Consiglio Nazionale delle Ricerche

ISTITUTO DI ELABORAZIONE DELLA INFORMAZIONE

PISA

Recursive Query Optimization and Partial Evaluation

P. Asirelli, P. Inverardi, V. Raffaelli

Nota Interna B4-09
Marzo 1990

Recursive Query Optimization and Partial Evaluation

P. Asirelli ♦, P. Inverardi ♦, V. Raffaelli ♥

♦ Istituto di Elaborazione dell'Informazione CNR,

via S. Maria n.46, I-56100 PISA

♥ Dipartimento di Informatica - Università di Pisa

Abstract

The paper presents an approach to recursive query optimization based upon the integration of partial evaluation and already existing rule transformation based, optimization methods such as Magic Set, Minimagic and Counting. The basic idea is to partially evaluate a logic program with respect to a query (goal), thus eliminating unnecessary intermediate (IDB) predicates and then further optimize the program, with respect to evaluation, by using one of the mentioned optimization methods. The advantage of the proposal is twofold: on one hand to give the user freedom of using all the power of a logical language to define the query program and, on the other, to keep efficiency into reasonable range.

Keywords: logic programming, partial evaluation, deductive databases, recursive query optimization.

1. Introduction

Here we present an approach to recursive query optimization based upon the integration of partial evaluation and already existing rule transformation based, optimization methods.

The idea relies on the evidence that partially evaluating a logic program with respect to a query (goal) eliminates unnecessary intermediate (IDB) predicates. The resulting program computes the same set of answers to the query but it has a reduced *deductional* complexity. The partially evaluated program can thus be considered as minimal with respect to the amount of predicates and recursion needed to answer the query. It can then be further optimized, with respect to evaluation, by using one of the already existing optimization methods.

It is well known that the main advantage in using a logical query language is its declarativeness which provides increased expressive power. In particular, the use of many derived predicates can be very useful. It helps in defining the IDB, allowing to introduce many intermediate predicates so that the program can be incrementally, and more easily, written. It helps in defining the single query-goal, since predicates with fewer arguments allow the user

to express the query-goal in a more simple and direct way. That is, it should be possible to freely use also unnessary predicates (derivate predicates) without being concerned of the implication of such redundancy on the query evaluation/optimization process.
On the other hand, in the database context, efficiency considerations are a must.

Many new evaluation strategies, or optimization of already known methods, have been proposed that try to increase the query evaluation efficiency [Bancilhon&Ramakrishnan'86]. Some of these strategies are top-down but set-oriented and many others are bottom-up. In this second group, we can find the rewriting rules methods like Magic Set [Bancilhon et al.'86], Minimagic [Sacca&Zaniolo'87], Counting [Bancilhon et al.'86], Alexander [Rohmer et al.'86] (and various generalizations of them) that optimize the Naive/SemiNaive Evaluation, a sort of "pure" bottom-up strategy [Bancilhon'86].

These methods are based on techniques that rewrite the original program in a new equivalent one, with at least the same deductional complexity of the original one, but will be more efficiently (bottom up) evaluated. Unfortunately, the more complex the original program is, the more complex the translation and the evaluation processes are.

We want to find a compromise between efficiency and easiness of definition and this is where partial evaluation is brought in. The idea is to partially evaluate the program with respect to the query and then apply the more convenient optimization method. As it will become clearer in the following, the approach is even more effective when partially evaluating the program with respect to uninstantiated queries; this suggests a compilative approach to the query evaluation process.

The paper is organized as follows: In section 2 an overview of Partial Evaluation and of the rewriting rule methods is given together with an example that will be used throughout the paper. Section 3 discuss the integration of Partial Evaluation and the considered methods; section 4 presents a characterization of the suitable class of queries ; section 5 concludes.

2. The considered approaches

2.1 Partial Evaluation

Partial Evaluation is a program transformation technique that, given a program P and a set of values, produces a new program P', equivalent to P, that is more specialized of P with respect to the given values [Ershov'77].

In the Logic Programming framework the program to be transformed is a logic (definite or general) program and the transformation is performed with respect to a goal, that may or may not, be partially instantiated. The resulting program is again a logic program (a set of clauses) which is specialized with respect to the given goal [Komorowski'81].

