# Functional Dependencies in Logic Programs

*Alberto O. Mendelzon*

Computer Systems Research Institute
University of Toronto
Toronto, Canada M5S 1A4

**Abstract**

When logic programming is used for database access, there is a need to improve the backtracking behaviour of the interpreter. Rather than putting on the programmer the onus of using extra-logical operators such as *cut* to improve performance, we show that some uses of the cut can be automated by inferring them from functional dependencies. This requires some knowledge of which variables are guaranteed to be bound at query execution time; we give a method for deriving such information using data flow analysis.

## 1. Introduction

There has been much recent interest in the use of logic programming languages for database access [e.g. JCV,D,SW,U]. Some of this work deals with the integration of existing or future database systems with the logic programming component; another approach, and the one we pursue here, is simply to take a widely available logic programming system, such as Prolog [CM], and try to use it for database access without any further modification. It is clear that there are many limitations in such an approach, as discussed for example in [BJ]; but there are also advantages. For applications where the databases are small, or for quick prototyping of larger applications, or for interactive queries where data access is localized and the database can be held in virtual memory, the use of a widely available language such as Prolog makes for systems which are easy to build and maintain and easy to port.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

However, as Warren reported [W], the rigid backtracking behaviour of Prolog is inadequate even for relatively undemanding database applications. The logic programming community is well aware of this problem and several proposals for *selective backtracking* exist [BP,PP,KM]. In practice, Prolog systems provide a control primitive called *cut* that gives the programmer some say in how backtracking is handled. But the use of the cut or other extra-logical control operators makes programs less declarative and more procedural, partially defeating the goals of logic programming. In this paper we propose to use declarative information provided by the programmer in the form of *functional dependencies* to automate the insertion of cuts. While this does not cover all uses of the cut, we think it is a step in the right direction. Our approach differs from the work on selective backtracking cited above in that it requires no changes to the Prolog interpreter and can be implemented in a preprocessor.

For example, suppose we have a database with the following predicates:

*capital(A,C) : the capital of country A is city C*
*hq(X,C): company X is headquartered at city C*
*sales(X,S): company X sells S million dollars a year*

We are interested in those countries whose capital cities are headquarters for some company that has more than 100 million a year sales:

*big(C) ← capital(C,X), hq(Y,X), sales(Y,S), S> 100.*

Now suppose we try to evaluate the query

*big(usa)*

with the usual Prolog backtracking mechanism. First, the *capital* predicate will be evaluated, binding $X$ to "Washington". Suppose there is only one company with headquarters in Washington, and its sales are only 1 million. When $S > 100$ fails, the Prolog interpreter will backtrack to try to find another sales figure for this company; when in turn this fails, it will backtrack again and, failing to find another company with headquarters in Washington, it will try to find another capital of the U.S.A. Many versions of Prolog provide a control operator called *cut*, denoted "!", that lets the programmer avoid these unnecessary attempts to re-satisfy predicates that cannot be re-satisfied. For example, we could write

$big(C) \leftarrow \alpha(C,X), hq(Y,X), sales(Y,S), S > 100.$
$\alpha(C,X) \leftarrow capital(C,X),!.$

The effect of the cut in the $\alpha$ clause is to prevent the interpreter from trying to re-satisfy $\alpha$ if any goal to the right of it in the *big* clause fails. We can apply the same transformation to the *sales* predicate and obtain

$big(C) \leftarrow \alpha(C,X),hq(Y,X),\beta(Y,S), S > 100.$
$\alpha(C,X) \leftarrow capital(C,X),!.$
$\beta(Y,S) \leftarrow sales(Y,S),!.$

However, there is still one serious source of inefficiency in the backtracking behaviour of this program. Suppose there are 100 companies with sales over 100 million headquartered in Paris; then the query

*big(france),big(usa)*

will re-satisfy *big(france)* 100 times before failing. Again, the cut is helpful in avoiding this as follows:

$big(C) \leftarrow \alpha(C,X),\gamma(Y,X,S).$
$\gamma(Y,X,S) \leftarrow hq(Y,X),\beta(Y,S), S > 100, !.$

with $\alpha$ and $\beta$ defined as before. The cut at the end of the $\gamma$ clause means that, once a country has been proven *big*, the interpreter will not try to re-prove this in a different way.

