Non-Determinism in Deductive Databases

Fosca Giannotti
CNUCE-CNR
Via Santa Maria 36, 56100 PISA, Italy
fosca@gmsun.cnuce.cnr.it

Dino Pedreschi
Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 PISA, Italy
pedre@dipisa.di.unipi.it

Domenico Saccà
Dipartimento di Sistemi, Università della Calabria
Rende (CS), Italy
2101sac@icsuniv.bitnet

Carlo Zaniolo
MCC
3500 West Balcones Center Dr.
Austin, Texas 78759
carlo@mcc.com

Abstract

This paper examines the problem of adding non-deterministic constructs to a declarative database language based on Horn Clause Logic. We revise a previously proposed approach, the *choice* construct introduced by Krishnamurthy and Naqvi, from the viewpoints of amenability to efficient implementation and expressive power. Thus, we define a construct called *dynamic choice*, which is consistent with the fixpoint-based semantics, cures the deficiencies of the former approach, and leads to efficient implementations in the framework of deductive databases. Also the new construct extends the expressive power of Datalog programs considerably, as it allows to express negation under Closed World Assumption, as well as a class of relevant deterministic problems.

1 Introduction

The importance of constructs for expressing non-determinism in logic programs is well-known; for instance, Prolog contains a spurious construct, called the *cut*, which is widely used to improve execution speed and to extend the expressive power of Horn Clauses – e.g., the cut allows the expression of negation-by-failure.

A clear need for non-determinism is also emerging in deductive databases as more experience is gained in programming with languages such as $\mathcal{LDL}$ [NT89]. While in most Prolog implementations the system stops after returning an answer, deductive databases tend to prefer the all-answer semantics, whereby the set of all answers is returned from the execution of a query. The all-answer semantics exacerbates the need for special constructs to deal with situations where the user is not interested in all possible answers: any answer will do. An important example of this situation, is when the user wants to see an arbitrary but unique sequence number assigned to each tuple in the answer, to serve as the object ID for a tuple [Zan89]. A second situation is exemplified by the following example: a new student must be given one (and only one) advisor. If the application of various qualification criteria fails to narrow the search to a single qualified professor, then an arbitrary choice from the eligible faculty will have to be made and recorded.

The desire to express applications as those above, where non-deterministic queries and modeling of non-deterministic behavior are needed, provided the original motivation for the introduction of the *choice* construct [KN88]. Only more recently, these authors became fully aware of the important role that these non-deterministic constructs can play in computing deterministic queries or transformations. This important facet of the problem is discussed in [AV89], where it is shown that simple deterministic functions which cannot be expressed in deterministic FO logic (with fixpoint) can be expressed once a non-deterministic operator called *witness* is added. An example of this added power follows from our previous observation that unique integers can be assigned to derived tuples once non-deterministic constructs are available – thus attaching an ordering to domains. It is known that deterministic languages on ordered domains are more powerful than deterministic languages on unordered domains [Imm87].

The objective of this paper is to revisit the issue of non-deterministic extensions to Horn-clause based languages from the viewpoints of expressive power and amenability to efficient implementation. We show that the current proposal, namely the *choice* proposal described in [KN88] and [NT89] suffers from undesirable properties that follow from its static nature. Thus, we introduce a new definition called *dynamic choice* that cures the problems of the construct proposed in [KN88]—which will hereafter be referred to as *static choice*. The declarative semantics of such a construct is based on the concept of stable models: the multiplicity of stable models for a given program provides a model theoretical characterization of non-determinism [SZ89].

We then turn to the problem of the operational semantics of choice, and address this problem in the framework of *backtracking fixpoint* procedure proposed in [SZ89]. In the case of definite Horn Clause programs augmented with dynamic choice, the fixpoint procedure is particularly simple and directly supplies the basis for a very efficient implementation.

Turning our attention to the issue of expressive power, we show that Datalog augmented with dynamic choice is strictly more expressive than Datalog augmented with the static choice constructs. In particular, dynamic choice can express negation under the Closed World Assumption. Hence, we conclude that in both the bottom-up and top-down procedural interpretations of Logic Programming appropriate non-deterministic operators can emulate some notions of negation, and conversely.

2 Basic notions

In this section, we summarize the basic notions of Horn Clauses logic, and its extensions to allow negative goals. Therefore we also briefly review the notion of stable models, which will be used later in the discussion. A more detailed discussion of these topics can be found in the referenced works [GL88].

A *term* is a variable, a constant, or a complex term of the form $f(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms. An *atom* is a formula of the language that is of the form $p(t_1, \ldots, t_n)$ where p is a predicate symbol of arity n. A *literal* is either an atom (positive literal) or its negation (negative literal). A *rule* is a formula of the language of the form

$$Q \leftarrow Q_1, \ldots, Q_m.$$

where Q is a atom (*head* of the rule) and $Q_1, \ldots, Q_m$ are literals (*body* of the rule). A term, atom, literal or rule is *ground* if it is variable free. A ground rule with empty body is a fact. A logic program is a set of rules. A rule without negative goals is called positive (a Horn clause); a program is called positive when all its rules are positive.

Let P be a program. Given two predicate symbols p and q in P, we say that p *depends on* q, written $p \prec q$, if either there exists a rule r in P such that p is the head predicate symbol of r and q occurs in the body of r, or there exists a predicate symbol s and a rule r in P such that $p \prec s$, s is the head predicate symbol of r and q occurs in the body of r. Moreover, two predicate symbols of P, say p and q, are *mutually recursive* if both $p \prec q$ and $q \prec p$. Finally, two atoms are said to be *mutually recursive* if their corresponding predicate symbols are mutually recursive; a rule is *recursive* if its head predicate symbol is mutually recursive with some predicate symbol occurring in the rule body.

