[
:- p(Y_1, \ldots, Y_n), \ \text{assert}(s(Y_1, \ldots, Y_n)), \ \text{fail}.
\]

Procedure \( p' \) maintains procedure \( p \) if for all parameters \( Y_1, \ldots, Y_n \) if \( S \) is returned by \( p' \) then there exists an ordering of the facts in the database such that \( S \) is returned by \( p \). Basically, the above states that \( p' \) maintains \( p \) if \( p' \) returns the same set of results as \( p \), although each is allowed to look at a different permutation of the facts in the database. This is consistent with a point of view in which base relations are considered as sets and scanning them returns the tuples in some order.

### 2.3. Definitions

**Definition - Immediate Terms**

Terms that are not unified (in the Prolog sense) are called **immediate terms**, e.g. negation ("not"), "fail", \( K \oplus C \), \( X \Theta Y \), where \( X, Y \) are variables, \( C \) is a constant and \( \Theta \) is a comparison operator.

**Definition - Semi Elementary Term.**

A term in a body of a rule is semi elementary in exactly one of the following cases:

1. All its corresponding rules are base facts.
2. It is of the form \( X = C \) where \( X \) is a variable and \( C \) is a constant.
3. It is of the form \( X \leq Y \) where \( X \) or \( Y \) appear in a semi elementary term written prior to this one.
4. All its corresponding rules are semi elementary (see below):
Definition - Semi Elementary Rule.

A rule is semi elementary exactly when it satisfies both of the following constraints:

1. Every head variable of the rule appears in a semi elementary term in the body of the rule.
2. Each of its r.h. terms is either semi elementary or an immediate term over attributes set S of the form X Θ C (with S = \{X\}) or X Θ Y (with S = \{X,Y\}) or not(...), such that all variables in S appear in semi elementary terms in the body of the rule which are written prior to t.

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 is 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 written prior to this inequality.
2. It is the Prolog axiom 'fail'.
3. It is a Prolog negation predicate 'not' which involves variables not appearing in any other semi elementary term of the rule written prior to the 'not'.

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

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

2.4. Graph Representation of Rules

In this subsection, we will relate the above concepts with known relational database concepts [111]. Consider an elementary Prolog rule possessing a r.h. side in the Prolog subset of interest. The rule has the form:

\[ q(X_1, \ldots, X_n) \leftarrow \bigwedge_{i=1}^{m} p_i(Y_{i1}, \ldots, Y_{in}). \]

where \( (X_1, \ldots, X_n) \) is the set of target variables and the terms on the r.h. side are called the qualification. The \( X_i \)'s are variables and the \( Y_j \)'s may be \( X_i \)'s, constants or variables.

We need to encode the way in which 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 (resp. rule) schema corresponds to a relation (resp. database) schema in the relational databases terminology. In fact, we shall use term schemas as relation schemas for relations (in the database sense) used in the support structure.

A qual graph 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:

\[ q(A,B,X) \leftarrow p(A,B,C,X), s(A,C), t(A,B,D), r(B,D). \]

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
    \|      /  \n    abs    bd
```

**Example:**

Consider the rule:

\[ q(A,X) \leftarrow p(A,B), r(A,C), s(B,C,D), t(B,D). \]

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

