Theoretical Computer Science 78 (1991) 245-259
North-Holland

An efficient strategy for non-Horn deductive databases*

Robert Demolombe

ONERA/CERT, 2 avenue E. Belin, B.P. 4025, 31055 Toulouse, France

Abstract

Demolombe, R., An efficient strategy for non-Horn deductive databases, Theoretical Computer Science 78 (1991) 245-259.

Many efficient strategies have been presented in the literature to derive answers to queries in the context of Deductive Databases. Most of them deal with definite Horn clauses. More recently, strategies have been defined to deal with non-Horn clauses but they adopt the Generalized Closed World Assumption to reduce incompleteness. In this paper we present an efficient strategy to deal with non-Horn clauses in pure logic, i.e. without any kind of assumption. This strategy is an extension of the so-called ALEXANDRE strategy to this particular context. The set of rules is transformed into another set of rules, in a compilation phase, in such a way that their execution in forward chaining simulates a variant of SL resolution. Particular attention is payed to the constants in the queries to reduce the set of derived clauses.

1. Introduction

A Deductive Data Base (DDB) is usually defined as a set of facts, the Extensional Data Base (EDB) and a set of rules, the Intensional Data Base (IDB). Many studies have been devoted to defining efficient strategies for query evaluation in the case where the rules are Horn clauses [2, 14, 13, 11], i.e. clauses of the form

$$A_1 \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_m$$

where the B_i are positive atomic formulas.

In the context of Logic Programming, the form of the rules has been extended to clauses where the B_i may be negative atomic formulas which correspond in pure logic to non-Horn clauses of the form

$$A_1 \vee A_2 \vee \cdots \vee A_n \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_m.$$

However, the semantics assigned to these clauses in Logic Programming is not exactly the same as in Logic. Indeed the negation is evaluated by default, according to the negation as failure (NAF) rule [5].

* This work was partially supported by the C.E.C. in the context of the ESPRIT project ESTEAM-316.

Let us consider, for example, the Logic Program P_1

(1) $\text{Man}(x) \leftarrow \text{General}(x) \wedge \text{not Woman}(x)$,

(2) $\text{Woman}(x) \leftarrow \text{Mother}(x, y)$,

(3) $\text{Man}(x) \leftarrow \text{Father}(x, y)$.

$$\text{General}(a)\ \text{General}(b)\ \text{Mother}(b, c)\ \text{Father}(c, d).$$

If we ask the query $\text{Man}(x)$?, we get the answer $\{a, c\}$, because there is no way to prove $\text{Woman}(a)$ and then not $\text{Woman}(a)$ is assumed to be true, though $\neg\text{Woman}(a)$ is not provable in standard logic.

A similar approach is adopted in the context of Stratified Programs [1] which are Logic Programs with the additional restriction that recursive definitions of the predicates cannot contain negations in their definition cycle. For example P_1 is a stratified program. However if we add to P_1 the rule

(4′) $\text{Woman}(x) \leftarrow \text{Love}(x, y) \wedge \text{Man}(y)$,

we get a non-stratified program because in the Man definition cycle we have Man, not Woman, Woman, Man. A simpler case of non-stratified programs is the single rule

(5′) $\text{Woman}(x) \leftarrow \text{Love}(x, y) \wedge \text{not Woman}(y)$.

The intuitive idea behind stratified programs is that to compute a negation we have to compute first all the elements satisfying the negated predicate, and then we assume that they are the only ones. This is not possible for the previous rule (5′) because to compute not $\text{Woman}(y)$ we have to compute first the set of all women, which needs that not $\text{Woman}(y)$ have been computed.

For Stratified Programs as for NAF, an assumption is made which is not valid in standard logic. A more general approach in the same stream is to define default rules [10]. In this approach the rule (1) above is written

$$\frac{\text{General}(x) : \text{not Woman}(x)}{\text{Man}(x)}.$$

Its intuitive meaning is that for any x who is a general, if there is no contradiction to assume that x is not a woman, then we assume that x is not a woman and we can derive that x is a man.

Though it may be perfectly relevant to adopt this kind of default reasoning in some particular context, an important drawback is that we cannot make the distinction between theorems derived using only standard inference rules and theorems whose proof involves default rules. Roughly speaking we cannot distinguish "safe" answers and "extended" answers. For example in the case of the program P_1 for the answer to the query $\text{Man}(x)$?, we get in the same set the answer c, which is a safe answer and a, which is an extended answer.

The objective of this paper is to offer the possibility *not to use default reasoning* in order to be able to provide answers which are guaranteed to be safe and *not to restrict clauses to be Horn clauses.*

Thus we consider Deductive Data Bases with non-Horn clauses, including clauses with no positive literal, and the proofs involve only standard inference rules.

Let us consider for example the DDB P_2

(1) $\text{Man}(x) \vee \text{Woman}(x) \leftarrow \text{General}(x)$,

(2) $\text{Woman}(x) \leftarrow \text{Mother}(x, y)$,

(3) $\text{Man}(x) \leftarrow \text{Father}(x, y)$,

(4) $\leftarrow \text{Woman}(x) \wedge \text{Love}(x, y) \wedge \text{Woman}(y)$,

(5) $\leftarrow \text{Man}(x) \wedge \text{Woman}(x)$.