Given a logic program P, the Herbrand universe for P, denoted H_P, is the set of all possible ground terms recursively constructed by taking constants and function symbols occurring in P. The Herbrand Base of P, denoted B_P, is the set of all possible ground atoms whose predicate symbols occur in P and whose arguments are elements from the Herbrand universe. A *ground instance* of a rule r in P is a rule obtained from r by replacing every variable X in r by a ground term in H_P. The set of ground instances of r are denoted by $ground(r)$; accordingly, $ground(P)$ denotes $\bigcup_{r \in P} ground(r)$. A *(Herbrand) model* M of P is a subset of B_P that makes each ground instance of each rule in P *true* (where a positive ground atom is *true* if and only if it belongs to M and a negative ground atom is *true* if and only if it does not belong to M—total models). A model of P is a *minimal model* if none of its proper subsets is a model. Each positive logic program has a unique minimal model which defines its formal declarative semantics.

Given a program P with model M let $ground_M(P)$ denote the program obtained from $ground(P)$ by

1. removing every rule having as a goals some literal $\neg q$ with $q \in M$
2. removing all negated goals from the remaining rules.

Since $ground_M(P)$ is a positive program, it has a unique minimal model. A model M of P is said to be *stable* when M is also the minimum model of $ground_M(P)$ [GL88]. A given program can have one or more stable (total) model, or possibly none. Positive and stratified programs are among those that have exactly one stable model [GL88]. The program $p \leftarrow \neg p$ is the simplest example of a program with no stable model. Of particular relevance to our discussion is the occurrence of multiple stable models, as in the following example:

$p \leftarrow \neg q$
$q \leftarrow \neg p$

This has two stable models: one where p is true and q is false, and the other where p is false and q is true. Every stable model for P is a minimal model for P.

3 Model-theoretical non-determinism

The problem of non-determinism in the framework of database logic languages was first addressed in [KN88], where an elegant solution based on the notion of minimal model and functional dependencies is proposed. According to [KN88, NT89], special goals, of the form *choice*$((X),(Y))$, are allowed in the rules to denote the functional dependency (FD) $X \rightarrow Y$ [Ull90]. Then the meaning of such programs is defined by its *choice models*, as discussed next.

Example 1. Consider the following program with choice.

a_st(St,Crs) ← takes(St,Crs), choice((Crs),(St)).
takes(andy,engl).
takes(ann, math).
takes(mark,engl).
takes(mark,math).

The choice goal in the first rule specifies that the *a_st* predicate symbol must associate exactly one student to each course. Thus the functional dependency $Crs \rightarrow St$ holds in the (choice model defining the) answer. Thus the above program has the following four choice models:

$M_1 = \{$ *a_st(andy, engl),a_st(ann, math)*$\} \cup X$,
$M_2 = \{$ *a_st(mark, engl),a_st(mark, math)*$\} \cup X$,
$M_3 = \{$ *a_st(mark, engl),a_st(ann, math)*$\} \cup X$,
$M_4 = \{$ *a_st(andy, engl),a_st(mark, math)*$\} \cup X$,

where X is the set of *takes* facts in Example 1.

A *choice predicate* is an atom of the form *choice*$((X),(Y))$, where X and Y are lists of variables (note that X can be empty). A rule having one or more choice predicates as goals will be called a *choice rule*, while a rule without choice predicates will be called a positive rule. Finally, a *choice program* is a program consisting of positive rules and choice rules.

The set of the choice models of a program with choice rules formally defines its meaning. The main operation involved in the definition of a choice model is illustrated by the previous example. Basically, any choice model $M_1, ..., M_4$ can be constructed by first removing the choice goal from the rule and computing the resulting *a_st* facts. Then the basic operation of enforcing the FD constraints is performed, by selecting a maximal subset of the previous *a_st* facts that satisfies the FD $Crs \rightarrow St$ (there are four such subsets). Unfortunately Example 1 hides the complexity involved in the general case. In particular, in this example all the variables appearing in the choice predicate are also contained in the head of the rule. To guarantee this property in the general case, a preliminary step is needed to construct a positive program called the extended version of P. Moreover, unlike Example 1 where the definition of *a_st* is not used anywhere else in the program, in general, several predicates might be dependent on those defined by the choice rules. Thus, after the selection of the maximal subset obeying the given FDs, a final step to compute the minimal model for the dependent predicates is needed. Let us formalize the operations just outlined. For now let us assume that P contain only one choice rule r, as follows:

$$r : A \leftarrow B, C.$$

where C denotes the conjunction of all choice goals and B is the conjunction of all remaining goals. The positive version of P denoted $PV(P)$ is the positive program obtained from P by eliminating all *choice* goals.

The *extended version* of P, denoted by $EV(P)$, is the positive program obtained from P by replacing r the two following rules:

$A \leftarrow B, extChoice(Z).$
$extChoice(Z) \leftarrow B.$

where Z are all the variables in the choice goals, listed in the order they occur in such goals. Thus, for the example at hand, we have:

a_st(St,Crs) ← *takes(St,Crs), extChoice(St,Crs).*
extChoice(St,Crs)← *takes(St,Crs).*

Let I_1 and I_2 be two interpretations from the Herbrand bases of two (possibly different) programs. Then we define I_1/I_2 as $\{A|$ A is in I_1 and the predicate symbol of A also occurs in I_2 $\}$. It turns out that, when $I_1/I_2 = I_2$, then I_1 is identical to I_2 modulo additional literals whose predicate symbols are not in I_2.