```
ab    ac
   / \   
  \   /
  bed
```

A 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 a 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 all r.h. terms are eliminated. If the input rule is in fact a tree rule then \( T \) is a qual tree for the rule.

3. Dynamic Rule Support

3.1. The Basic Mechanism

We illustrate the dynamic rule support concept with a simple example. Consider the rule:

\[ q(A,B,X) \leftarrow p(A,B,C,X), s(A,C), t(A,B,D), r(B,D). \]
\[ q(X) :\neg b(A,B,C,X), c(A,B,D), d(A,C), e(B,D,E). \]

Suppose \( b, c, d \) and \( e \) are base relations. Rule \( q(X) \) is a tree rule as depicted by the following qual graph:

\[
\begin{array}{c}
x \\
\downarrow \text{abcx}
\end{array}
\]

\[
\begin{array}{c}
ab \\
\downarrow \text{bcx}
\end{array}
\]

The support structure will consist of two new relations \( bb \) and \( cc \) corresponding to the internal tree nodes "abcx" and "ab", 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 \( cc \) and from \( b \) to \( bb \), when facts are deleted some facts may migrate from \( cc \) to \( c \) or from \( bb \) to \( b \).

The original rule is transformed to allow for dynamic support (see section 5.3). The result of this transformation on \( q(X) \) is:

(1) \( q(X) : b(A,B,C,X). \)
(2) \( q(X) : b(A,B,C,X) \left\{ \text{okc}(A,B,\_). \right\} \left\{ \text{d}(A,C) \right\}, \text{retract}(b(A,B,C,X)), \text{uniqueasserta}(bb(A,B,C,X)). \)
(3) \( \text{okc}(A,B,D) : cc(A,B,D). \)
(4) \( \text{okc}(A,B,D) : c(A,B,D), \left\{ \text{e}(B,D,\_). \right\}, \text{retract}(c(A,B,D)), \text{uniqueasserta}(cc(A,B,D)). \)

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 real attempt is made to prevent duplicate satisfaction of \( q(X) \) (although this could be done with some extra work). 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 \( \text{okc} \) 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. The \{ \} operator is used to eliminate the search for more evidence for support which is not needed. In case fact \((a_1,b_1,c_1,x_1)\) is supported by \( c, d \) and \( e \) this fact is moved from \( b \) to \( bb \).

The predicate \( \text{uniqueasserta} \) prevents the installment 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 \( c \) has been derived this fact is moved from \( c \) to \( cc \). The braces, again, prevent useless scanning of "e" facts.

The above rules use previously 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 a new procedure called \( q(X) \) may use already derived knowledge. When this knowledge is exhausted, new derivations are tried which may lead to yet more stored knowledge. The problem is that few database's are that stable and hence we must specify how new facts are inserted and how facts are deleted.

Insertions are simply performed into the original relations, i.e. \( b, c, d \) and \( e \). We assume that a deletion implies the removal of at most one qualifying tuple. Deletions may invalidate derived facts, as illustrated in a continuation of the
example.

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

Deletions are performed using the rule \texttt{detractx} and are reflected in both original and derived relations. Rule 5 states that if a fact \((a_1,b_1,c_1,x_1)\) is deleted, it is first searched for in \(bb\) and removed if found. Otherwise, by rule 6, deletion is performed from \(b\). Rule 8 states that deletion of a fact \((a_1,b_1,d_1)\) 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 some facts in \(bb\) may be invalidated. This is checked for by the first alternative for procedure 7; \(bb\) facts which are no longer supported are located and are moved from \(bb\) to \(b\). In case \((a_1,b_1,d_1)\) is not found in \(cc\) the deletion is carried out from \(c\) (rule 9). Deletion of a fact \((b_1,d_1,e_1)\) is performed on \(e\) (rule 11). This may invalidate \(cc\) facts which are handled in procedure 10. Observe that once facts are moved from \(cc\) to \(c\) in procedure 10 and there may be unsupported facts in \(bb\), procedure 7 is invoked. Rule 12 handles deletions from \(d\).