What started as a clean logical definition of *big* countries has become, through the introduction of cuts, a complex program that can only be understood procedurally. In this paper we propose to automate this process by using *functional dependency* statements to infer where cuts can be inserted. For example, the introduction of $\alpha$ and $\beta$ is allowed by the fact that in *capital(C,X)*, $C$ functionally determines $X$, and in *sales(Y,S)*, $Y$ functionally determines $S$. We would like the program to be written as follows:

$capital(C,X)\text{: } C \rightarrow X$
$sales(Y,S)\text{: } Y \rightarrow S$
$big(C) \leftarrow capital(C,X),hq(Y,X),sales(Y,S),S > 100.$

and let the system produce the improved version. The method we present handles the introduction of $\alpha$ and $\beta$, but not of $\gamma$; this is actually the problem of decomposition into independent subqueries that Warren solved in [W].

Note that, even though the functional dependency $hq(Y,X)\text{:}Y \rightarrow X$ holds, we have not used it. The reason is that *hq(Y,X)* is invoked with the $Y$ variable free and the $X$ variable bound; if *sales(Y,S)* fails, it is in fact necessary to backtrack and find another binding for $Y$, that is, another company with headquarters in city $X$. In fact, the introduction of $\alpha$ and $\beta$ preserves the meaning of the program only if we are using *big* as a test, rather than as a generator. For example, if the query is

$$big(W)$$

where $W$ is a variable, and it happens that the first country that appears in the *capital* predicate is usa, then the answer will be empty. Thus the application of our method requires some knowledge of which variables are guaranteed to be bound on invocation of each predicate. We use data flow analysis techniques to obtain this knowledge statically.

In the next section, we formalize the concepts above. In Section 3, we present the cut-insertion process and show its correctness. In Section 4, we give a method for computing the information on binding of variables that is required by the cut-inserter.

## 2. Definitions

### 2.1. Logic Programming

A *program* **P** is a sequence of Horn clauses $C_1, \ldots, C_n$. Each $C_i$ is of the form

$$C_i : A(x_1, \ldots, x_n) \leftarrow B_1(y_{11}, \ldots, y_{1k_1}), \ldots, B_m(y_{m1}, \ldots, y_{mk_m}).$$

where $k \geq 0$, and each $x_i$ is either a variable or a constant. The left hand side of the clause is also called its *head*. We assume a function-free language. We also assume that every $x_i$ appears on the right hand side or it is a constant. In our examples, predicates that do not appear in the head of any clause are assumed to be *database predicates* wholly defined by ground clauses. A program can be viewed as a first-order theory by interpreting "$\leftarrow$" as logical implication and universally quantifying all variables.

A *query* is an atomic formula

$$\phi : G(z_1, \ldots, z_p)$$

where each $z_i$ is either a variable or a constant. The *meaning* of **P** on query $\phi$ is

$$\mathbf{P}(\phi) = \{ < c_1, \ldots, c_p > \mid c_i \text{ is a constant } \forall i \text{ and } \mathbf{P} \models G(c_1, \ldots, c_p) \}.$$