Proposition 1 *[SZ89] Let P be a choice program, M and N be the minimal models of $PV(P)$ and of $EV(P)$, respectively. Then $N/M = M$.*

Note that the only predicates of $EV(P)$ which do not occur in $PV(P)$ are those with symbol *extChoice*. Consider any of such predicates, say $extChoice(Z)$ with arity n. This predicate defines a n-ary database relation [Ull90] having as attribute the names of the variables in Z and as tuples the following set: $\{$ $(z)|extChoice(z)$ is in the minimal model of $EV(P)$ $\}$. We define the following set F of functional dependencies on the relation corresponding to $extChoice(Z)$:

$$F = \{X \rightarrow Y | choice((X),(Y)) \text{ is a goal of } r\text{—i.e., it is in } C\}$$

A *reduced version* of (the relation defined by) $extChoice(Z)$, denoted by $chosen(Z)$ is defined as any maximal subset of *extChoice* for which all the functional dependencies in F hold. Note that such reduced version is not necessarily unique and is empty if and only if $extChoice(Z)$ is empty.

We can now define a *reduced version* of P, denoted as $RV(P)$, as the program obtained from P by replacing each rule r by

$$r' : A \leftarrow B, chosen(Z).$$

where $chosen(Z)$ denotes an (arbitrarily chosen) reduced version of $extChoice(Z)$.

Definition 1 *Let P be a choice program. The minimal model of every reduced version of P is a* choice model *for P.*

From a pragmatic viewpoint, it is understood that a user will only want to see the answer to a query corresponding to one (arbitrarily chosen) choice model. From a formal viewpoint, however, the meaning of any given program is formally defined by the set of its choice models. Thus, for instance, the set of choice models of the program in Example 1 corresponds to the minimal models of the following transformed program:

a_st(St,Crs) $\leftarrow$ *takes(St,Crs), chosen((Crs),(St)).*

where the extension of *chosen* is one the following four sets.

{ *chosen(andy, engl),chosen(ann, math)* }
{ *chosen(mark, engl),chosen(mark, math)* }
{ *chosen(mark, engl),chosen(ann, math)* }
{ *chosen(andy, engl),chosen(mark, math)* }

These four sets were derived by first computing the relation *extChoice* from the following extended version of the program:

a_st(St,Crs) $\leftarrow$ *takes(St,Crs), extChoice(Crs,St).*
extChoice(Crs,St) $\leftarrow$ *takes(St,Crs).*

and then deriving the maximal subsets of *extChoice* satisfying the dependency $Crs \rightarrow St$.

An extrapolation of these definitions to the case involving several choice rules is presented in [KN88, NT89]. We will not discuss this problem now, since various definitional and computational problems of the current definition must be addressed first.

4 Choice in recursion

The definition of choice models presented in [KN88] is not conducive to effective implementation. Take for instance, the following recursive definition describing nodes reachable from a given node a and their distance from a on a graph g.

Example 2.

p(a,0).
p(Y, J) $\leftarrow$ *p(X,I), g(X,Y), J= I+1,choice((Y), (J)).*

Observe that, in this example, any choice model is finite, even when graph g has cycles. Computing *extChoice* using the [KN88] rules described in the previous section we obtain:

p(a,0).
p(Y, J) $\leftarrow$ *p(X,I), g(X,Y), J= I+1, extChoice(Y, J).*
extChoice(Y, J)$\leftarrow$ *p(X,I), g(X,Y), J= I+1.*

Observe that the extensions of predicates p and *extChoice* are identical; furthermore, they are both infinite when the graph g is cyclic. Thus, we cannot use the static choice definition [KN88] to effectively compute choice models. Instead, we need the ability to compute a choice model, without having to first compute the infinite *extChoice* relation.

A second and more fundamental problem with choice models is that they fail to deliver models that maximally satisfy the given functional dependencies. Again, this anomaly pertains to recursive definitions. With reference to the above example, assume that graph g contains the edges $g(a,b)$ and $g(b,b)$. Hence the minimal model of the extended program contains the following p- and *extChoice* pairs: $(a,0),(b,1),(b,2),\ldots$. Considering now the reduced version of the program, and selecting the set $\{(a,0),(b,1)\}$ as the extension of predicate *chosen*, we find that the resulting choice model contains the p-pairs $\{(a,0),(b,1)\}$. But if we use $\{(a,0),(b,2)\}$ as the extension of *chosen*, the resulting choice model contains only $(a,0)$, since $(b,2)$ cannot belong to the extension of p if $(b,1)$ does not belong to the same. Thus one choice model properly contains another: a situation which contradicts the expected property of maximality of choice models. Also observe that these problems remain when finite domains are considered. For instance, it is simple to construct a finite version of the example above, by encoding the successor relation for the first k integers by $k-1$ facts. The resulting Datalog program still suffers from the same problems.

Therefore we need a new notion of choice models to guarantee (i) the maximality of the resulting sets with respect to the given FDs, and (ii) the availability of an effective computation procedure. The next sections propose a solution that satisfies both these requirements. Basically, static choice assumes that a single global selection is performed "at the end" of recursion, i.e., at the end of the deduction process. Instead, we will adopt an approach where many local choices are performed dynamically "during" recursion, i.e. interleaved with the deduction steps as to restrict the scope of later choices.

5 Choice by negation

An alternative approach to define non-determinism in a declarative fashion was proposed in [SZ89]. According to said proposal, programs with choice are transformed into programs with negation which exhibit a multiplicity of stable models. Each stable model corresponds to an alternative set of answers for the original program. Following [SZ89], therefore, let P be a choice program (which, for simplicity of exposition, we will initially assume contains only one choice rule). Then, the *stable version* of P, denoted by $SV(P)$, is the program with negation obtained from P by the following two transformation steps:

1. In each choice rule of P, say

$$r : A \leftarrow B, C.$$

   where C denotes the conjunction of all choice goals and B denotes the conjunction of all remaining goals, replace C with the atom $chosen(Z)$, where Z are all the variables in the choice goals, listed in the order they occur:

$$r' : A \leftarrow B, chosen(Z).$$

   then add the following rule:

$$chosen(Z) \leftarrow B, \neg diffChoice(Z).$$

2. for each goal $choice((X),(Y))$ in C, add a new rule:

$$diffChoice(Z) \leftarrow chosen(U), Y \neq Y'.$$

where U is a list of variables obtained from Z by replacing every variable y which is in Z but not in X by a new variable y', and $Y \neq Y'$ is the inequality predicate between lists. Thus, $Y \neq Y'$ is true when, for some i, the i-th component of Y is different from the i-th component of Y'. (Obviously, given the decomposition rule of FDs [Ull90], $choice((X),(Y))$ can be replaced by several $choice((X),(y))$—one for each component y of Y. Then several $diff$ rules would be generated. The list notation provides a more succinct equivalent.)