$\text{General}(a)\ \text{General}(b)\ \text{Mother}(b, c)\ \text{Father}(c, d)$,

$\text{General}(e)\ \text{Love}(e, b)$.

The meaning of clause (4) is that it is contradictory that a woman x loves another woman y, and (5) means that it is contradictory for any x to be both a man and a woman.

If we ask the query $\text{Man}(x)$?, we get $\{c, e\}$. These two answers are derived by standard proofs. For c the proof is obvious using (3). For e, we derive $\text{Woman}(b)$ from (2) and $\text{Mother}(b, c)$, then we derive $\neg\text{Woman}(e)$ from (4) and $\text{Love}(e, b)$; from $\text{General}(e)$ and (1) we derive $\text{Man}(e) \vee \text{Woman}(e)$ which gives $\text{Man}(e)$ using $\neg\text{Woman}(e)$.

Here we cannot derive $\text{Man}(a)$; the only safe information derivable about a is $\text{Man}(a) \vee \text{Woman}(a)$. However, if in a particular context it is relevant to apply some kind of default rule, that can be done in a further step. For example it is possible to define a preference order on the predicates or on the facts such that from $A < B$ and $A \vee B$ we derive B. In this approach instead of stratified programs we can consider the corresponding non-Horn clauses to compute the safe answer, and then in a second step, the order on the predicates defined by the stratified program can be used to compute the extended answer, which is identical to the answer computed by the initial stratified program (see [12]). Our conclusion is that standard reasoning and default reasoning are complementary and can be applied in two sequential steps.

In Sections 2 and 3, we first formally define the notion of Indefinite DDB and associated least fixpoint operator. Then in Section 4, an efficient strategy to compute answers is defined for propositional calculus; this strategy is extended to predicate calculus in Section 5.

2. Indefinite Deductive Data Base

In this section we introduce some formal definitions:

- Indefinite Deductive Data Base (IDDB): set of clauses of a First Order Language without function symbols. It is composed of three parts: the extensional part, the intensional part and the query part.

- Extensional Data Base (EDB): set of ground positive clauses, i.e. clauses with only positive literals and without variable. Example: $\text{Love}(e, b) \vee \text{Love}(b, e)$, $\text{General}(a)$.
- Intensional Data Base (IDB): set of Range Restricted clauses having at least one negative literal. A clause is Range Restricted (see [6]) iff each variable occurring in a positive literal also occurs in a negative literal. Clauses are represented in the implicative form just to make easier their understanding. Example: $\text{Man}(x) \vee \text{Man}(y) \leftarrow \text{Love}(x, y)$.
- Query: any formula of the language such that its prenex conjunctive normal form is Range Restricted and does not contain universal quantifiers. The latter restriction is to does not introduce Skolem functions when a query is transformed in the clausal form. Examples:

$$q_1 = \text{General}(x) \wedge \exists y(\text{Love}(x, y) \vee \text{Love}(y, x)),$$

$$q_2 = \text{General}(x) \wedge \neg\text{Man}(x).$$

- Query clauses: let $F(t)$ be a query with the free variable tuple t, we extend the language with a new predicate symbol $q(t)$ and we consider the formula

$$\text{qf} = (q(t) \leftarrow F(t)) \quad \forall t.$$

The set of query clauses of $F(t)$ is the set of clauses we obtain when qf is put in clausal form. Examples:

$$\text{qf}_1 = (q(x) \leftarrow \text{General}(x) \wedge \exists y(\text{Love}(x, y) \vee \text{Love}(y, x))) \quad \forall x,$$

$$\text{qf}_2 = (q(x) \leftarrow \text{General}(x) \wedge \neg\text{Man}(x)) \quad \forall x.$$

For example the set of query clauses Q_1 of qf_1 is

$$Q_1 = \{q(x) \leftarrow \text{General}(x) \wedge \text{Love}(x, y), q(x) \leftarrow \text{General}(x) \wedge \text{Love}(y, x)\}.$$

The set of query clauses Q_2 of qf_2 is

$$Q_2 = \{q(x) \vee \text{Man}(x) \leftarrow \text{General}(x)\}.$$

- Logic Program: we define a Logic Program P as a consistent set of clauses of the form: $Q \cup \text{IDB} \cup \text{EDB}$.
- Answer: the answer Ans to the query Q w.r.t. $\text{IDB} \cup \text{EDB}$ is

$$\text{Ans} = \{t_0 \mid P \vdash q(t_0)\}$$

where t_0 is a constant tuple.

3. Least fixpoint operator for an Indefinite Deductive Data Base

The answer definition presented in the previous section gives a formal semantics but it is not operational. We introduce now a least fixpoint operator which can be viewed as an intermediate step between the theoretical definition and an efficient strategy definition.

This operator has been defined in [8] in a more general context where function symbols are allowed. The only sensible difference is that we allow clauses with no positive literal. However the results presented in [8] remain valid for consistent Logic Programs.