The various partial evaluation procedures that can be found in the literature [Venken'84, Levi&Sardu'88, Fanti et al.'88, Sakama&Itoh'88, etc.] are essentially based on top-down

evaluation strategies such as SLD-Resolution [Lloyd'87]. They must have three basic properties:

- termination of the transformation process;
- semantic equivalence of the transformed program with respect to every possible further instantiation of the given goal;
- more efficient top-down evaluation of the transformed program.

Within the database area, Partial Evaluation is introduced as a transformation and specialization of the query to the IDB (Intensional Data Base) [Venken'84, Venken&Demoen'88, Sakama&Itoh'88, etc.]. Every call to the EDB (Extensional Data Base) is "delayed", i.e. it cannot be unfolded.

Given the intrinsic top-down nature of the transformation, many authors consider the partial evaluation technique as part of an evaluation strategy that is completely top-down (extention of Prolog meta-interpreters).

In [Sakama&Itoh'88] bottom-up evaluation is considered. After transforming the given program by partially evaluating it with respect to the given query, they rewrite the obtained program into expressions of the relational algebra. Then the query is evaluated bottom-up. This approach combines the two evaluation strategies, thus allowing for better efficiency, given that the relational algebra expressions are computed starting from a semplified program. On the other hand, the inefficiencies that lead to adopt methods such as Magic Set, and then, Minimagic, still remain.

In the following, we will partially evaluate the sample program with respect to a generic query. The partial evaluation procedure we apply is the one used in [Sakama&Itoh'88]. For sake of simplicity we are using this procedure instead of more complex ones [Levi&Sardu'88, Fanti et al.'88], since, during the partial evaluation process, it does not take into account the binding information of variables, that is, the constant information that is actually present in the query, and, from our point of view, this is uninfluent given that it will be considered when applying Minimagic to the partially evaluated program.

Partial Evaluation Procedure [Sakama&Itoh'88]:

The HCT (Horn Clause Transformation) procedure is as follows:

(1) Predicates defined in the edb are extensional predicates.

(2) Predicates which appear twice at first during the unfolding of a given query are recursive predicates.

(3) Select all clauses in the idb whose head predicate is the same as the one in a given query or recursive predicates, then unfold their bodies until they contain only extensional predicates or recursive predicates.

Let us now present our sample example, "Francois example", that we have taken from [Saccà&Zaniolo'87].

Example :

P: r_0: P(x, y):-B1(x, w, x1), Q(x1, y), B2(w, x2), Q(x2, y), B3(y, z).

r_1: Q(x, y):-B4(x, z), P(z, y).

r_2: P(x, y):-B5(x, y).

query: P(a, y)?

Step 1-2

P(x, y):-B5(x, y). (stop)

P(x, y):-:-B1(x, w, x1), B4(x1, z1), P(z1, y), B2(w, x2), B4(x2, z2), P(z2, y), B3(y, z). (stop)

Note that the Bi, (i=1..5) are extensional while P is recursive; here terminates step 2.

Step 3 Unfolding:

r_0: P(x, y):-B1(x, w, x1)$_{ext}$, Q(x1, y), B2(w, x2)$_{ext}$, Q(x2, y), B3(y, z)$_{ext}$.

unfolding Q:

:-B1(x, w, x1)$_{ext}$, B4(x1, z1)$_{ext}$, P(z1, y)$_{ric}$, B2(w, x2)$_{ext}$,B4(x2, z2)$_{ext}$, P(z2, y)$_{ric}$, B3(y, z)$_{ext}$. (stop)

r_2: P(x, y):-B5(x, y)$_{ext}$. (stop)

Note that in this case the third step is a repetition of the second one, since the only recursive predicate coincides with the query predicate already unfolded in the second step. The final program is:

P^{PE} :

r_1*: P(x, y):-B5(x, y).

r_2*: P(x, y):-B1(x, w, x1), B4(x1, z1), P(z1, y), B2(w, x2), B4(x2, z2), P(z2, y), B3(y, z).

query: P(a, y)?