When the given program P is such that none of its choice rules is recursive, then P and its stable version are semantically equivalent in the sense that the set of choice models of P is equivalent to the set of stable models of $SV(P)$ [SZ89]:

Proposition 2 *Let P be a choice program such that every choice rule is non-recursive. Then $SV(P)$ has at least one stable model.*

Proposition 3 *Let P be a choice program, containing no recursive choice rules. Then*

1. *for each choice model M for P, there exists a stable model N of $SV(P)$ such that $N/M = M$, and*

2. *for each stable model N for $SV(P)$ there exists a choice model M for P such that $N/M = M$.*

The following is the stable version of Example 1.

a_st(St,Crs) $\leftarrow$ *takes(St,Crs), chosen(Crs,St).*
chosen(Crs,St) $\leftarrow$ *takes(St,Crs),*$\neg$ *diffChoice(Crs,St).*
diffChoice(Crs,St) $\leftarrow$ *chosen(Crs,* $\overline{St}$ *),St* $\neq$ $\overline{St}$*.*
takes(andy,engl).
takes(ann, math).
takes(mark,engl).
takes(mark,math).

Let us turn now to choice in recursive predicates, which was not discussed in [SZ89], and consider the stable version of Example 2.

p(a,0).
p(Y, J) $\leftarrow$ *p(X,I), g(X,Y), J=I+1,chosen(Y, J).*
chosen(Y,J) $\leftarrow$ *p(X,I), g(X,Y), J=I+1,* $\neg$ *diffChoice(Y,J).*
diffChoice(Y,J) $\leftarrow$ *chosen(Y,J'),J* $\neq$ *J'.*

If the graph relation g is defined by $g(a,b)$ and $g(b,b)$, then there is only one stable model in this example, which contains the p-facts $p(a,0)$ and $p(b,1)$. This is a maximal sets of p-facts satisfying the given functional dependency.

As the last example suggests, this new characterization of non-determinism provides a ready-made solution to problem (i) listed at the end of section 4. Indeed, choice goals in rules can now be viewed as a shorthand for mutually recursive predicates with negation, giving rise to a new semantics of choice. This is stated by the following:

Definition 2 *Let $SV(P)$ be the stable version of a program P with choice constructs. The stable models of $SV(P)$ are named* stable choice models *of P. The meaning of P is defined by the set of its stable choice models.*

This new characterization of choice overcomes the deficiencies of static choice in treating choice within recursion. Also observe that the generalization to the case of several choice rules in the program is trivial. All is needed is a to assign different names to the distinguished *chosen* and *diffChoice* predicates generated from each rule (e.g., by the addition of a subscript).

The following result points out the declarative meaning of stable choice models as maximal sets satisfying the functional dependencies. Thus the new semantics extends the expected meaning to cope with any situation.

Proposition 4 *Given a program P with choice constructs, consider its extended version $EV(P)$ (i.e., with extChoice predicates) and its stable version $SV(P)$ (i.e., with chosen and $diffChoice$ predicates). For each stable model N of $EV(P)$ there exists a stable model M of $SV(P)$ such that the following property holds. Let $r_j : H \leftarrow B, C$ be a choice rule in P. Let E_j the set of extChoice$_j$-facts in N, and C_j be the set of chosen$_j$-facts in M. Then C_j is a maximal subset of E_j which satisfies the functional dependencies stated by the choice construct C, and such that if chosen$_j(x)$ is true in N then the body of the associated chosen$_j$-rule is true in N.*

Proof. We develop here the proof for programs P with a single choice rule, i.e.:

$$P = P' \cup \{H \leftarrow B, choice((X), (Y)).\}$$

where P' is a definite program. The argument directly extends to the general case.

We have:

$$\begin{aligned} EV(P) = P' \cup \{ \; & H \leftarrow B, extChoice(X, Y). \,, \\ & extChoice(X, Y) \leftarrow B.\} \end{aligned}$$

$$\begin{aligned} SV(P) = P' \cup \{ \; & H \leftarrow B, chosen(X, Y). \,, \\ & chosen(X, Y) \leftarrow B, \neg diffChoice(X, Y). \,, \\ & diffChoice(X, Z) \leftarrow chosen(X, Y), Y \neq Z.\} \end{aligned}$$

We first show that any stable model of $SV(P)$ satisfies the functional dependency $X \rightarrow Y$. Assume, by contradiction, that both $chosen(a, b)$ and $chosen(a, c)$ belong to C_j, with $b \neq c$. $chosen(a, b) \in C_j$ implies that $diffChoice(a, c) \in N$, using the ground rule:

$$diffchoice(a, c) \leftarrow chosen(a, b), b \neq c.$$

Analogously, $chosen(a, c) \in C_j$ implies that $diffChoice(a, b) \in N$, using the ground rule:

$$diffchoice(a, b) \leftarrow chosen(a, c), b \neq c.$$

Hence, the rule

$$chosen(a, b) \leftarrow B, \neg diffChoice(a, b)$$

cannot belong to the positive version $ground_N(SV(P))$ of $SV(P)$ with respect to N, as $\neg diffChoice(a, b)$ is false in N, and thus the fact $chosen(a, b)$ cannot be true in the minimal model of $ground_N(SV(P))$, as that rule is the only one for inferring $chosen(a, b)$. Analogously, we can conclude that $chosen(a, c)$ does not belong to the minimal model of $ground_N(SV(P))$. Hence N is not a stable model of $SV(P)$, as it is not equal to the minimal model of $ground_N(SV(P))$.