We briefly recall some definitions. Let P be a Logic Program.

- Herbrand Universe U of P: set of constant symbols occurring in P.
- Non-Horn Herbrand Base NHB(P) of P: set of ground positive clauses formed with the elements of U, with no duplicated literals.

Notations:

- powset(S): power set of S.
- fact($L_1 \vee L_2 \vee \cdots \vee L_n$): factorization of the literals L_i; i.e. duplicated literals are removed in the fact result.

Definition 3.1 (T_P *operator*). The signature of T_P is powset(NHB(P))→ powset(NHB(P)). Let S be an element of powset(NHB(P)); the T_P definition is

$$T_P(S) = \{A_1 \vee A_2 \vee \cdots \vee A_p \mid A_1 \vee A_2 \vee \cdots \vee A_p \in S \text{ or}$$
$$\text{there exists a clause instance in } P \text{ of the form:}$$
$$A_1 \vee A_2 \vee \cdots \vee A_n \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_m$$
$$\text{and for } i \in [1, m] \text{ there exists a clause of the form } B_i \vee c_i$$
$$\text{in } P \text{ or in } S \text{ such that:}$$
$$A_1 \vee A_2 \vee \cdots \vee A_p$$
$$= \text{fact}(A_1 \vee A_2 \vee \cdots \vee A_n \vee c_1 \vee c_2 \vee \cdots \vee c_m)\}$$

where $B_i \vee c_i$ is a ground positive clause containing B_i, and $n + m > 0$.

It can be noticed that $T_P(\emptyset) = \text{EDB}$. When there is no risk of ambiguity $T'_P(\emptyset)$ is denoted by T'.

The most important properties of the T operator are presented in the following theorems (we will give here only the most important steps of the proofs).

Theorem 3.2 (T Termination). *For a given Logic Program P there exists an n such that for any $p > n$ we have $T_P^p(\emptyset) = T_P^n(\emptyset) = M$. M is the least fixpoint of P for T.*

Proof. It is proven in [8] that T is monotonic for the inclusion. Since there is no function symbol in the language, U is finite and NHB(P) is also finite. □

Theorem 3.3 (T validity). *Any clause c in M is a theorem of P.*

Proof. We can notice that the T operator computes the same clauses as the clauses obtained by a saturation by level using the Positive Hyperresolution inference rule (PH) defined in [3, p. 108]. It is also proven in [3] that PH is valid and complete to derive the empty clause. Moreover we can show that any proof of the empty clause from P and the clausal form of $\neg c$ can be transformed into a direct proof of c or of a clause subsuming c, in the case where c is a clause. Hence PH is also valid and complete (up to subsumption) when it is used to derive clauses. □

Theorem 3.4 (T completeness). *For any positive ground clause c which is a theorem of P there exists a clause c′ in M such that c′ subsumes c.*

Proof. The theorem is a direct consequence of the PH completeness mentioned above. □

Lemma 3.5. *We have* $\text{Ans} = \{t_0 | q(t_0) \in M\}$.

4. Evaluation strategy in the case of Propositional Calculus

The basic idea of the strategy is to try to reconstruct the intermediate steps of the computations leading to the answer. This needs to alternate forward chaining and backward chaining. The role of backward chaining is to determine, using the rules in $\text{IDB} \cup Q$, the form of the clauses in EDB or in previous results which can generate answers by execution of T. The role of forward chaining is to generate new clauses with T using only clauses of the form determined by the backward chaining steps.

For example if we have to produce a clause c_0 like

$$(c_0) \quad A_1 \vee A_2 \vee A_3 \vee A_4,$$

we have to look for all the clauses in $\text{IDB} \cup Q$ whose consequence is a subset of c_0. If there is for example a clause C:

$$(C) \quad A_1 \vee A_2 \leftarrow B_1 \wedge B_2,$$

the backward chaining phase determines that we have to look for the following clauses c_1 and c_2 or clauses which subsume them:

$$(c_1) \quad B_1 \vee A_1 \vee A_2 \vee A_3 \vee A_4,$$

$$(c_2) \quad B_2 \vee A_1 \vee A_2 \vee A_3 \vee A_4.$$

In the following we call $\text{Pb}(c_0)$ the initial problem which can be formulated: "is it possible to generate c_0?", and we call $\text{Pb}(c_1)$ and $\text{Pb}(c_2)$ the two sub-problems derived from (C).

If, for example, we find in EDB or in previous results the clauses c_1' and c_2':

$$(c_1') \quad B_1 \vee A_2,$$

$$(c_2') \quad B_2 \vee A_1 \vee A_4,$$

the forward chaining phase will produce with T the following solution c_0' to the initial problem $\text{Pb}(c_0)$:

$$(c_0') \quad A_1 \vee A_2 \vee A_4$$

which is denoted $\text{Sol}(c_0')$, while the solutions to the sub-problems are denoted by $\text{Sol}(c_1')$ and $\text{Sol}(c_2')$.