It is easy to see that the resulting program is simpler since one clause has been eliminated together with the predicate Q; the other effect is that some more solved base predicates appear in the body of r_2*; the latter may be a problem if one wants to apply the Magic Set method in order to perform an efficient bottom-up evaluation of the program. In the original program, in fact, the base predicate B4 appeared only in the body of the clause whose head is Q. We will see in section 3 how to deal with such a problem.

2.2 Magic Set, Minimagic and Counting

Magic Set, Minimagic and Counting are methods that essentially rewrite the original program. The resulting program is then evaluated by a bottom-up strategy such as Naive/Seminaive Evaluation [Bancilhon'86]. These methods produce a new set of clauses that

are equivalent to the original ones with respect to the given query, i. e. the set of computed answers is the same. Furthermore, the new clauses are more suitable for an efficient bottom-up evaluation, i. e. they prevent the generation of a number of irrelevant facts in computing the answers. These methods are applied to partially instantiated queries and their philosophy is to take into account only those facts that are relevant to that instantiation.

Magic Set method, presented in [Bancilhon et al.'86], applies to Horn clauses without function symbols. Many generalizations and/or specializations (see [Beeri&Ramakrishnan '87, Saccà&Zaniolo'86a, Saccà&Zaniolo '87]) have been proposed in comparison with other evaluation strategies.

Magic Set is not suitable when clauses have a lot of solved base predicates in their bodies. In this case, in fact, many redundant accesses to the EDB are necessary. Another strategy, called Minimagic, has been proposed [Saccà&Zaniolo'87]. With respect to the transformation performed by Magic Set, Minimagic transformed clauses fold up several calls to the same base predicate, so that the number of accesses to the EDB is reduced and then, according to Magic Set philosophy, constrained (by the instantiation of the query) as much as possible. The transformation is performed by inserting new predicates whose role is to collect the already computed values.

A similar use of new derived predicates has been adopted in the Generalized Counting method, which is an extension of Counting [Bancilhon et al.'86]. With respect to magic strategies, Generalized Counting inserts, in each derived predicate, three new integer arguments, respectively denoting:

i) the recursive call level;

ii) an index to the recursive rule;

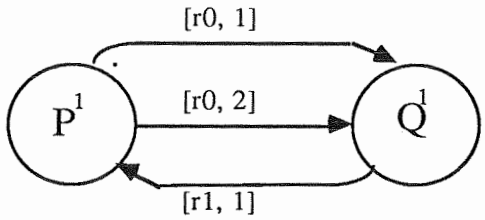
iii) an index to the recursive predicate used in the body of the rule.

In the following Magic Set, Minimagic and Generalized Counting are applied to our sample example:

Magic Set

PMg:

binding graph:



magic.P^1(a).

magic.Q^1(x1):-B1(x, w, x1), B2(w, x2), magic.P^1(x).

magic.Q^1(x2):-B1(x, w, x1), B2(w, x2), magic.P^1(x).

magic.P^1(z):-B4(x, z), magic.Q^1(x).

P^1(x, y):-magic.P^1(x), B1(x, w, x1), Q^1(x1, y), B2(w, x2), Q^1(x2, y), B3(y, z).

Q^1(x, y):- magic.Q^1(x), B4(x, z), P^1(z, y).

P^1(x, y):-magic.P^1(x), B5(x, y).

query: P^1(a, y)?

Minimagic

$_P$Minimg :

binding graph: the same as Magic Set.

Supplementary variables V_{sp} :

V_H={x, y}, V_U={y, z}, V_C={x1, y, x2}, V_B={x, w, x1, x2}

V_{sp}={x, x1, x2}.

minimagic.P^1(a).

supmagic.$P^1.r_0$(x, x1, x2):-minimagic.P^1(x), B1(x, w, x1), B2(w, x2).

supmagic.$Q^1.r_1$(x, z):-minimagic.Q^1(x), B4(x, z).

minimagic.Q^1(x1):- supmagic.$P^1.r_0$(x, x1, x2).

minimagic.Q^1(x2):- supmagic.$P^1.r_0$(x, x1, x2).

minimagic.P^1(x):- supmagic.$Q^1.r_1$(x, z).