## 2.2. Prolog

The *Prolog* programming language [CM] provides a practical implementation of logic programming. A Prolog interpreter attempts to compute the meaning of program **P** on query $\phi$ by applying the following procedure. Construct a tree whose nodes are sequences of atomic formulae, called *goals*. The root of the tree is $\phi$. Suppose a node contains the goal $< L_1, \ldots, L_n >$, and there is a clause

$$C_j : A \leftarrow B_1, \ldots, B_m$$

in **P** such that $A$ "matches" $L_1$, or, more formally, $A$ can be unified with $L_1$ using unifier $\theta$. Then the node has a child labelled with

$$< \theta(B_1), \ldots, \theta(B_m), \theta(L_2), \ldots, \theta(L_n) > .$$

The children of a node are ordered according to the value of $j$ in $C_j$. There are three kinds of branches in the tree: *success* branches are those that end in a leaf labelled with the empty goal; *failure* branches end in a leaf whose leftmost literal cannot be unified with the left hand side of any clause; and *infinite* branches do not terminate.

The Prolog interpreter searches this tree in depth-first order; whenever a leaf at the end of a successful branch is reached, the composition of all the substitutions $\theta$ applied in the path from the root to this leaf is applied to the original query and the result is output. We will denote the result of this computation by

$$Pr(\mathbf{P}, \phi).$$

It can be shown that the above procedure is sound, that is, $Pr(\mathbf{P}, \phi) \subset \mathbf{P}(\phi)$ for function-free programs [L]. However, it is not complete; that is, there are cases where the search procedure does not terminate and fails to find successful leaves that exist in the tree.

## 2.3. The cut operator

The *cut* operator, denoted by "!", may appear anywhere on the right hand side of a clause. Cut is treated as the constant **true**, but it also has the following side effect on the depth-first search procedure. Suppose a failure branch is found in a subtree whose root node $r$ starts with a cut. Let $v$ be the lowest ancestor of $r$ that does not contain a cut; then the whole subtree rooted at $v$ is considered visited by the search procedure and it is not searched further. Note that, if the part of the tree that is not searched does not contain a success branch, the cut does not make the interpreter miss any answers that would be found without the cut. However, if the omitted part of the tree contained some infinite branches, the presence of the cut might cause some additional sound answers to be found, because the interpreter will avoid those infinite paths.

## 2.4. Functional Dependencies

A *functional dependency* is a statement of the form $A(x_1, \ldots, x_n): x_{i_1}, \ldots, x_{i_p} \rightarrow x_j$. The appearance of this functional dependency in program **P** means that if two tuples of predicate $A$ agree on the $i_k$'th arguments, $1 \leq k \leq p$, then they also agree on the $j$th argument. In all that follows we assume our programs and databases satisfy the given set of functional dependencies; questions of how to enforce this constraint are outside the scope of this paper.

## 2.5. Binding Rules

A *binding rule* $b$ for program **P** is a statement of the form

$$A : i_1, \ldots, i_k.$$

where $A$ is an $n$-ary predicate that appears on the left hand side of some clause of **P**, and $k \leq n$. We say that **P** runs *under* $b$ if in every execution of **P**, predicate $A$ is invoked with the $i_1$'th, ..., $i_k$'th arguments bound to constants. An *input rule* $r$ for program **P** is a statement of the same form as a binding rule. We say that **P** runs under input rule $r$ if whenever **P** is run with goal $A(x_1, \ldots, x_n)$, all the arguments $x_{i_1}$, ..., $x_{i_k}$ are constants.

An input rule is a constraint provided by the programmer; it gives the system some information on how certain predicates will be used as goals. Given some input rules, it is possible to infer from them binding rules. Consider the following example

EXAMPLE 1: Let **P** be the program below, that computes the transitive closure $T$ of a binary relation $E$:

$C_1$: $T(X,Y) \leftarrow E(X,Y)$.
$C_2$: $T(X,Y) \leftarrow E(X,Z), T(Z,Y)$.

Suppose we are given the input rule

$T$:1

that is, we are told that this program will be used to compute all nodes reachable from a given fixed node. Since $Z$ will always be bound to some constant after executing $E(X,Z)$, we know the invocation of $T$ in $C_2$ will also have the first argument bound. It follows that **P** will run under binding rule $T$:1. This in turn implies that it will run under binding rule $E$:1, because $E$ always inherits its first argument from $T$.