There is a high cost which may be associated with a single 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 at most once for each installment, the overall cost is bounded by a constant times the cost Prolog would have "normally" invested (assuming a hash based \(O(1)\) fact locating method). 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". The worst case behavior is a manifestation of non-monotonicity which may convert the support mechanism into pure overhead.
3.2. Utilizing More Information

A disadvantage of the method described above is that information is only kept about 'good' facts. Consider the following example: during a scan of relation \( R \) it is found that tuple \( t \) is supported by tuple \( t' \) in \( c \) and that \( t \) is supported by no tuple in \( d \). This information is not retained. Therefore, if tuple \( t \) is re-scanned, assuming the database remains stable, then again it will be found that \( t \) is supported by \( t' \) but is not supported by any tuple in \( d \).

The redundant \( cc \) search above could be avoided if partial-match information is also maintained. One solution is to associate with each tuple \( t \) in relation \( R \) a down pointer for each of \( R \)'s children. In the above example, once \( t' \) is found to support \( t \), the down pointer in \( t \) records this knowledge by pointing at \( t' \). Therefore, if \( t \) is re-scanned, its down pointer to \( t' \) could be used to avoid re-scanning relation \( cc \). Initially, a down pointer is nil, i.e. pointing to no tuple.

Of course, the pointing information may be invalidated due to changes in the database. In the above example, \( t' \)'s may migrate back to \( c \) and when \( t' \)'s pointer is followed \( t \) will not be found. It seems that \( cc \) must be scanned a new. One way to utilize "old non-valid pointers" is to use them for scan continuation. Continuing the previous example, instead of scanning \( cc \) from the beginning, the scan should start at the point \( t' \) used to reside. Utilizing the scan continuation concept depends on the availability of this feature in the Prolog implementation used.

The scan continuation concept is useful only for scanning known 'good' facts. Once all good facts are scanned, the scan must traverse the rest of the facts. The continuation idea may be extended to known 'bad' facts as follows. If no support is found for a tuple \( t \) then \( t \)'s down pointer should point at the last (bad fact) scanned below. The next time a scan is performed on behalf of \( t \), the scan continues at the pointed 'bad' tuple below.

The extension just described has a major flaw. The fact that support did not exist before does not imply that it does not exist now. Insertions into descendant relations may cause a re-scan of previously scanned facts to become successful. Thus down pointers into 'bad' tuples should be used for scan continuation only if no insertions into descendant relations took place. A mechanism for enforcing this convention is obviously needed.

Further performance gains may be achieved by avoiding forward searches whose failure can be a priori determined. With each rule activation failure lists are formed. A failure list at a node is associated with a child of that node. It contains those values, on attributes in common between the parent and the child, for which failure occurred on a previous call. Thus, before calling a child with certain values, the failure list associated with that child is checked and if the values appear there, then no calling is performed. This avoids futile searches which may scan a large portion of the internal database. The failure lists are valid as long as no insertions into descendant relations take place. For clarity of presentation, the failure list and down pointer options will not be further discussed or included in examples.

4. 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 some non serviceable rule may be transformed into a correctly maintained procedure. This transformation may affect other procedures which take part, directly or indirectly, in the definition of the procedure that need be serviced. The transformation succeeds precisely when the definition of \( p \) is not recursive; i.e. there is no sequence of procedures \( p=p_1, \ldots , p_n=p_1, n>1, \) such that for \( i=1 \ldots n-1 \) procedures \( p_i \) has a rule with a r.h.
