Datalog with non-deterministic choice computes *NDB-PTIME*

L. Corciulo, F. Giannotti, D. Pedreschi

Computer Science/Department of Algorithmics and Architecture

Report CS-R9364 September 1993

CWI is the National Research Institute for Mathematics and Computer Science. CWI is part of the Stichting Mathematisch Centrum (SMC), the Dutch foundation for promotion of mathematics and computer science and their applications.
SMC is sponsored by the Netherlands Organization for Scientific Research (NWO). CWI is a member of ERCIM, the European Research Consortium for Informatics and Mathematics.

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

Datalog with Non-deterministic Choice Computes *NDB-PTIME*

Luca Corciulo[2], Fosca Giannotti[1], and Dino Pedreschi[2]

[1] *CNUCE Institute of CNR, Via S. Maria 36, 56125 Pisa, Italy*
e-mail: fosca@cnuce.cnr.it

[2] *Dipartimento di Informatica, Univ. Pisa, Corso Italia 40, 56125 Pisa, Italy*
e-mail: pedre@di.unipi.it

Abstract

This paper addresses the issue of non deterministic extensions of logic database languages. After providing a quick overview of the main proposals in the literature, we concentrate on the analysis of the *dynamic choice* construct from the point of view of the expressive power. We show how such construct is capable of expressing several interesting deterministic and non deterministic problems, such as forms of negation, and ordering. We then prove that Datalog augmented with the dynamic choice expresses exactly the non deterministic time-polynomial queries. We thus obtain a complete characterization of the expressiveness of the dynamic choice, and conversely achieve a characterization of the class of queries *NDB-PTIME* by means of a simple, declarative and efficiently implementable language.

AMS Subject Classification (1991): 68N17
CR Subject Classification (1991): D.1.6, F.4.1, I.2.3, F.1.2, H.2.3
Keywords & Phrases: Deductive databases, logic–based languages, non determinism, expressive power
Note: This paper has been revised during the stay of two of the authors, Giannotti and Pedreschi, at CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands. This paper will appear in Proc. DOOD'93, Third Int. Conf. on Deductive and Object–oriented Databases, Springer–Verlag, 1993.

1. Introduction

Two main classes of logic database languages have been proposed in the literature. One is the class of *FO* database languages, based on the relational calculus, i.e. on the first-order logic interpretation of the relational data model. The other one is the class of Datalog languages, a subset of the logic programming paradigm which supports and extends the basic mechanisms of the relational data model.

Indeed, both classes served as the basis of several extensions, aimed at enhancing the expressive power of the relational data model. For instance, the set of queries expressed by the relational algebra is strictly included in that of the *fixpoint queries* (the transitive closure is a fixpoint query which is inexpressible in the relational algebra), whereas it is well known that every fixpoint query can be expressed in *FO* extended with an inflationary fixpoint operator, or equivalently in Datalog extended with inflationary negation.

Unfortunately, the expressiveness achieved by this kind of deterministic extensions of logic database languages is not satisfactory. Surprisingly enough, no known deterministic logic language can express all deterministic queries computable in polynomial time (e.g., no known deterministic language expresses the *parity* query [7]).

From a pragmatical viewpoint, a clear need for non-determinism is also emerging from applications. The *all-answers* paradigm for query execution exacerbates the need for special constructs to deal with situations where the user is not interested in all the possible answers. This problem is exemplified by

Report CS-R9364
ISSN 0169-118X
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands

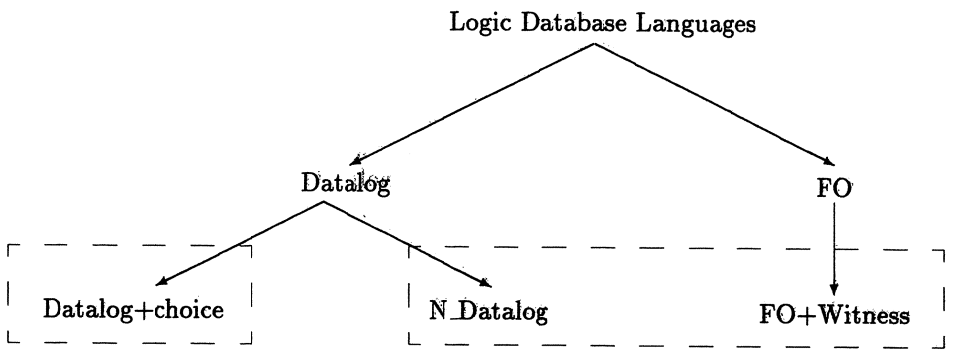
the following situation: a new student must be given one (and only one) advisor. If the application of various qualification criteria fails to narrow the search to a single qualified professor, then an arbitrary choice from the eligible faculty will have to be made and recorded.

Moreover, it has been pointed out in the literature that non deterministic operators provide an explicit means for controlling the computation. Several examples illustrating this point are given in this paper. Explicit control mechanisms are often essential in real applications, in order to achieve efficient implementations—a natural parallel arises here with the technique of meta-interpreters in other programming paradigms. From this perspective, a tight connection exists between non-determinism and ordered databases [16, 6]. It is worth observing that languages over ordered domains are more expressive than those over unordered domains [16].