On the other hand, consider program **Q**:

$C_1$: $T(X,Y) \leftarrow E(X,Y)$.
$C_2$: $T(X,Y) \leftarrow T(Z,Y), E(X,Z)$.

Note that this is logically equivalent to program **P**. From the same input rule, $T$:1, no binding rules can be inferred. In particular, **Q** will not run under binding rule $T$:1, because the invocation of $T$ in $C_2$ will be with a free variable as first argument.

Given a set of input rules $I$ for program **P**, we say $I$ *implies* binding rule $b$ if whenever **P** is run under every input rule in $I$ it necessarily runs under binding rule $b$. We will defer the problem of computing an approximation to the binding rules implied by a given set of input rules until Section 3.

## 3. Inserting Cuts

Our method inserts a cut after a literal if the functional dependencies imply that the bound variables for that literal functionally determine every variable in the literal that is used further to its right or that appears in the head of the clause. The idea is that backtracking to this literal again cannot make any difference in what happens to the right of it, since all free variables that are "live" must take exactly the same values again. We can show that the method is correct in the sense that the modified program computes at least all the tuples that the original one computed for the same query.[†]

We shall use the following notation:

$$A \leftarrow B_1(\mathbf{x}_1), \ldots, B!_i(\mathbf{x}_i), \ldots, B_n(\mathbf{x}_n).$$

is equivalent to

$$A \leftarrow B_1(\mathbf{x}_1), \ldots, B_{i-1}(\mathbf{x}_{i-1}), \alpha(\mathbf{x}_i), B_{i+1}(\mathbf{x}_{i+1}), \ldots, B_n(\mathbf{x}_n).$$
$$\alpha(\mathbf{x}_i) \leftarrow B_i(\mathbf{x}_i), !.$$

[†] In some cases, the modified program may actually compute *more* tuples than the original one, while remaining sound.

where $\alpha$ is a predicate appearing nowhere else.

Given program **P**, input rules $I$, and set of functional dependencies $F$, let $C_j$ be a clause

$$\begin{aligned}C_j : A(x_1, \ldots, x_n) \leftarrow\ & B_1(y_{11}, \ldots, y_{1k_1}),\\ & B_i(y_{i1}, \ldots, y_{ik_i}),\\ & B_m(y_{m1}, \ldots, y_{mk_m}).\end{aligned}$$

Let $\{z_1, \ldots, z_p\}$ be the set of variables that appear among the $y_{ij}$'s and also appear as some $y_{kj}$ for $k > i$ or as some $x_l$; that is, the variables that appear both as arguments of $B_i$ and either to the right of it or as arguments to the left hand side of the clause. Suppose that the input rules $I$ imply the binding rule

$$B_i : j_1, \ldots, j_q$$

and the functional dependencies $F$ imply the functional dependency

$$B_i(y_{i1}, \ldots, y_{ik_i}) : y_{ij_1}, \ldots, y_{ij_q} \rightarrow z_1, \ldots, z_p.$$

Then replace $B_i$ in clause $C_j$ by $B!_i$.

This transformation can be shown correct in the following sense (we omit the proof to save space).

THEOREM 1: Let **P!** be the result of applying the transformation above to program **P**. Then for any query $\phi$ that satisfies the input rules $I$,

$$Pr(\mathbf{P},\phi) \subset Pr(\mathbf{P!},\phi) \subset \mathbf{P}(\phi)$$

EXAMPLE 2: Consider the program for computing transitive closure from Example 1, but augmented with a functional dependency that constrains the relation to be a function:

```
E(X,Y) : X→ Y.
T(X,Y) ← E(X,Y).
T(X,Y) ← E(X,Z), T(Z,Y).
```

Suppose the input rule, as in Example 1, is $T$ :1, which implies the binding rules $T$ :1 and $E$ :1. Then the transformation can be applied to both clauses of the program, producing:

```
T(X,Y) ← E!(X,Y).
T(X,Y) ← E!(X,Z), T(Z,Y).
```

Intuitively, what our transformations have done is to take advantage of the fact that any node related to a given fixed node can be found by following the edges of $E$ without backtracking.