term referencing procedure $p_{q+1}$.

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 m.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 'p' containing a reference to procedure 'g' is duplicated m times; in the i-th copy $g$ is replaced by $g'_i$. If neither 'p' nor 'g' contains cuts, then the pad operation preserves the semantic structure of the original program [26].

2. **duplicate and subsume (das)**

   Let the rules for a procedure 'f' appear in the order $f_1, ..., f_n$. Each program rule 'p' 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'. If neither 'p' nor 'f' contains cuts, then the das operation preserves the semantic structure of the original program [26].

The transformation program is sketched below:

1. Sort the support requests (of the form 'maint(q)') so that if procedure p uses (directly or indirectly) procedure q, then 'maint(q)' appears before 'maint(p)'; this is basically a topological sort. For each 'maint(q)' request perform the following steps.

2. If $q$ has a rule containing cuts in its r.h. side then proceed as follows. Replace each sub-rule consisting of a sequence of terms over variables $X$ appearing between two adjacent cuts in 'q' by a new term $p(Y)$; $Y \subseteq X$ are those variables which also appear in other sub-rules of $q$ or in $q$'s head. Add a new rule $p(Y)$ to the database with the sub-rule as its r.h. side. Add a request 'maint(p)'. Do the above, in turn, for each sub-rule of $q$. Handle the new 'maint(p)' requests and only then re-handle 'maint(q)' for other $q$ rules. (the q rule just handled need not be re-considered).

**Example:**

consider the following program:

$$q(X,Y) :- b(X,D,C,E), d(A,C), l, c1(A,B,D), a(A,E,F,Y), l, e(F,G,E), f(E).$$

maint(q(X,Y)).

The transformed program is:

$$q(A,B) :- q1(C,D,A), l, q2(G,H,B,C,D), l, q3(H,G).$$

$$q1(C,D,A) :- b(C,D,A), d(C,E),$$

$$q2(G,H,B,C,D) :- c1(C,D,F), a(C,G,H,B).$$

$$q3(H,G) :- e(H,G), f(G).$$

maint(q1(C,D,A)).

maint(q2(G,H,B,C,D))

maint(q3(H,G)).

maint(q(X,Y)).
(3) Procedure \( q \) does not contain any cuts. For each rule in \( q \) scan the terms appearing in \( q \)'s r.h. side. If the rule contains only terms which match either supported procedures in which no rule contains a cut or base relations, then do nothing. Otherwise,

[3.1] If rule \( q \) has a term \( p \) which matches an unsupported procedure in which no rule contains cuts, then apply the pad and das operators to \( p \) and to procedures which are called, directly or indirectly, by \( p \). This 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.

[3.2] If rule \( q \) has a term \( p \) which matches a procedure containing cuts, then make \( p \) the last term of \( q \). Add the request 'maint(\( p \))' to the program. Handle 'maint(\( p \))'.

[3.3] Handle the "new" rules produced in [3.1] or [3.2].

(4) If \( q \)'s r.h. side contains terms which match procedures containing cuts, then replace the sub-rule appearing before those terms with a new term \( p \). Add a new rule \( p \) to the database having the replaced sub-rule as its r.h. side. Replace the original request 'maint(\( q \))' with a new one 'maint(\( p \))'. Handle 'maint(\( p \))'.

(5) Rename variables (into variables or constants) reflecting equality constraints (i.e. immediate terms of the form \( X=Y \) or \( X=C \)). Make all other immediate terms in \( q \) the last terms in \( q \).

Example:

(1) \( q(X, Y) :- e(X, Z), c(Z, Y), Y > 90, g(K, L, M) \).

(2) \( a(X, Y) :- b(Y, Z), f(Z) \).

(3) \( c(X, Y) :- e(Y, Z), \text{not } h(Z, X) \).

(4) \( c(X, Y) :- e(X, Y), e1(Y) \).

(5) \( g(K, L, M) :- h(K, L), i(L, M), j(M) \).

(6) \( g(K, L, M) :- h(K), i(L), j(M) \).

(7) \( \text{maint}(g(_, _, _)) \).

(8) \( \text{maint}(q(_, _, _)) \).