To complete the proof, we need to show that C_j is maximal. Again, assume by contradiction that there exists a fact $chosen(a, b)$ such that: (i) for a ground rule $extChoice(a, b) \leftarrow B'$ of $EV(P)$, B' is true in M, and hence $extChoice(a, b)$ is true in M; (ii) $C_j \cup \{chosen(a, b)\}$ respects the FD $X \rightarrow Y$, and (iii) B' is true in N. By the fact that N is a stable model of $SV(P)$ we conclude that no fact of the kind $diffChoice(a, Y)$ is in N for some Y, as no fact of the kind $chosen(a, Z)$ is in C_j for some Z, and this is one of the premises of the $diffChoice$-rule. This also implies that the rule:

$$chosen(a, b) \leftarrow B'.$$

belongs to the positive version $ground_N(SV(P))$ of $SV(P)$ w.r.t. N, as $\neg diffChoice(a, c)$ is true in N. The above rule allows us to conclude that $chosen(a, b)$ is true in the minimal model of $ground_N(SV(P))$, as, by assumption, B' is true in N. This contradicts the fact that N is a stable model of $SV(P)$. □

It is interesting to notice how Proposition 4 applies to Example 2 describing nodes reachable from a given node a and their distance, on a graph g defined by the edges $g(a, b)$ and $g(b, b)$. In this case, the extension of *extChoice* in the unique stable model of $EV(P)$ is the infinite set $\{(a, 0), (b, 1), (b, 2), \ldots\}$, whereas the extension of *chosen* in the unique stable model of $SV(P)$ which satisfies the hypothesis of Proposition 4 is the set $\{(b, 1)\}$, which yields $\{(a, 0), (b, 1)\}$ as the (expected) extension of predicate p.

Therefore, a suitable construct for a non-deterministic pruning operator for deductive databases should retain the syntax proposed in [KN88, NT89] but adopt a semantics based on the equivalence with negative programs just described.

6 Implementation of choice

In this section we address the issue of actually computing stable choice models. We use the non-deterministic procedure, called Backtracking Fixpoint, that was introduced in [SZ89] for determining the total stable models of a negative program. In the simpler case of choice programs, it reduces to a much simpler fixpoint procedure, which is conducive to efficient implementation (the generic results regarding the computational complexity of stable models notwithstanding).

Let us first adapt the *Backtracking Fixpoint Procedure* of [SZ89] to the case of general choice programs. The program to which this procedure is applied is the the stable version of the general choice program under consideration and for sake of notation simplicity is demoted by P rather than $SV(P)$ as usual. Since the choice program is positive, the negative rules of the stable version P are only those of this form *chosen rules*:

$$r_j : chosen_j(Z) \leftarrow B, \neg diffChoice_j(Z).$$

thus they contain exactly one negative literal in the body.

In the procedure we use the transformation S_P defined as follows. Let T_P be the immediate consequence transformation and $T_P^\infty(\emptyset)$ be its least fixpoint. Moreover, let P' denote the positive program obtained from P by viewing each negative literal $\neg p(A)$ as a new positive literal with predicate symbol $\neg p$. Given a set of negative ground literals X (regarded as facts), we define $S_P(X) = T_{P' \cup X}^\infty(\emptyset) - X$ — i.e., the positive literals in the least fixpoint (and minimum model) of P' given a fixed set of negative ground literals X. The Backtracking Fixpoint Procedure is presented next:

```
begin
$M_0 := S_P(\emptyset)$; $\tilde{M}_0 := \emptyset$; $stable$ := true;
$if\ C_i = \emptyset\ then$
        $stable$ := true
$else$
        $L_i := order(C_i)$;
$endif$;
$while$ not $stable$ and $i > 0$ $do$
         $if\ L_i \neq \emptyset\ then$
                 take from $L_i$ the head rule, say $chosen_j(z) \leftarrow B, \neg diffChoice_j(z))$ ;
                 $\tilde{M}_i := \tilde{M}_{i-1} \cup \{\neg diffChoice_j(z)\}$;
                 $M_i := S_P(\tilde{M}_i)$;
                 $if\ conflict(M_i, \tilde{M}_i)\ then$
                         $i := i + 1$;
                         $if\ C_i = \emptyset\ then$
                                 $stable$ := true
                         $else$
                                 $L_i := order(C_i)$
                         $endif$
                 $endif$
         $else$
                 $i := i - 1$
         $endif$
         $od$
$if\ stable\ then$
         $output$ $M_{i-1}$ "is a stable model"
$else$
         $output$ "No stable models"
$endif$
$end.$
```

At the generic level i, C_i denotes the set of all rule instances in $ground(P)$, having the form:

$$chosen_j(z) \leftarrow B, \neg diffChoice_j(z).$$

and such that:

- each literal in B is in M_{i-1},
- neither $diffChoice_j(z)$ is in M_{i-1} nor $\neg diffChoice_j(z)$ is in $\tilde{M}_{i-1}$,
- neither $chosen_j(z)$ is in M_{i-1} nor $\neg chosen_j(z)$ is in $\tilde{M}_{i-1}$.