EXAMPLE 3: Consider the following program **P**, where a *project*$(p, m, e)$ tuple means employee $e$ works for project $p$ and manager $m$ manages project $p$.

```
project ( p , m, e) : p → m
project ( p , m, e) : e → p
project ( Turing, Ric, Jim).
project (Turing, Ric, Steve).
salary (Ric, 50K).
bigproj ( p ) ← project( p, m, e),
                salary(m, s),
                s> 50K.
```

Suppose the input rule given is *bigproj* :1. The *bigproj* predicate defines those projects whose manager makes more than 50K. Suppose a query that satisfies the input rule, such as

*bigproj(Turing)*

is posed. After we find that *Jim* does work for project *Turing* but the project manager's salary is not greater than 50K, there is no need to look for other managers of project *Turing*, since the functional dependency tells us there are no others. Our transformation lets us replace *project* by *project*! to reflect this fact.

Note that the potential payoff in the last example is greater than that of the previous one. In the example above, we avoid examining every employee that works on a project once we determine that the project does not meet the query criteria. It is interesting to point out that the potential payoff comes from the inefficiency introduced by the database design: the *project* predicate is not in Boyce-

Codd Normal Form.

It is also interesting to look at the case where the arguments of a predicate do not appear anywhere to its right. Even if there are no functional dependencies, the method can be applied according to our definition.

EXAMPLE 4: Consider the query "find an employee who makes more than 20K and works for a department that sells something":

$$\begin{aligned} ans(e) \leftarrow\ & dept(e,d), \\ & sells(d,p), \\ & salary(e,s), \\ & s > 20K. \end{aligned}$$

Note that the variables in the *sells* literal do not appear to its right nor to the left of the arrow; therefore we can replace *sells* by *sells!*. The effect of this transformation is that, if an employee's salary is not greater than 20K, we are not going to recheck his salary for each product sold by his department. Thus our method solves as a special case the "deep backtracking" problem mentioned by [SW]. It is also similar to the decomposition into independent subproblems described in [W].

## 4. Computing the Binding Rules

Given a set of input rules $I$ for a program P we would like to compute the set of binding rules $b$ such that $I$ implies $b$. However, as is common in data flow analysis problems, this is impossible. In proof, suppose we want to test whether an arbitrary Turing machine, say the $m$th one, halts on the empty tape. Then we can write the following program:

$$\begin{aligned} q(x) &\leftarrow p(x). \\ q(x) &\leftarrow T_m, p(z). \end{aligned}$$

where $T_m$ is a predicate that succeeds if the $m$th Turing machine halts on the empty tape; otherwise the evaluation of $T_m$ does not terminate. Then the input rule $q:1$ implies the binding rule $p:1$ if and only if the Turing machine does not halt.

For this reason, we compute instead a subset of all the binding rules implied by a set of input rules, by assuming that all literals that appear in a clause actually get invoked in some execution. For our purposes, it is safe to compute a subset, since the fewer binding rules we have, the fewer transformations we can apply to our program.

Given a program P and set of input rules $I$, we construct a graph $G$ as follows. For each $n$-ary predicate $A$ appearing in P, $G$ has $n$ vertices labelled $A_1, A_2, ..., A_n$. If $A$ also appears as the head of some clause in P, $G$ has $n$ additional vertices labelled $A_i^I$, for $1 \le i \le n$. These are called the $I$-vertices. There is also one special node, labelled $\omega^I$.