P^1(x, y):- supmagic.$P^1.r_0$(x, x1, x2), Q^1(x1, y), Q^1(x2, y), B3(y, z).

Q^1(x, y):- supmagic.$Q^1.r_1$(x, z), P^1(z, y).

P^1(x, y):-minimagic.P^1(x), B5(x, y).

query: P^1(a, y)?

Counting

$_P$Cnt.

binding graph: the same as Magic Set.

Supplementary variables: none.

cnt.P^1 (0, 0, 0, a).

cnt.Q^1 (j+1, 2*k+0, 2*h+1, x_1):- cnt.P^1 (j, k, h, x), B1(x, w, x_1), B2(w, x_2).
(da [r_0 , 1])

cnt.Q^1 (j+1, 2*k+0, 2*h+2, x_2):- cnt.P^1 (j, k, h, x), B1(x, w, x_1), B2(w, x_2).
(da [r_0 , 2])

cnt.P^1 (j+1, 2*k+1, 1*h+1, z):- cnt.Q^1(j, k, h, x), B4(x, z).
(da [r_1, 1])

P^1 (j-1, (k-0)/2, h/2, y):- Q^1 (j, k, h, y), Q^1 (j, k, h+1, y), B3(y, z).

Q^1 (j-1, (k-1)/2, h/1, y):- P^1(j, k, h, y).

P^1 (j, k, h, y):-cnt.P^1 (j, k, h, x), B5(x, y).

query : P^1 (0, 0, 0, y) ?

3. The integration

In this section, we discuss on the basis of our sample example, the effect of introducing partial evaluation and Minimagic/Counting. We will not consider the combination with magic, because, in this case, partial evaluation results in a program, where the bodies of clauses contain solved base predicates, more than the original one. This is exactly the case where one would apply Minimagic or Generalized Counting.

As we have seen, the two transformation techniques, Partial Evaluation and Minimagic/Generalized Counting, are in a sense one opposite to the other. In fact, while Partial Evaluation unfolds clauses so that base predicates are put together into bodies as much as possible, and some IDB predicates are eliminated, Minimagic and Generalized Counting go the opposite direction, that is, they try to fold up, in the same clause, calls to the same EDB predicate, by inserting new predicates.

Thus it may sound odd to propose the integration of Partial Evaluation with these two rewriting strategies, since it seems that Minimagic and Generalized Counting destroy what the Partial Evaluation has done.

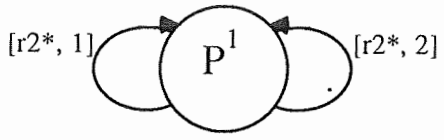
On the contrary, it is worthwhile noticing that, Minimagic and Generalized Counting trasform clauses by inserting new IDB predicates that are ad-hoc for the particular instantiated query. Thus, if they are applied to a complex program, i.e. with intermediate (IDB) predicates, they may produce a lot of IDB minimagic/counting and transformed intermediate predicates. Then, the effect of combining Partial Evaluation and Minimagic/Generalized Counting is that of reducing the negative side effects of both, thus producing a more efficient final program.

In the following we show how our method works on the sample example and compare the resulting programs with the programs obtained by simply applying Minimagic and Generalized Counting.

Partial Evaluation + Minimagic

(P^{PE})Minimg :

binding graph:



Supplementary variables V_{sp}:

V_H={x, y}, V_U={y, z}, V_C={z1, y, z2}, V_B={x, w, x1, z1, x2, z2}

V_{sp}={x, z1, z2}.

minimagic.P^1(a).

supmagic.$P^1.r_{2*}$ (x, z1, z2):-minimagic.P^1(x), B1(x, w, x1), B4(x1, z1), B2(w, x2), B4(x2, z2).

minimagic.P^1(z1):- supmagic.$P^1.r_{2*}$ (x, z1, z2).

minimagic.P^1(z2):- supmagic.$P^1.r_{2*}$ (x, z1, z2).

P^1(x, y):- supmagic.$P^1.r_{2*}$ (x, z1, z2), P^1(z1, y), P^1(z2, y), B3(y, z).