This example suggests an operational view of the clause C which expresses the backward chaining and forward chaining ideas. Indeed if we have to solve a problem of the form: $\mathrm{Pb}(A_1 \vee A_2 \vee X)$ using (C) we have to solve the two sub-problems $\mathrm{Pb}(B_1 \vee A_1 \vee A_2 \vee X)$ and $\mathrm{Pb}(B_2 \vee A_1 \vee A_2 \vee X)$ where X can be substituted by any ground positive clause.

The sub-problems can be solved in parallel or in sequence. Since we want to generalize the strategy to Predicate Calculus we will solve the sub-problems in sequence because the solutions to the first sub-problem allows us to generate second ones where some variables may be instantiated, and then which are more specific.

The solutions we are looking for are represented by $\mathrm{Sol}(B_1 \vee A_1 \vee A_2 \vee X)$ and $\mathrm{Sol}(B_2 \vee A_1 \vee A_2 \vee X)$. This means that any previous solutions which subsumes these clauses are solutions to the sub-problems. The solutions to the initial problem are represented by $\mathrm{Sol}(A_1 \vee A_2 \vee X)$. Then the initial clause C can be translated, as it is done for the ALEXANDRE strategy [11, 7] in the case of definite horn clauses, into the following rules whose execution in a forward strategy simulates the backward chaining and forward chaining processes. That is, their execution alternatively generates problems to be solved and/or solutions to the generated problems.

(C_0) $\mathrm{Pb}(A_1 \vee A_2 \vee X) \rightarrow \mathrm{Pb}(B_1 \vee A_1 \vee A_2 \vee X)$,

(C_1) $\mathrm{Pb}(A_1 \vee A_2 \vee X) \wedge \mathrm{Sol}(B_1 \vee A_1 \vee A_2 \vee X) \rightarrow \mathrm{Pb}(B_2 \vee A_1 \vee A_2 \vee X)$,

(C_2) $\mathrm{Pb}(A_1 \vee A_2 \vee X) \wedge \mathrm{Sol}(B_1 \vee A_1 \vee A_2 \vee X)$
$\wedge \mathrm{Sol}(B_2 \vee A_1 \vee A_2 \vee X) \rightarrow \mathrm{Sol}(A_1 \vee A_2 \vee X)$.

If we "execute" these rules with

$\mathrm{Pb}(A_1 \vee A_2 \vee A_3 \vee A_4)$,

X instantiated by $A_3 \vee A_4$ and (C_0) generates

$\mathrm{Pb}(B_1 \vee A_1 \vee A_2 \vee A_3 \vee A_4)$,

from $\mathrm{Sol}(B_1 \vee A_2)$ the rule (C_1) generates

$\mathrm{Pb}(B_2 \vee A_1 \vee A_2 \vee A_3 \vee A_4)$

and from $\mathrm{Sol}(B_2 \vee A_2 \vee A_4)$ the rule (C_2) generates

$\mathrm{Sol}(A_1 \vee A_2 \vee A_4)$.

We can make some comments about these transformed rules:

- From a formal point of view the predicates $\mathrm{Pb}(x)$ and $\mathrm{Sol}(x)$ are metapredicates whose arguments are codes of formulas, and the rules are axioms of a meta-Theory. These axioms define the derivation process control relative to the object theory. However it would be very difficult to define this meta-Theory in detail and we will stay a bit informal when formula meaning is obvious.
- The order of the literals in the disjunctions is irrelevant; in particular $\mathrm{Pb}(X \vee Y)$ (resp. $\mathrm{Sol}(X \vee Y)$) are supposed to represent the *same problem* (resp. the *same solution*) as $\mathrm{Pb}(Y \vee X)$ (resp. $\mathrm{Sol}(Y \vee X)$).

- Conditions of the form $\text{Sol}(B_1 \vee A_1 \vee A_2 \vee X)$ in the rules are satisfied, for a given X instantiation, by any solution of the form $\text{Sol}(B_1 \vee D_1 \vee D_2 \vee \cdots \vee D_p)$ such that $D_1 \vee D_2 \vee \cdots \vee D_p$ is a subset of $A_1 \vee A_2 \vee X$. The reason why we impose the presence of B_1 is that a solution which does not contain B_1 is a direct solution to the initial problem, and the execution of the rules in this case introduces useless computations.
- The reason why we have conditions of the form $\text{Sol}(B_i \vee A_1 \vee A_2 \vee X)$, and not of the simpler form $\text{Sol}(B_i)$ is that this latter condition is too strong to derive $A_1 \vee A_2$. For example we can derive A from $A \leftarrow B$ and $B \vee A$; it is not necessary to have B. Hence a strategy with conditions of the form $\text{Sol}(B_i)$ would not be complete.
- The clauses with no positive literal have transformed rules which can potentially be used to solve any problem, $\text{Pb}(X)$.

We give now a general definition of the clause transformation.