An edge from $A_i$ to $B_j$ in $G$ intuitively means that the $i$th argument of $A$ will be bound only if the $j$th argument of $B$ is. There are three types of edges:

1) If predicate $A$ appears as head of a clause, there is an edge from $A_i$ to $A_i^I$ for each $1 \le i \le n$.
2) If a clause has head $A$ and an occurrence of some predicate $B$ in its body, and the $j$th argument of $A$ is also the $i$th argument of $B$, and is not an argument to any predicate appearing before $B$ in the body, then there is an edge from $B_i$ to $A_j$.
3) If $A$ occurs in the body of a clause and its $i$th argument is the leftmost occurrence of a variable in the clause, then there is an edge from $A_i$ to $\omega^I$. Figure 1 shows the graph for the transitive closure program of Example 1.

Let $\hat{G}$ be the acyclic condensation of $G$. Mark all $A_i^I$ nodes such that the input rule $P:i$ is in $I$. Now repeat exhaustively the following process: mark any non-$I$ node if all its successors are marked. The following theorem shows that this method computes a subset of the binding rules implied by $I$.

THEOREM 2: For each component $C$ of $\hat{G}$ that is marked by the algorithm, and each $A_i \in C$, $I$ implies the binding rule $A:i$.

EXAMPLE 5: Let us apply the algorithm to the graph in Figure 1, with the single input rule $T:1$. Since the graph is acyclic (except for the $T_2$ self-loop), its acyclic condensation is itself. By the given

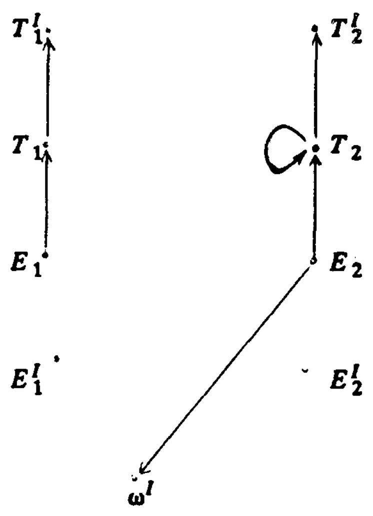

**Figure 1**

input rule, we mark $T_1^l$. It follows that $T_1$ and $E_1$ also get marked, and no other nodes. The result is that binding rules $T$ :1 and $E$ :1 are implied.

**Acknowledgements**

Thanks to Peter Wood for his helpful comments.

**References**

[BJ] M. Brodie and M. Jarke, "On Integrating Logic Programming and Databases," Proc. First Int'l Workshop on Expert Database Systems, 1984, 40-62.

[BP] M. Bruynooghe and L.M. Pereira, "Deduction Revision by Intelligent Backtracking," in *Implementations of Prolog*, J. Campbell (ed.), Ellis-Horwood, 1984.

[CM] W.F. Clocksin and C.S. Mellish, "Programming in Prolog," Springer-Verlag, 1981.

[D] V. Dahl, "On Database Systems Development through Logic," *ACM TODS* 7:1, March 1982, 102-123.

[JCV] M. Jarke, J. Clifford, and Y. Vassiliou, "An Optimizing Prolog Front-End to a Relational Query System," Proc. ACM SIGMOD 1984, 296-306.

[L] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 1984.

[KM] M. Kohli and J. Minker, "Intelligent Control Using Integrity Constraints," Proc. AAAI Conference, 1983, pp. 202-205.

[PP] L.M. Pereira and A. Porto, "Selective Backtracking," in *Logic Programming*, K. Clark and S.-A. Tarnlund (eds.), Academic Press, 1982.

[SW] E. Sciore and D.S. Warren, "Towards an Integrated Database-Prolog System," Proc. First Int'l Workshop on Expert Database Systems, 1984, 801-815.

[U] J.D. Ullman, "Implementation of Logical Query Languages for Databases," Proc. ACM SIGMOD 1985.

[W] D.H.D. Warren, "Efficient Processing of Interactive Relational Database Queries Expressed in Logic," VLDB 1981, 272-281.