Procedure \( g \) displays the case of a supported procedure containing no cuts. Procedure \( c \) displays case [3.1] and procedure \( a \) displays case [3.2]. In order to service procedure \( 'q' \) an equivalent serviceable procedure is generated in four steps:

Step 1:

Apply pad to procedure \( 'c' \):

(3') \( e1(X, Y) :- e(Y, Z), \text{not } h(Z, X) \).
(4') \( e_2(X, Y) :- e(X, Y), c_1(Y) \).

(1.1) \( e(X, Y) :- a(X, Z), c_1(Z, Y), Y > 90, g(K, L, M) \).

(12) \( e_2(X, Y) :- a(X, Z), e_2(Z, Y), Y > 90, g(K, L, M) \).

\textbf{step 2:}

Apply case to procedure \( c_1 \).

(1.1') \( e(X, Y) :- a(X, Z), e(Y, W), \text{not } h(W, Z), Y > 90, g(K, L, M) \).

\textbf{step 3:}

Handling term \( a(K, Z) \) and the negation (case 6 & 8).

(1.1') \( e(X, Y) :- e(Y, W), Y > 90, g(K, L, M), a(K, Z), \text{not } h(W, Z) \).

(1.2') \( e(X, Y) :- e_2(Z, Y), Y > 90, g(K, L, M), a(X, Z) \).

(9) \text{maint}(a(_, _)).

\textbf{step 4:}

Handling case 4.

(1.1'') \( e(X, Y) :- q_1(Y, W), a(X, Z), \text{not } h(W, Z) \).

(12'') \( e(X, Y) :- q_2(Z, Y), a(X, Z) \).

(10) \( q_1(Y, W) :- e(Y, W), Y > 90, g(K, L, M) \).

(11) \( q_2(Z, Y) :- e_2(Z, Y), Y > 90, g(K, L, M) \).

(12) \text{maint}(q_1(_, _)).

(13) \text{maint}(q_2(_, _)).

\section{CONSTRUCTING A SUPPORT STRUCTURE}

\subsection{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 described below. Its goal is to build a qual tree such that each inequality either involves attributes appearing within some relation or in adjacent relations in the qual tree.

(1) Consider a schema for a negation term \( t \) over variables \( X \). Let \( Y \subseteq X \) be those variables which also appear in other term schemas in the rule schema. If there is no non-negation term schema containing \( Y \) then add \( Y \) as a new term schema, \( Y \) is called a template term. Do the above, in turn, for each negation schema.

(2) If the rule is cyclic then transform it to an acyclic one by adding templates. Several heuristic methods for choosing a template are given in [Z61]. Add a new rule \( p \) to the database over the template attributes with the template's generators (i.e., relations which can produce it) on its r.h. side. Rule \( p \) is
supported according to its own qual tree.

(3) Generate a qual tree (e.g. by using an algorithm similar to that given in section 2.2). Force the negation terms in the rule schema to be leaves in the tree; this is made possible by (1). The reasons for forcing negation terms to be leaves are given in [29].

(4) Check inequality constraints; each inequality associated with two attributes is treated as follows:

[4.1] If the inequality attributes appear within a single relation then treat this inequality as a selector on that relation.

[4.2] Else, if there already exists an edge between two relations in the tree, each containing one of the inequality attributes, or this inequality can be derived from existing tree edges and constraints, then do nothing.

[4.3] Otherwise, arbitrarily choose two relations, each containing one of the inequality attributes. To each of them, temporarily, add the other's attribute.

(5) If this results in a cyclic schema then add templates, as in (2), and transform it into an acyclic schema. Finally, produce a qual tree for the resulting schema and discard the attributes added in [4.3].

If there is more than one schema for t, with the same 'target_name', (i.e. there is more than one rule unifying with t) an extra node called a union node is created. This node will be the parent node of all the subtrees having the same target name. A union node is not created for the rules of the procedure we wish to service.

5.2. Templates

As mentioned before, a template is created in order to transform a cyclic schema into an acyclic one. It is always possible to add a new relation (i.e a template) which contains all the attributes in the original schema. The new schema is an acyclic schema with the template as the root and all the original relations as leaves. Unfortunately, this template may be too big and contain redundant information; therefore supporting it is expensive.

example:

Consider the rule:

\[ q(X,Y) :- p(X,Z), r(Y,Z), s(W,A,B), t(B,C), u(C,A), v(Z,W). \]

The minimal qual graph for this rule is:

```
  xy
 /\  
zx --- yz --- zw ---- wab
     / \
    bc --- ca
```

A bad choice for template is \((A,B,C,W,X,Y,Z)\).
A better choice of templates is \((A,B,C)\) and \((X,Y,Z)\).

The rule \( q \) is replaced by the following program:
(1) q(X,Y) :- template2(X,Y,Z), s(W,A,B), template1(A,B,C), v(Z,W).
(2) template1(A,B,C) :- t(B,C), u(C,A).
(3) template1(X,Y,Z) :- p(X,Z), r(Y,Z).
(4) maint(template1(A,B,C)).
(5) maint(template2(X,Y,Z)).

The corresponding qual graphs for rules 1-3 are:

```
  1  2  3
  x  y  a
  y  z  b
  x  z  c

S.3. A General Algorithm for Dynamic Support

The input are qual trees created by the above algorithm. The output is a new Prolog program in which every procedure q for which there is a 'maint(q)' request is replaced by suitable procedures. The algorithm operates as follows.

For each internal node p in the tree, including the root, a new relation pp is created; initially pp is empty. Rule p is replaced by two new rules as follows. The first calls the new relation created for q (named qq) and the second calls q's children s1,...,sn. The second rule causes a new fact supported by children s1,...,sn to be moved into qq. If si is a leaf then the original relation is checked for support. Otherwise, si is an internal node and a call to a new procedure 'oks' is performed. The definition of the 'oks' procedure depends on the procedure that node si represents.

Case 1: if si is a base relation, then the first rule of 'oks' calls the corresponding new relation ss created for si. The second rule tries to locate a new fact t from relation si and if t is supported by all the children of si, then t is moved from si into ss.

Case 2: si corresponds to a supported procedure. The second rule of 'oks' causes no 'retract' from si, while moving a new fact into ss.

Case 2a: If in addition si corresponds to a supported procedure containing loose constraints then the facts of si cannot be maintained. Therefore, the first rule of 'oks' must check the validity of the ss facts by explicitly calling upon the si procedure with instantiated parameters.

example:

In the following program 'a' displays case 1 and 'y' displays case 2.
p(A,B) :- z(B,D), x(A), y(B,D).
y(M,N) :- a(N), b(L,M), c(N,P).

maint(p(_,_)).
maint(y(_,_)).

The following are the qual trees for 'p' and 'y':

```
p(A,B)         y(M,N)
  \   \        \     \        \     
x(A)  y(B,D)   a(N)  b(L,M)  c(N,P)
z(D,E)         
```

The following are the support rules for 'p' and 'y':

1. p(A,B) :- ppl(A,B).
2. p(A,B) :- x(A), oky(B,D), uniqueasserta(ppl(A,B)).
3. oky(A,B) :- yy2(A,B).
4. oky(A,B) :- y(A,B), not yy2(A,B), { z(B,_) }, uniqueasserta(yy2(A,B)).
5. y(X,Y) :- yy1(X,Y).
6. y(X,Y) :- oky(Y), b(_,X), uniqueasserta(yy1(X,Y)).
7. oky(X) :- aa1(X).
8. oky(X) :- a(X), { c(X,_) }, retract(a(X)), uniqueasserta(aa1(X)).

The first rule tries to satisfy p(A,B) by stepping through all known ppl facts which contain all the answers deduced thus far. Once all known results in ppl are exhausted, a search for new answers commences in rule 2. A fact x(a1) is drawn out of x using x(A) and oky returns a supported fact y(b1,d1) from y. The fact (a1,b1) is now created in ppl. Rule 3 indicates that oky(b1,d1) is satisfied if it has been previously derived and inserted into yy2. Rule 4 presents another way to satisfy oky(b1,d1) by locating a fact (b1,d1) in y and a supporting fact (d1,c1) in z. Again, once a support for a fact (b1,d1) in y has been derived this fact is recorded in yy2. Rules 5-8 are similar to rules 1-4 but they refer to the second-qual tree (created for y).

Insertions are simply performed into the original relations, i.e. x,z,a,b and c. If a base relation R appears in more than one supported rule then asserting facts into R implies asserting the same facts into all the rule schemas containing R. Deletions may invalidate derived facts, as illustrated in a continuation of the example.