The procedure starts at level 0 by determining all ground predicates that can be inferred using only positive ground literals. In terms of the S_P notation, $M_0 = S_P(\emptyset)$ is computed. No negative ground literal is assumed: so we set $\tilde{M}_0 = \emptyset$. Then we move up to level 1. Here, we consider the set C_1 and, more in general, the set C_i. If C_i is empty, then we are done, and M_{i-1} is a stable model. Otherwise, all the rules in C_i are inserted into the list L_i in an arbitrary order (see function *order*). Then the first entry (a rule with head *chosen*) is removed from L_i and taken into consideration. Then we add $\neg diffChoice_j(Z)$ to the set $\tilde{M}_{i-1}$ of all negative ground literals that have been assumed up to level $i-1$. In this way, we obtain $\tilde{M}_i$, the set of all negative ground literals assumed up to level i; we use such negative literals to infer all possible positive ground literals through the program P, i.e., we compute M_i as $S_P(\tilde{M}_i)$. At this point, we invoke the function $conflict(M_i, \tilde{M}_i)$ which returns true only when there exists some Q in M_i such that $\neg Q$ is in $\tilde{M}_i$. If there is no conflict, then we move up to the next level and we set up the next list L_{i+1}; otherwise, we remain at level i and we retry with another rule in L_i in the next step of the while iteration. If all rules of L_i happens to be already used (thus L_i is empty) then we backtrack to the level $i-1$ and select another rule for this level. If we eventually get back to level 0, no more alternatives are possible and the procedure stops by declaring that the program has no stable models.

Although the procedure is guaranteed to terminate for finite domains, its time complexity is in general exponential. However, in the case of choice programs, the above fixpoint procedure simplifies dramatically. To see this point, suppose that we choose a rule

$$chosen(z) \leftarrow B, \neg diffChoice(z)$$

from C_i. This implies that in $\tilde{M}_i$ we add $\neg diffChoice(z)$, while the further fixpoint saturation adds $chosen(z)$ which will forbid to derive the fact $diffChoice(z)$. But the only way to create a contradiction is to derive a fact $diffchoice(z)$. As a consequence the fixpoint procedure never backtracks since the non-deterministic choices never produce a contradiction. Moreover the sequence of sets M_i and $\tilde{M}_i$ is monotonically increasing and we do not need to keep indexed versions of the sets; two monotonically increasing variables M and $\tilde{M}$ will suffice.

These observations are summarized by the following proposition:

Proposition 5 *Let P be a choice program. Then the Backtracking Fixpoint procedure applied to $SV(P)$ has the following properties:*

1. *it never backtracks and never outputs "no stable models";*
2. *for each level i, $M_i \subseteq M_{i+1}$ and $M_i \subseteq M$, where M is a stable model of $SV(P)$.*

The procedure for computing stable choice models can be simplified, by exploiting Proposition 5. Given $M \subseteq B_P$, let C_M be the set of *chosen* rules $chosen(x) \leftarrow B, \neg diffChoice_i(x)$ from $ground(P)$ such that: (i) all the atoms in the conjunction B belong to M, and (ii) the atom $diffChoice_i(x)$ does not belong to M. In other words, C_M is the set of *chosen* rules which can be fired consistently with M.

The simplified procedure for *computing stable choice models* is as follows:

```
begin
$M := S_P(\emptyset)$; $\tilde{M} := \emptyset$;
while not $C_M = \emptyset$ do
    select any $r : chosen_j(z) \leftarrow B, \neg diffChoice_j(z))$ from $C_M$;
    $\tilde{M} := \tilde{M} \cup \{\neg diffChoice_j(z)\}$;
```

```
        $M := S_P(\tilde{M})$;
        od;
    output $M$ "is a choice model"
    end.
```

The procedure operates by choosing, at each iteration, an arbitrarily instantiated *chosen* rule rj in C_i, and adding the associated $\neg diffChoice(z)$ literal to $\tilde{M}$. Then the consequences of the operated choice are derived by saturating the positive rules. It is worth noting that among such rules there are also the $diffChoice$ rules enabled by the choice, which will prevent future choices which would violate the functional dependency.

It is easy to see that the above procedure can be easily implemented, thus providing a viable extension to existing logic database languages, such as $\mathcal{LDL}$ [NT89]. As a final remark, a realistic implementation of the above algorithm does not actually compute the $diffChoice$ relation, but simply records the *chosen* tuples as they are generated in order to discard or not the future candidates. Therefore, choice can be implemented, as it is done in $\mathcal{LDL}$ [Chi90] by simply memoing old values of *chosen* [Die87].

The behavior of the procedure is illustrated by the following two examples. Consider first the student-course example of Section 3.

a_st(St,Crs) $\leftarrow$ *takes(St,Crs), chosen(Crs,St).*
chosen(Crs,St) $\leftarrow$ *takes(St,Crs),*$\neg$ *diffChoice(Crs,St).*
diffChoice(Crs,St) $\leftarrow$ *chosen(Crs,* $\overline{St}$ *),St* $\neq$ $\overline{St}$.
takes(andy,engl).
takes(ann, math).
takes(mark,engl).
takes(mark,math).

Let M_i and $\tilde{M}_i$ denote the content of variables M and $\tilde{M}$ at the i-th iteration of the procedure. The following are the partial results of the above procedure.

$\tilde{M}_0 = \emptyset$
$M_0 = \{$*takes(andy,engl), takes(ann, math), takes(mark,engl), takes(mark,math)*$\}$

$\tilde{M}_1 = \{\neg diffChoice(andy, engl)\}$
$M_1 = M_0 \cup$ $\{$*a_st(andy,engl), chosen(andy,engl),diffChoice(mark,engl)*$\}$

$\tilde{M}_2 = \tilde{M}_1 \cup \{\neg diffChoice(mark, math)\}$
$M_2 = M_1 \cup$ $\{$*a_st(mark,math), chosen(mark,math),diffChoice(ann,math)*$\}$

The procedure halts with M_2 as a result, as C_{M_2} is empty.