P^1(x, y):-minimagic.P^1(x), B5(x, y).

query: P^1(a, y)?

Partial Evaluation + Generalized Counting

$(P^{PE})^{Cnt}$:

binding graph: the same as Partial Evaluation + Minimagic.

Supplementary variables: none.

cnt.P^1(0, 0, a).

cnt.P^1(j+1, 2*h+1, z1):- cnt.P^1(j, h, x), B1(x, w, x1), B4(x1, z1), B2(w, x2), B4(x2, z2).

cnt.P^1(j+1, 2*h+2, z2):- cnt.P^1(j, h, x), B1(x, w, x1), B4(x1, z1), B2(w, x2), B4(x2, z2).

P^1(j-1, h/2, y):- P^1(j, h+1, y), P^1(j, h+2, y), B3(y, z).

P^1(j, h, y):- cnt.P^1(j, h, x), B5(x, y).

query: P^1(0, 0, y)?

Note that the resulting programs, $(P^{PE})^{Minimg}$ and $(P^{PE})^{Cnt}$, are simpler and more efficiently bottom-up evaluable than P^{Minimg} and P^{Cnt} (section 2.2). In fact, $(P^{PE})^{Minimg}$ and $(P^{PE})^{Cnt}$ do not contain all the derived predicates related to the intermediate predicate Q (the one eliminated from the Partial Evaluation): minimagic.Q^1, supmagic.$r_1.Q^1$ and Q^1 that are in P^{Minimg} and cnt.Q^1 and Q^1 in P^{Cnt}. Thus $(P^{PE})^{Minimg}$ and P^{Cnt} do not contain the above *unnecessary* predicates and, obviously, the rules that define them. Note that, in comparison with P^{Minimg} (P^{Cnt}, respectively), the evaluation of base predicates may be considered equivalent, e.g. the base predicate B4 that appears twice in the body of the second clause of $(P^{PE})^{Minimg}$ (and P^{Cnt}) computes the same set of values than the evaluation of B4 that, although appearing in a single body of a clause in P^{Minimg} (and P^{Cnt}), is called with two sets of values originated by the two rules for minimagic.Q^1 (and cnt.Q^1).

Furthermore, due to the simplification achieved by Partial Evaluation, there is a single recursive rule in P^{PE}. So, the second index in all the derived predicates of $(P^{PE})^{Cnt}$ remains constant and can be eliminated.

Finally, it is worth noticing that due to the correctness of both Partial Evaluation and the

rewriting methods, the correctness of the combined method is straightforwardly achieved.

4 A characterization od suitable partially evaluable queries

In this section we single out and describe the kind of query which is particularly suitable for the optimization technique introduced above.

One of the most important consequences of such optimization technique is the reduction of the number of derived predicates (magic or not) still remaining in the IDB once it has been transformed. This becomes more evident when the Partial Evaluation phase finds out - and deletes - a lot of "intermediate" predicates. Consequently, we will characterize the nature of "intermediate" predicates deleted by the HCT procedure.

According to the HCT procedure, the group of recursive predicates ("not intermediate") consists of the query predicate together with those predicates that appear twice during the unfolding of the given query (step (2)). We will use a simple kind of graph in order to make it clear what happens in step (2). Note that, since neither predicate arguments, nor the different names of base predicates in the rules we are unfolding, do alter the unfolding itself, we can use an oriented graph where nodes only represent derived predicate names. Starting from query predicates, we draw an arc for any pair of nodes (P, Q) providing that there is a rule in the IDB with P in the head and Q in the body.

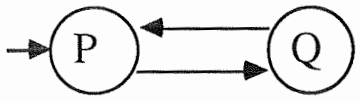
Example

Although we have already introduced the "Francois Example", its rules will be repeated here in the graph-oriented simplified way: predicate arguments will be ignored and the *B* symbol will stand for any database predicate.

P : P:- B, Q, B, Q, B.
Q:- B, P.
P:- B.

query-goal: P?

graph:



According to the HCT procedure, P is recursive (because it is the query predicate), while Q is deleted.