9. detractx(x(A)) :- retract(x(A)), I, not(x(A)), delmove(ppl(A,_)).
10. detractx(z(A,B)) :- retract(z(A,B)), I, not(z(A,_)),
      delmove(yy2(_,A)).
Deletions are performed using the rule `detractx` and are reflected in both original and derived relations. Rule 9 states that the deletion of a fact (a1) from x may invalidate pipl facts of the form (_,a1). Rules 10-12 handle deletions from x, c, and b, respectively. Procedure 13 states that the deletion of a fact (a1) is first attempted from relation a1. In case the fact is in a1, it is deleted and if no other fact in a1 has the same A component, then a fact in yy1 may be invalidated. This is checked for by procedure 17, yy1 facts which are no longer supported are located and are moved from yy1. Because yy1 is the 'base' for yy2, this may cause a deletion from yy2 (procedure 18). In case (a1) is not found in relation a1, the deletion is carried out from relation a (the second alternative in 13).

In general, deleting a fact from a base relation R could affect the migration of many other facts from S to S, where S is R's father. Such migrations may affect tuples in S's father, this may propagate up to the root of the tree. We distinguish between two kinds of deletions. The first is called 'detractx' which is a 'real' deletion, the second is called 'detmove' which causes a migration of no longer supported tuples to their original base relations. The 'detmove(R)' operation is initiated by earlier 'detractx' or 'detmove' operations that caused some of the tuples in R to become unsupported. The 'detractx(R)' operation is a user request and R must be a base relation. The 'detractx' operation may also be caused by an earlier 'detmove' operation on the root of another tree which R "represents" (e.g. yy1 represents rule y within rule p).
6. FUTURE DIRECTIONS

The dynamic support mechanism previously described is not the only possible mechanism. An alternative mechanism has its roots in the static maintenance idea, i.e. maintaining all answers at all times [STZ]. The idea is to perform incremental static support, i.e. invest work in static support in accordance with the user’s demands.

Upon an insert operation, the static support mechanism may turn many facts from ‘bad’ to ‘good’ on the path from the relation into which the fact is inserted to the rest. The dynamic support derivative mechanism performs a limited number of static support steps and indicates with an ‘insertion list’ those values for which the insert operations has not yet been done (such a list is associated with each base relation).

Upon a delete operation, all (recursively) affected relations are updated; here we cannot only perform partial work as this may result in answering incorrectly. A search operation may cause a delayed insert operation to continue (such a delayed operation is indicated by the ‘insertion list’).

The new mechanism sketched above compares favorably with the one described throughout the paper, as long as the user’s requests mostly involve uninstantiated variables. Its performance weakens when the user’s requests involve instantiated variables. This is due to the ‘bottom up’ answer generation mode which is insensitive to instantiations. Scan heuristics may be used to improve performance. Another possibility is a hybrid algorithm which uses both methods.

Options for exploiting parallelism exist in both mechanisms. In the first mechanism, searches at various children of a node may often be performed in parallel. The same holds for the ‘detmove’ operation—each call, on a separate set of values, may be carried out independently. Of course, insertions may also be done in parallel. In the static derivative mechanism, effects of insertions may be performed in parallel similarly to the ‘detmove’ effects. It also offers the ability to operate in ‘background’ mode while other operations are done concurrently. Some synchronization is needed to assure that searches will wait for currently performed updates in a subtree.

Lastly, an important extension of the above mechanisms is the maintenance of recursive procedures. The basic problem is the interactions among maintained facts which were derived at different recursive depths. A maintenance mechanism based on keeping a level number per fact is currently being designed.