Consider next the "reachable points" example of section 4.

g(a,b).
g(b,b).
p(a,0).
p(Y, J) $\leftarrow$ *p(X,I), g(X,Y), J=I+1,chosen(Y, J).*
chosen(Y,J) $\leftarrow$ *p(X,I), g(X,Y), J=I+1,* $\neg$ *diffChoice(Y,J).*
diffChoice(Y,J) $\leftarrow$ *chosen(Y,J$'$),J* $\neq$ *J$'$.*

The partial results of the procedure are described next.

$\tilde{M}_0 = \emptyset$
$M_0 = \{g(a,b),\ g(b,b),\ p(a,0)\}$

$\tilde{M}_1 = \{\neg diffChoice(b,1)\}$
$M_1 = M_0 \cup \{p(b,1),\ chosen(b,1),\ diffChoice(b,J)_{J \neq 1}\ \}$

Again, the procedure halts as no rule instance is in C_{M_1}. It is worth noting that the procedure delivers the unique stable choice model of the program. Observe here that we have used a suitable compact notation for recognizing all true *diffChoice* literals so that the problem of infinite domains is overcome. In the actual implementation, this is also easy to do by not storing *diffChoice* and $\neg$*diffchoice* explicitly, and simply deriving their values from those of *chosen* as needed [Chi90].

7 Negation by choice

We will now investigate expressiveness issues for the dynamic choice construct. With this aim in mind, we will modify the fixpoint procedure for choice programs presented in the previous section and show that this implementation of the dynamic choice construct is powerful enough to model negation under the *Closed World Assumption* for Datalog programs.

Informally, the modified procedure, that we name DCF for *dynamic choice fixpoint* behaves as follows. Given a choice program P and its stable version $SV(P)$, call **C** the set of *chosen* rules in $SV(P)$, **D** the set of *diffChoice* rules in $SV(P)$, and **O** the set of the remaining rules in $SV(P)$.

Then, the DCF procedure is as follows:

1. find the fixpoint of the **O** part;
2. while there exists an enabled *chosen* rule in **C**, repeat:
   (a) choose a rule r in **C** and execute it;
   (b) execute all rules in **D** enabled by r;
3. repeat steps 1 and 2 until no rule is enabled.

Notice that we used the term "execute" to mean the ordinary bottom-up computation mechanism of asserting the head of a rule whenever its body is true. The idea underlying the DCF procedure can be explained as follows. There are two modes of operation: a saturation mode and a choice mode. In the saturation mode, the consequences of the original rules are computed by an ordinary fixpoint mechanism. When nothing more can be deduced, the procedure switches to the choice mode. In the choice mode, a *chosen* rule together with the associate *diffChoice* rules are executed, until no more choices can be made. Then the procedure switches to the saturation mode again, and the process continues until a fixpoint is reached.

In other words, when DCF is in the choice mode, it makes all the choices that are compatible with the functional dependency constraints, before it switches to the saturation mode again. The following code formalizes the DCF procedure.

DCF Procedure

```
begin
$M := \emptyset$; $\tilde{M} := \emptyset$;
repeat
    $OldM := M$;
    $M := S_O(M)$;
    while not $C_M = \emptyset$ do
        $\tilde{M} := \tilde{M} \cup \{\neg diffChoice_i(z) | r : chosen(z) \leftarrow B, \neg diffChoice_i(z) \in C_M\}$;
        $M := M \cup \{chosen(z) | r : chosen(z) \leftarrow B, \neg diffChoice_i(z) \in C_M\}$;
        $M := S_D(M)$;
        od;
until $M \neq OldM$;
output $M$ "is a choice model"
end.
```

We claim that the DCF procedure is correct with respect to the stable choice model semantics of the program, in the sense that every model it produces is a stable choice model for our program. This claim can be easily established by observing that an *early choice* is clearly correct with respect to the functional dependencies, although it may inhibit possible later choices. This implies that DCF cannot compute every stable choice model of a program, but only some *preferred* ones.

The importance of the DCF procedure lies in the fact that it allows to compute efficiently some relevant *deterministic* problems. A remarkable example is the implementation of negation under Closed World Assumption for (finite domain) Datalog programs. The following program defines *not_p* to be the complement of a relation p with respect to a universal relation u which contains all the domain elements.

Example 3.

```
p(a).
p(b).
u(a).
u(b).
u(c).

not_p(X) ← comp_p(X,2).

comp_p(X,0) ← p(X).
comp_p(X,I) ← aux(X,I), choice((X),(I)).

aux(X,1) ← comp_p(X,0).
aux(X,2) ← u(X),comp_p(_,1).
```

By applying the DCF procedure to the above program, we obtain a set of answers where *comp_p*$(x, 2)$ holds if and only if x is not in the extension of p, i.e., $x = c$ in the example. This behavior is due to the fact that the extension of *aux* must obey the functional dependency $X \rightarrow I$, and that DCF operates early choices which binds to 1 all the elements in the extension of p. This implies that all the elements which do not belong to p will be chosen in the next saturation step, and hence bound to 2.

More precisely, in the first saturation phase the facts $comp\_p(x, 0)$ and $aux(x, 1)$ are inferred, for x in the extension of relation p. In the following choice phase the facts $chosen(x, 1)$ are chosen, again for x in the extension of p, as all possible choices are operated. In the second saturation phase the facts $comp\_p(x, 1)$ are inferred for x in the extension of p, and the facts $aux(x, 2)$ for every x in the universe. In the following choice phase, the facts $chosen(x, 2)$ are chosen in a maximal way to satisfy the FD, i.e., for x *not* in the extension of p, as all x's in p have been chosen with tag 1. In the third saturation step, the extension of not_p becomes the complement of p with respect to u.

It is important to observe that the above construction only works properly under the assumption that relation p is completely materialized in the first saturation phase of the procedure. This observation leads us to conclude that the proposed procedure can be adopted to compute *stratified negation* as well. In fact, if the relation p to be complemented belongs to a lower stratum than that of the code for not_p, an iterated application of DCF will correctly behave as in the example above, where p is an extensional relation.