Definition 4.1 (*Θ transformation for propositional calculus*). Let P be a Logic Program; the transformed Logic Program $P' = \Theta(P)$ is defined as follows. For any clause C in $\text{IDB} \cup Q$ of the form

$$(C) \quad A \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_m$$

where $A = A_1 \vee A_2 \vee \cdots \vee A_n$, we have in P' the set of rules:

$$(C_0) \quad \text{Pb}(A \vee X) \rightarrow \text{Pb}(B_1 \vee A \vee X)$$

$$(C_1) \quad \text{Pb}(A \vee X) \wedge \text{Sol}(B_1 \vee A \vee X) \rightarrow \text{Pb}(B_2 \vee A \vee X)$$

$$\vdots$$

$$(C_{j-1}) \quad \text{Pb}(A \vee X) \wedge \text{Sol}(B_1 \vee A \vee X) \wedge \text{Sol}(B_2 \vee A \vee X) \vdots \wedge \text{Sol}(B_{j-1} \vee A \vee X) \rightarrow \text{Pb}(B_j \vee A \vee X)$$

$$\vdots$$

$$(C_m) \quad \text{Pb}(A \vee X) \wedge \text{Sol}(B_1 \vee A \vee X) \wedge \text{Sol}(B_2 \vee A \vee X) \vdots \wedge \text{Sol}(B_m \vee A \vee X) \rightarrow \text{Sol}(A \vee X).$$

For any clause c in EDB we have in P', $\text{Ax}(c)$. Finally we also have in P'

$$(\text{ax}) \quad \text{Pb}(X \vee Y) \wedge \text{Ax}(X) \rightarrow \text{Sol}(X)$$

$$(q) \quad \text{Pb}(q).$$

The transformed Logic Program P' is completely defined by the rules $(C_i)s$, $\text{Ax}(c)s$, (ax) and (q).

We define now a least fixpoint operator for the transformed program P'.

Definition 4.2 ($T'_{P'}$ *operator for propositional calculus*). Let us define SOL and PB as:

$$\mathrm{SOL} = \{\mathrm{Sol}(c) \mid c \in \mathrm{NHB}(P)\}, \qquad \mathrm{PB} = \{\mathrm{Pb}(c) \mid c \in \mathrm{NHB}(P)\}.$$

The Transformed Non-Herbrand Base of P' is defined as $\mathrm{TNHB}(P') = \mathrm{SOL} \cup \mathrm{PB}$.

The signature of T' is $\mathrm{powset}(\mathrm{TNHB}(P')) \to \mathrm{powset}(\mathrm{TNHB}(P'))$. Let S' be an element of $\mathrm{powset}(\mathrm{TNHB}(P'))$; the $T'_{P'}$ definition is

$T'_{P'}(S') = \{\mathrm{Pb}(B_j \vee A \vee C) \mid \mathrm{Pb}(B_j \vee A \vee C) \in S'$ or
there exists a rule (C_{j-1}) instance where X is instantiated by C
s.t. for $k \in [1, j-1]$ there is in S' a solution of the form
$\mathrm{Sol}(B_k \vee A_k \vee C_k)$ where A_k and C_k are respectively
subsets of A and C
and there is in S' the problem $\mathrm{Pb}(A \vee C)\}$
$\cup$
$\{\mathrm{Sol}(A \vee C') \mid \mathrm{Sol}(A \vee C') \in S'$ or
there exists a rule (C_m) instance where X is instantiated by C
s.t. for $k \in [1, m]$ there is in S' a solution of the form
$\mathrm{Sol}(B_k \vee A_k \vee C_k)$ where A_k and C_k are respectively subsets of A and C
and there is in S' the problem $\mathrm{Pb}(A \vee C)\}$
and $C' = \mathrm{fact}(C_1 \vee C_2 \vee \cdots \vee C_m)$ is a subset of $C\}$
$\cup$
$\{\mathrm{Sol}(A) \mid \mathrm{Ax}(A) \in S'$ and $\mathrm{Pb}(A \vee C) \in S'\}$

where A, C, and C' are ground positive clauses and the B_k are ground positive literals.

In order to improve the efficiency of T' execution we assume in the following that T' is executed according to the *semi-naive strategy* [2], also called delta strategy. This means that a rule is applied only if at least one operand (pre-condition) is new. This strategy prevents repetition of the same computations.

We briefly present the most important properties of the transformed program P' and of the T' operator. We call M' the least fixpoint of $T'_{P'}(\emptyset)$. Theorem 4.5 shows that M' is well defined.

Theorem 4.3 (T' validity). *If* $\mathrm{Sol}(A) \in M'$ *or* $\mathrm{Ax}(A) \in M'$ *then* $A \in M$, *where* A *is a ground positive clause.*

Proof. The idea of the proof of Theorems 4.3 and 4.4 is very close to the proof given by Kerisit in [7]. The proofs here are only more complex due to non-Horn clauses.

The proof is by induction on i in the computation of $T'^{\uparrow}_{P'}$. So we have to prove that for any i we have

$$\mathrm{Sol}(A) \in T'^{\uparrow}_{P'}(\emptyset) \vee \mathrm{Ax}(A) \in T'^{\uparrow}_{P'}(\emptyset) \Rightarrow A \in M.$$

For $i = 1$ the result is obvious.