7. PERFORMANCE EXPERIMENTS

This section presents preliminary experimental results performed on a query defined by a single Prolog rule whose r.h. side terms all reference base relations. The following Prolog execution modes for that query are compared:

(1) The query under ‘ordinary’ Prolog execution.

```
q(A,B,C):-
a(A,B,C,E,H),
b(A,B,M),
c(A,M),
d(B,M),
e(E,F),
f(B,C).
```
(2) The query improved by braces:

\[ q(A,B,C):=\]
\[ a(A,B,C,E,H), \]
\[ \text{curly}((b(A,B,M), \text{curly}(c(A,M)), \text{curly}(d(B,M))), \text{curly}(e(E,F)), \text{curly}(f(B,C))), \]
\[ \text{curly}(X):- \text{call}(X), 1. \]

which is equivalent to

\[ q(A,B,C):-\]
\[ a(A,B,C,E,H), \]
\[ \{ b(A,B,M), \{ c(A,M), \{ d(B,M), \{ e(E,F), \{ f(B,C) \} \} \} \} \} \]

The improved program, as well as the maintained program below, are derived using the following Quad tree for \( q \):

```
    q
   / \    
  a b e f
 / \   
c d
```

(3) The maintained program for the query, assuming the above Quad tree (see fig. 1).

Each experiment is characterized by the following parameters:

(1) \text{search-ratio}: determines the ratio between \( q \)'s activations which call searches to those which perform updates. For example, a \text{search-ratio} of 15:1 means that one out of 18 activations is an update (randomly, half the updates are insert operations and half are delete operations).

(2) \text{run-extension}: the probability that upon a successful answer to a search activation another answer will be requested, e.g. a \text{run-extension} factor of 1/10 implies that the expected length of a search query is 10 (assuming enough answers exist).

(3) \text{range}: determines the domain size of base relation attributes, e.g. a range of 5 means that an attribute value must be a natural number between 1 and 5.

(4) \text{fill-up}: determines the amount of data in the database. It is the ratio between the number of tuples in each base relation and the maximum possible number of such tuples (i.e. its domain size). For example, a 40\% fill up of a 3-ary base relation, implies that the relation holds \( \frac{40}{100} \times 5^3 = 50 \) randomly chosen tuples.
(5) instantiation-ratio: determines the ratio between uninstantiated and instantiated call parameters, e.g. a ratio of 1:49 implies that the probability that a call parameter is instantiated is 0.02.

The experiments were performed with the following parameters:

**experiment 1:**
- search-ratio: 15:1
- run-length: 10
- range: 5
- fill-up: 40%
- instantiation-ratio: 1:50

**experiment 2:**
- search-ratio: 15:1
- run-length: 10
- range: 5
- fill-up: 40%
- instantiation-ratio: 1:25

**experiment 3:**
- search-ratio: 7:1
- run-length: 10
- range: 5
- fill-up: 40%
- instantiation-ratio: 1:50
\[ q(A,B,C) :- \]
\[ \text{assert}(q(A,B,C)). \]
\[ q'(A,B,C) :- \]
\[ \text{assert}(q'(A,B,C)). \]

\[ \text{detmove}(q(A,B,C)) :- \]
\[ \text{retract}(q(A,B,C)). \]

\[ \text{detmove}(q'(A,B,C)) :- \]
\[ \text{retract}(q'(A,B,C)). \]

\[ \text{detach}(d(A,B)) :- \]
\[ \text{retract}(d(A,B)). \]

\[ \text{detach}(e(A,B)) :- \]
\[ \text{retract}(e(A,B)). \]

\[ \text{detach}(f(A,B)) :- \]
\[ \text{retract}(f(A,B)). \]

\[ \text{detach}(g(A,B)) :- \]
\[ \text{retract}(g(A,B)). \]

\[ \text{insert}(A) :- \]
\[ \text{assert}(A). \]

\[ \text{uniqueassert}(A) :- \]
\[ (\text{not } A, \text{assert}(A)). \]

\[ \text{Figure 1: MAINTAINED PROGRAM} \]
The following graph summarizes the measured time elapses on the VAX/780 running C-Prolog 1.4a under UNIX. It is derived from the statistics C-Prolog system call. These results are by no means conclusive. In the experiments performed, the dynamically supported version eventually performed best. Considering maintenance is done in Prolog (and not at a lower level), the results appear promising.
REFERENCES