The same kind of analysis can be performed on the graph (instead of on the program) to determine which nodes (predicates) are eliminable, i.e. can be deleted.
Generally, a predicate R is considered as recursive if it is the root (query-predicate) or if there is a "reconnecting" arc (T, R), where R is a node belonging to an *elementary* path (i. e. without repeated nodes) going from the query-predicate node to T.

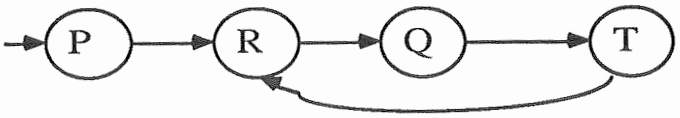
Let us now consider two more general queries (in the simplified form) and show their graphs.

Example

P1: P:- R, B.
R:- Q, B.
Q:- T, B.
T:- R, B.
T:- B.

query-goal: P?

graph:



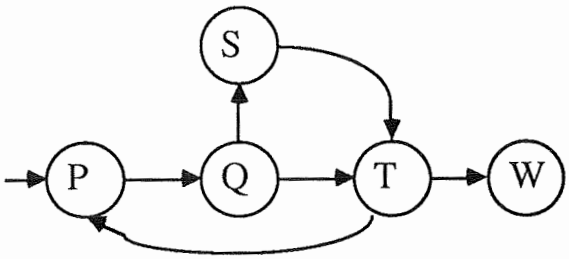
Recursive predicates: P and R. Eliminable predicates: Q and T.

Example

P2: P:- Q, B.
Q:- S, T, B.
T:- W, P, B.
S:- T, B.
P:- B.
S:- B
W:- B.

query-goal: P?.

graph:

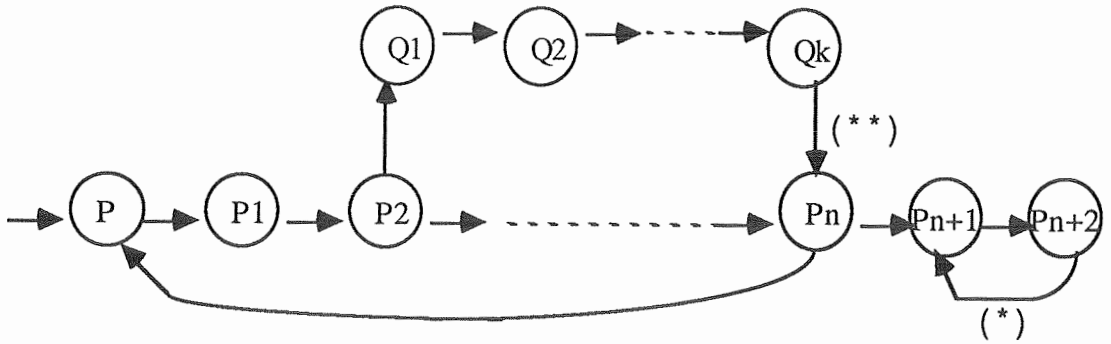


P is the only recursive predicate. The others are all eliminable. In fact, the only reconnecting arc is (T, P). Note that (S, T) is not a reconnecting arc, since T does not belong to any elementary path from P to S.

The intermediate predicates we are going to delete are those that have no entering reconnecting arcs. So, any suitable query for our optimization technique, is supposed to have a graph with one or more paths, each of them starting from the query-predicate node, with n arcs, n>>1, and such that the only reconnecting arc in it enters the query-predicate node. All predicates in such paths are eliminable, except the query-predicate, which can never be deleted.

Fig.1 shows a typical case. Predicates P_1, ..., P_n, P_{n+2} in the graph are eliminable, such as Q_1, ..., Q_k. The only derived predicates that cannot be deleted are: P (the query predicate) and P_{n+1}, because of the reconnecting arc (*). On the other hand, the arc (**) is not a reconnecting arc, because P_n does not belong to an elementary path going from P to Q_k. So,

P_n is also eliminable.



-Fig.1-

For a given IDB, it is possible to make a static analysis for each derived predicate (i.e. for each possible query-predicate) of the eliminable ones. In fact, when changing the query-predicate, the kind of each predicate eliminable, or not, may change too. Differently, the presence of constants in the query-goal does not alter the classification at all. Thus, in a given IDB, we can evaluate the opportunity of applying the optimization for every predicate and keep this information. Then, once the query-goal (with or without constants) has been fixed, it is possible to decide whether it is convenient to execute the Partial Evaluation phase, or else to immediately apply other optimization methods.