Let us assume the induction hypothesis for any $j \leqslant i$. If we have $\mathrm{Ax}(A) \in T'^{i+1}_{P'}(\emptyset)$ the property directly follows the P' definition. If we have $\mathrm{Sol}(A) \in T'^{i+1}_{P'}(\emptyset)$, there exists an instance of rule (C_m) whose conclusion is $\mathrm{Sol}(A)$ and s.t. its pre-conditions, $\mathrm{Pb}(A)$ and all the $\mathrm{Sol}(B_k \vee A)$ belong to $T'^{i}_{P'}(\emptyset)$. Then by induction hypothesis we can conclude that for all the $B_k \vee A$ we have $B_k \vee A \in M$. Hence the application of T to the clause C which has generated C_m, and to the $B_k \vee A$ generates A in M. □

Theorem 4.4 (T' completeness). *If* $\mathrm{Pb}(A) \in M'$ *and* $A' \in M$ *then* $\mathrm{Sol}(A') \in M'$, *where* A *is a ground positive clause and* A' *is a sub-set of* A.

Proof. The proof is by induction on i in the computation of T'_P. So we have to prove that for any i we have

$$\mathrm{Pb}(A) \in M' \wedge A' \in T'^{i}_{P}(\emptyset) \Rightarrow \mathrm{Sol}(A') \in M'.$$

For $i = 1$, a direct application of the (ax) rule gives the result.

Let us assume the property for any $j \leqslant i$. From $A' \in T'^{i}_{P}(\emptyset)$ we conclude that there exists an instance of a clause C and a set of clauses $B_k \vee C_k \in T'^{i-1}_{P}(\emptyset)$ s.t. $A' = \mathrm{fact}(A \vee C_1 \vee C_2 \vee \cdots \vee C_m)$.

From the rule (C_0) and the hypothesis $\mathrm{Pb}(A) \in M'$ we conclude that $\mathrm{Pb}(B_1 \vee A) \in M'$. From the induction hypothesis and $\mathrm{Pb}(B_1 \vee A) \in M'$ and $B_1 \vee C_1 \in T'^{i-1}_{P}$ we have $\mathrm{Sol}(B_1 \vee C_1) \in M'$. By application of the rule (C_1) we derive $\mathrm{Pb}(B_2 \vee A) \in M'$. The same kind of reasoning successively applied to $(C_2), (C_3), \ldots, (C_m)$ gives $\mathrm{Sol}(A) \in M'$. □

Theorem 4.5 (T' termination). *For a given transformed Logic Program* P' *there exists an* n *such that for any* $p > n$ *we have* $T'^{p}_{P'}(\emptyset) = T'^{n}_{P'}(\emptyset) = M'$. M' *is the least fixpoint.*

Proof. The proof is a direct consequence of T' monotonicity and of pow-set(TNHB(P')) finiteness. □

5. Extending the evaluation strategy to Predicate Calculus

A trivial extension of the strategy to Predicate Calculus would be to consider all the clause instances in IDB $\cup$ Q obtained with the elements in the Herbrand Universe, and to come back to Propositional Calculus. This approach is possible in theory since this Universe is finite and since we have no universal quantifiers in the queries. However we would get a huge amount of clauses and the efficiency of the method would be quite poor.

Then our approach is to really deal with Predicate Calculus. The most important difficulty in this context is to have a finite set of transformed clauses able to solve an infinite set of distinct problems. Indeed we may have an infinite set of distinct problems or queries of the form

$$A(x, z_n) \vee A(x, z_{n-1}) \vee \cdots \vee A(x, z_1) \vee A(x, y)$$

which may be generated by a recursive clause like

$$A(x, y) \leftarrow A(x, z) \wedge B(z, y).$$

The basic idea to solve this difficulty is similar than those used in ALEXANDRE for Horn clauses. That is, we define a more abstract notion of problem, called problem type, and we define the transformation for problem types in such a way that a finite set of rules can potentially solve an infinite set of distinct problem instances. Now the question is: how to define problem types for non-Horn clauses?

For Horn clauses the two problems, $\mathrm{Pb}(A(a, y))$ and $\mathrm{Pb}(A(b, y))$, are considered as two instances of the same problem type: $\mathrm{Pb}(x, y)$ where x is known and y is unknown. So we need to represent, at the meta level, the mode of each predicate argument. A formal representation would be to add a new argument for each argument in the initial predicate to represent the modes. For example,

$$\mathrm{Pb}(A(i, x, o, y))$$

where the constant i means input mode, and the constant o means output mode. Since this notation is quite heavy the mode of predicate arguments will be implicitely denoted using the following notations: $x_1, x_2, \ldots, x_k, \ldots$ variables denote arguments having the *input mode*, and $y_1, y_2, \ldots, y_k, \ldots$ variables denote arguments having the *output mode*. So the same problem type will be represented by

$$\mathrm{Pb}(A(x_1, y_1))$$

when there is no risk of confusion. In the case of non-Horn clauses we generalize the notion of problem type to positive clauses and a problem like $A_1(a, y) \vee A_2(b, y)$ is an instance of a problem type represented by