This result offers the opportunity for a comparison between top-down logic languages like Prolog, and bottom-up logic database language like $\mathcal{LDL}$. In both cases, by adding a non-deterministic mechanism, it is possible to enhance the expressiveness of the pure language to capture useful forms of negation. In Prolog, the *cut* operator enables the implementation of negation-by-failure, whereas, in Datalog, the dynamic choice operator enables the implementation of negation under CWA.

The set of answers computed by the DCF procedure for the above program is actually a stable choice model for this program. Nevertheless, there exist stable choice models of the program where that property does not hold. For instance the set containing $aux(a, 1), aux(b, 2), aux(c, 2)$ is also a stable model of the stable version of Example 3. In these undesired models not computed using the choice policy of DCF, not_p is no longer the complement of p.

In other words, the DCF procedure is a *less non-deterministic* variation of the procedure presented in the previous section. With respect to the general procedure, DCF computes a subset of preferred stable choice models, those corresponding to the early choice policy. This characteristic makes DCF effective in dealing with relevant deterministic problems. From the point of view of efficiency, DCF is even more efficient than the procedure of Section 6, since early choices restrict the generation of new inferred facts, and the scope for future choices. Furthermore, it should be clear that the DCF approach is particularly suitable to the bottom-up framework of deductive databases inasmuch as DCF can simply be implemented via simple memoing/check operations [Die87].

As a further example, consider again Example 2 for computing nodes reachable from a given node a and their distance. By interpreting such a program with DCF we obtain a set of pairs (x, n), where n is the length of the minimum path from a to x, for each node x reachable from a in the graph.

The increased computational efficiency and expressive power that follows from the DCF approach make it very attractive as a basis for a practical operational semantics of for non-deterministic pruning operators in deductive databases.

8 Conclusions

In this paper, we have studied the problem of defining declarative constructs to express non-determinism in deductive databases. These constructs are important for improving both efficiency and the expressive power of logic based languages—e.g., by enabling the enforcement of functional dependency constraints in derived relations. We have shown that a simple declarative definition of non-deterministic constructs can be based on the stable model semantics for negative programs, and, using this approach, we have proposed the notion of dynamic choice that improves on the (static) notion of choice proposed in [KN88], with respect to expressive power and efficiency of implementation.

There is an interesting similarity between non-determinism in deductive databases and that in the traditional top-down computational framework of Prolog. In [GPZ89] it has been shown that the declarative aspects of the *cut* operator can be modeled in a way similar to that in which we modeled choice using negation. In Prolog, the nature of the inverse relationship is also clear: it is simple to express negation using the operational semantics of the cut. In this paper, we have examined the nature of that relationship for deductive databases, and we have shown that negation in Datalog can be expressed using dynamic choice under the modified fixpoint computation DCF.

In the course of this investigation new issues have emerged that we left open for further research. For instance, the stable choice model semantics justifies DCF, but it does not characterizes DCF completely. Thus, the problem remains to identify a fully declarative semantics for DCF. Alternatively, it is not clear whether one can emulate the behavior of DCF under the stable choice model semantics. Finally, it would be useful to compare dynamic choice with DCF with the *witness* operator of [AV89]. The witness mechanism appears a more non-deterministic operator, as it operates choices at each fixpoint iteration, while DCF operates choices at each saturation. We are currently working on these problems.

References

[AV89] Abiteboul, S., and Vianu, V., "Fixpoint extensions of first-order logic and Datalog-like languages", Proc. 4th Symp. on Logic in Computer Science (LICS), IEEE Computer Press, pp. 71-89, 1989.

[Chi90] Chimenti, D. et al., "The $\mathcal{LDL}$ System Prototype," *IEEE Journal on Data and Knowledge Engineering*, Vol. 2, No. 1, pp. 76-90, 1990.

[DW90] Debray, S.K., and Warren, D.S., "Towards banishing the cut from Prolog", IEEE Trans. on Software Engineering, Vol. 16, No. 3, pp. 335-349, 1990.

[Die87] Dietrich, S.W., "Extension Tables: Memo Relations in Logic Programming", *Fourth IEEE Symposium on Logic Programming*, MIT Press, pp. 264-272, 1987.

[GL88] Gelfond, M., and Lifschitz, V., "The stable model semantics for logic programming", Proc. 5th Int. Conf. and Symp. on Logic Programming, MIT Press, pp. 1070-1080, 1988.

[GPZ89] Giannotti, F., D. Pedreschi and C. Zaniolo, "Declarative Semantics for Pruning Operators in Logic Programming", Proc. NACLP90 Workshop on Logic Programming and Non-Monotonic Reasoning, Austin, Tx, Nov 1,2, 1989.

[KN88] Krishnamurthy, R., and Naqvi, S.A., "Non Deterministic Choice in Datalog", *Proc. 3rd Int. Conf. on Data and Knowledge Bases*, Morgan Kaufmann Pub., Los Altos, pp. 416-424, 1988.

[NT89] Naqvi, S.A., and Tsur, S., *A Logical Data Language for Data and Knowledge Bases*, Computer Science Press, New York, 1989.

[Imm87] Immerman, N., *Languages which Capture Complexity Classes*, SIAM J. Computing, 16,4, pp. 760-778, 1987.

[SZ89] Saccà, D., and Zaniolo, C., "Stable models and non determinism in logic programs with negation", In *Proc. Symp. on Principles of Database Systems PODS'89*, 1989.

[Ull90] Ullman, J.D., *Principles of Database and Knowledge-Base Systems*, Vol. 1 and 2, Computer Science Press, Rockville, Md., 1989.

[Zan89] Zaniolo, C. "Object Identity and Inheritance in Deductive Databases: an Evolutionary Approach," *Proc. 1st Int. Conf. on Deductive and O-O Databases*, Dec. 4-6, 1989, Kyoto, Japan.