Let us now have a query having *n* intermediate predicates. Then, we can try to extimate how the query would be simplified by applying the HCT followed by Magic Set. The same considerations, straightforwardly apply to the other methods.

Considering the number of predicates and rules in the IDB, resulting from the application of Magic Set to the original IDB, we want to extimate the number of rules and predicates that would not be in the IDB once tranformed by HCT and Magic Set.

In order to simplify the extimation, we assume that there exsists at least one recursive rule defining each of the n predicates. By *recursive rule*, we mean a rule with a recursive predicate in its head, R, and one or more predicates in its body that are part of the same *recursive clique* of R [Saccà&Zaniolo'87].

– number of eliminated *magic predicates*: k, $n \leq k \leq \Sigma_{i=1..n}(2^{narg_i} - 1)$. In fact, there is a magic predicate for each node P^T of the original query binding graph. For each derived predicate P there are as many nodes in binding graph as different adornments originated by bound variables of the query-goal. The number of different adornments is, at most, $(2^{narg_P} - 1)$, where $narg_P$ is the number of arguments of P (we have excluded the "all variables free" adornment).

– number of eliminated *adorned derived predicates*: k as above. In fact, there is an adorned derived predicate P^T for each node of the original query binding graph.

– number of *magic rules* defining the k eliminated magic predicates: k_1

$n \leq k \leq k_1 \leq \Sigma_{i=1..n}[(2^{narg_i} - 1)*m_i]$ where m_i is the number of eliminable i-predicate occurrences in the original IDB. In fact, k_1 is the number of the arcs (R^S, P^T) in the original query binding graph, where P is any of the n eliminable predicates. Each arc is identified both by the pair of nodes, R^S and P^T, and by a label $[r_i, j]$, where r_i is an index to a recursive rule defining R and j is an index to the P occurrence in the body of the rule. So, we must take into account both the number of eliminated predicates occurrences in recursive rules and the number of their different adornments.

– number of *modified recursive rules* defining the k eliminated adorned predicates: k_2, $n \leq k \leq k_2 \leq \Sigma_{i=1..n}[(2^{narg_i} - 1)*nric_i]$. Having a node R^S in the original query binding graph, where R is an eliminable predicate, there exists an eliminable modified recursive rule for each bundle of arcs labelled with the same rule r_i. Thus, k_2 is greater than or equal to k (the number of such nodes). Note that, as it has been assumed, there exists at least one recursive rule for each of the n eliminable predicates. Then, for each of the k nodes R^S there exists at least one arc starting from that node and, consequently, one eliminable modified recursive rule.

– number of *modified exit rules* for the k eliminated adorned predicates: k_3, $\Sigma_{i=1..n}(nexit_i) \leq k_3 \leq \Sigma_{i=1..n}[(2^{narg_i} - 1)*nexit_i]$, where $nexit_i$ is the number of the exit rules defining the eliminable i-predicate in the original IDB. For each of the $nexit_i$ rule in the original IDB, and for each different adornment of its head predicate, there exists a new modified exit rule.

5. Concluding Remarks

The idea of combining Partial Evaluation and optimization techniques like Magic/Minimagig/ Counting, comes from the attempt to compare, on one side, tools and methods developed in the Logic Programming area with, on the other side, logical query optmization techniques that have been entirely developed in the database field. Thus, when considering Magic/Minimagic/Counting it can be realized that part of the work they perform, besides the main action of reducing the IDB accesses, is a sort of trivial partial evaluation of the program, that eliminates all predicates that are not involved in the query.

When approaching deductive databases in a logic programming environment, it seems us more convenient to apply LP tools and methods, as much as possible, and use only the original contribution of such query optimization techniques. Our proposal follows this approach and we believe that the advantage of the resulting technique is twofold: On one side to give the user freedom of using all the power of a logical language to define the query program and, on the other, to keep efficiency into reasonable range