$$\mathrm{Pb}(A_1(x_1, y_1)) \vee A_2(x_2, y_1))$$

In this problem type representation, the variable numbering and the literal order in the clauses are irrelevant. For example the above problem and

$$\mathrm{Pb}(A_2(x_3, y_2)) \vee A_1(x_2, y_2))$$

are considered as two representations of *the same problem type*. However in some cases this notion of problem type is not general enough to represent problems of no bounded length and we extend the problem type definition to problem types of the form

$$\mathrm{Pb}(A_1(x_1, y_1) \vee A_2(x_2, y_1) \vee X)$$

whose meaning is: "is it possible, for a given instantiation of the variables x_1 and x_2 to find instantiations of X and y_1 such that $\mathrm{Sol}(A_1(x_1, y_1) \vee A_2(x_2, y_1) \vee X)$ can be generated from P'?". For example for the instantiation $\{a/x_1, b/x_2\}$ we could find the instantiation $\{A_3(x_3, x_3, y_1)/X, c/x_3, d/y_1\}$.

To have more concise notations a disjunction of the form $A_1(x_1, y_1) \vee A_2(x_2, y_1)$, is denoted by $A(t_x, t_y)$ where A is a positive disjunction, t stands for the free variable

tuple, and t_x (resp. t_y) stands for the input (resp. output) free variables (sometimes t will be considered as a set of variables). So, in general, a problem type is represented by

$$\mathrm{Pb}(A(t_x, t_y) \vee X)$$

and corresponding solutions are represented by

$$\mathrm{Sol}(A'(t') \vee B(s'))$$

where A' is a subset of A, t' is an instance of t, and $B(s')$ is a ground instance of X.

Using these notations, the transformation definition in the case of Predicate Calculus is:

Definition 5.1 (*Θ transformation for predicate calculus*). For any clause C in $\mathrm{IDB} \cup Q$ of the form

$$A_1(s_1) \vee A_2(s_2) \vee \cdots \vee A_n(s_n) \leftarrow B_1(t_1) \vee B_2(t_2) \vee \cdots \vee B_m(t_m)$$

which is written in the form

$$A(t) \leftarrow B_1(t_1) \vee B_2(t_2) \vee \cdots \vee B_m(t_m)$$

where $A(t) = A_1(s_1) \vee A_2(s_2) \vee \cdots \vee A_n(s_n)$.

For any choice of the argument modes of the literals in $A(t)$, called a *signature*, and represented by $A(t_x, t_y)$ we have the set of rules

$$(C_0) \quad \mathrm{Pb}(A'(t_x, t_y) \vee X) \rightarrow \mathrm{Pb}(B_1(t_{1x}, t_{1y}) \vee X \vee Y)$$

$$\vdots$$

$$(C_{k-1}) \quad \mathrm{Pb}(A'(t_x, t_y) \vee X) \wedge \mathrm{Sol}(B_1(t_1) \vee A(t) \vee X) \wedge \mathrm{Sol}(B_2(t_2) \vee A(t) \vee X) \vdots \wedge \mathrm{Sol}(B_{k-1}(t_{k-1}) \vee A(t) \vee X) \rightarrow \mathrm{Pb}(B_k(t_{kx}, t_{ky}) \vee A(t_x, t_y) \vee X)$$

$$\vdots$$

$$(C_m) \quad \mathrm{Pb}(A'(t_x, t_y) \vee X) \wedge \mathrm{Sol}(B_1(t_1) \vee A(t) \vee X) \wedge \mathrm{Sol}(B_2(t_2) \vee A(t) \vee X) \vdots \wedge \mathrm{Sol}(B_m(t_m) \vee A(t) \vee X) \rightarrow \mathrm{Sol}(A(t) \vee X)$$

where A' is a subset of the literals in A which may be equal to A or may be an empty set in the extreme cases,

$$t_{kx} = t_k \cap (t_x \cup t_1 \cup t_2 \cup \cdots \cup t_{k-1}) \quad \text{and} \quad t_{ky} = t_k / t_{kx}.$$

All the variables in the rules are universally quantified except t_y and the $t_i y$ which are constants at this meta level.

Some comments about this definition:

- The rule (C_0) has not the same form as the others. The reason is that if we had in the consequence, $\text{Pb}(B_1(t_{1x}, t_{1y}) \vee A(t_x, t_y) \vee X)$, in the case where the predicate B_1 is recursively defined, we could generate problems of an infinite length, which is not the case for the rules (C_k) since the generation of new problems depends on the existence of new solutions, and the number of new solutions is finite.
- It can be noticed that for a given instance of X, (C_0) generates a new problem where Y is a variable. We will consider that two problem types of the form $\text{Pb}(PB \vee Y)$ and $\text{Pb}(PB \vee Y \vee Y')$ represent the *same* problem type and are both represented by $\text{Pb}(PB \vee Y)$.
- The rules (C_k) may be executed even if A' is empty, for example to solve a problem of the form $\text{Pb}(D(s'_x, s_y) \vee Y)$ such that D has no common predicate with A. In that case X is instantiated by $D(s'_x, s_y)$ and the rule (C_m) may generate solutions like $\text{Sol}(A(t') \vee D(s'))$ which correspond to solutions to the initial problem where Y is instantiated by $D(s')$.
- For a given $A(t)$ signature, the literal B_i are ordered according to some heuristic as for Horn clauses. The simplest heuristic is to order the literal with the criteria to have the maximum number of arguments in each literal having the input mode. It can be noticed that the Range Restricted property mentioned at the beginning guarantees that all the input variables t_x in A appear in some B_i. Roughly speaking this guarantees that the constants in the initial query or in the problems are transmitted to the sub-problems and that the number of generated solutions is strongly reduced.
- The Θ transformation can be considered as some sort of compilation in the sense that all the clauses in IDB can be transformed only once independently of any query.

We have no room for a detailed presentation of the $T'_{P'}(\emptyset)$ operator definition but this definition is very similar to Definition 4.2. The main difference is that a rule is applied only when all its input variables are instantiated.

The proofs of Validity and Completeness are based on the same ideas but they are more complex.

The Termination proof is more problematic because here $\text{THNB}(P')$ is not finite, since it contains problems and solutions of an infinite length. However from the validity property we can conclude that the set of solutions $\text{Sol}(A)$ in M' is finite because they have a corresponding clause A in M which is finite. The finiteness of generated problems is less trivial. It is a consequence of the finiteness of the set of

solutions and of the finiteness of the number of distinct values for t'_{kx} and t'_x in the generated problems.

6. Conclusion

We have defined an efficient evaluation strategy to compute safe answers in the context of Indefinite Deductive Data Bases. This strategy is based on pure standard logic and does not apply any kind of default reasoning, as in [9, 4]. The basic idea was to transform the clauses into a set of rules whose execution according to a least fixpoint operator simulates forward chaining and backward chaining processes and focus the derivation on relevant clauses.

The strategy is very general but it can be improved for some particular cases. The simplest one is the case of clauses without recursive definitions, another one is the case of clauses with only linear recursions. Another direction for further investigations is to extend the Relational Algebra in order to efficiently compute formulas like $\mathrm{Sol}(B_1(x, z) \vee A(x, y)) \wedge \mathrm{Sol}(B_2(z, y) \vee A(x, y))$.

We can mention an interesting byproduct of the designed strategy. Since some sub-problems are of the form $\mathrm{Pb}(A(t_x, t_y) \vee Y)$, where Y is a variable, the strategy can compute answers to queries of the form $\mathrm{Pb}(q(t) \vee Y)$ which leads to solutions like $\mathrm{Sol}(q(a) \vee \mathrm{Man}(a))$, called *conditional answers* and whose intuitive meaning is: "if you are sure that a is not a Man then you can be sure that a satisfies the query".

References

[1] K. Apt, H. Blair and A. Walker, Towards a theory of declarative knowledge, in: J. Minker, ed., *Proc. Workshop on Foundations of Deductive Databases and Logic Programming* (1986).

[2] F. Bancilhon and R. Ramakrishnan, An amateur's introduction to recursive query processing strategies, in: *Proc. ACM PODS* (1986).

[3] C.-L. Chang and R.C. Lee, *Symbolic Logic and Mechanical Theorem Proving* (Academic Press, New York, 1973).

[4] S. Chi and L. Henschen, Recursive query answering with non-Horn clauses, in: *Proc. Conf. on Automated Deduction* (1988).

[5] K. Clark, Negation as failure, in: H. Gallaire and J. Minker, ed., *Logic and Data Bases* (Plenum Press, New York, 1978).

[6] R. Demolombe, A syntactical characterization of a subset of Domain Independent formulas, Technical Report, ONERA-CERT, 1982, also *J. ACM*, to appear.

[7] J.-M. Kerisit, La methode Alexandre: une technique de deduction, PhD thesis, Université de Paris 7, 1988.

[8] J. Minker and A. Rajasekar, A fixpoint semantics for Non-Horn Logic Programs, Technical Report UMIACS-TR-87-24, University of Maryland, IACS, 1987.

[9] J. Minker and A. Rajasekar, Procedural interpretation of non-Horn logic programs, in: *Proc. Conf. on Automated Deduction* (1988).

[10] R. Reiter, Nonmonotonic reasoning, *Annu. Rev. Comput. Sci.* **2** (1987).

[11] J. Rohmer, R. Lescoeur and J.-M. Kerisit, The Alexander method: a technique for the processing of recursive axioms in deductive databases, *New Generation Comput.* **4**(3) (1986).

[12] V. Royer, Modeling preference choices in Incomplete Deductive Databases, Technical Report, ONERA-CERT, 1988.

[13] L. Vieille, Recursive axioms in deductive databases: the query-sub-query approach, in: L. Kerschberg, ed., *Proc. 1st Internat. Conf. on Expert Database Systems* (Benjamin/Cummings, 1987).

[14] C. Zaniolo and D. Sacca, Rule rewriting methods in the implementation of the logic data language ldl, in: *IFIP WG2.6/8.1 Conf: The role of Artificial Intelligence in Databases and Information Systems* (1988).