These are the motivations underlying the introduction of non deterministic mechanisms in logic database languages. A first batch of proposals is due to Abiteboul and Vianu [3, 4, 5, 6], based on a non-deterministic *witness* construct for the fixpoint extensions of *FO*, and a non-deterministic operational semantics for Datalog¬ (*à la production systems*), giving rise to the class of *N_Datalog* languages. The expressive power of these classes of proposals has been thoroughly studied by the same authors, who show how certain non deterministic languages compute exactly the non deterministic time-polynomial queries (*NDB-PTIME*) and the non deterministic space-polynomial queries (*NDB-PSPACE*). On the other hand, these languages are described only in operational terms, without any declarative semantics, thus spoiling the logic nature of the original languages. Moreover, the proposals based on the witness construct are hardly amenable to efficient implementations, and therefore they do not suggest any construct which may be adopted in real database languages.

An alternative stream of proposals was started by Krishnamurthy and Naqvi [18], and later refined by Saccà and Zaniolo [20] and Giannotti, Pedreschi, Saccà and Zaniolo [13]. These proposals are based on a non deterministic *choice* construct for Datalog, which, in all cases, was designed on the basis of a declarative semantics—*choice models* in [18], and *stable models* in [20, 13]. Moreover, the choice construct can be efficiently implemented, and it is actually adopted in the logic database language $\mathcal{LDL}$ [19, 8]. On the other hand, an expressiveness characterization for these proposals is lacking, which allows to compare the choice construct with the other proposals.

The figure highlights the taxonomy of non deterministic logic languages. The dashed boxes indicate the mentioned two classes of proposals.



This work is aimed at bridging the existing gap between the two classes of proposals, by presenting

an expressiveness characterization of Datalog augmented with one of the choice mechanisms, namely the *dynamic choice* construct introduced in [13]. This study is conducted both pragmatically, on the basis of examples, and formally, on the basis of known expressiveness results. In particular, we show how the dynamic choice construct is a powerful means for controlling the fixpoint computation, in order to express relevant problems such as computing the complement of a relation, or computing an arbitrary ordering of a relation.

Finally, in the main result of this paper, we show that Datalog with dynamic choice expresses exactly the non deterministic time-polynomial queries, a complexity class known as *NDB-PTIME*. The result is achieved by showing how the dynamic choice allows us to express the control needed to execute N_Datalog¬ programs over ordered domains—a language which is known to capture *NDB-PTIME*. The relevance of this result is clear: Datalog with the dynamic choice has the same (high) expressiveness of languages which:

- are considerably more complex—Datalog with dynamic choice is negation-less,
- are lacking a declarative semantics—Datalog with dynamic choice is sound (although not complete) w.r.t. stable model semantics,
- are hard to be efficiently implemented—Datalog with dynamic choice is the kernel of $\mathcal{LDL}$.

As a conclusion, a simple, declarative characterization of *NDB-PTIME* is achieved by means of the dynamic choice extension of pure Datalog: such a language, although remarkably simple, is then capable of expressing all non deterministic time-polynomial queries and, therefore, all *deterministic* ones.

The plan of the paper follows. In Section 2 a short survey of the main proposals of non deterministic extensions of logic database languages is provided. Particular emphasis is placed on Datalog extended with the dynamic choice construct. Section 3 and 4 show how to compute negation and ordering using the dynamic choice. Section 5 is devoted to illustrating the emulation of N_Datalog¬ , a language which embodies a form of nondeterminism typical of rule-based systems. Section 6 presents the main result, namely that Datalog with dynamic choice captures the complexity class *NDB-PTIME*; we then draw some conclusions, and briefly illustrate future research directions.

1.1 Preliminaries

We assume that the reader is familiar with the relational data model and associated algebra, the relational calculus (i.e. the *first-order queries*, denoted *FO*), and Datalog [17, 21, 9, 11]. In the extended language Datalog¬ the use of negation in the bodies of clauses (or *rules*) is also allowed; in another extension of Datalog, N_Datalog(¬), we shall also admit the presence of multiple atoms in the heads of clauses. Datalog(¬) (and N_Datalog(¬)) rules obey the *safety* constraint, i.e. each variable occurring in the head of a clause also occurs in a positive literal in the body. The operational semantics of Datalog, in the usual deterministic case, consists of evaluating "in parallel" all applicable instantiations of the rules. This is formalized using the consequences operator T_P associated to a Datalog program P, which is a map over (Herbrand) interpretations defined as follows:

$$T_P(I) = \{\ A \mid A \leftarrow B_1, \ldots, B_n \in ground(P) \text{ and } I \models B_1 \wedge \ldots \wedge B_n\ \}$$

The least model M_P of program P can then be computed as the limit (union) of the finite powers of T_P starting from the empty interpretation, denoted $T_P \uparrow \omega$ [1]:

$$\begin{aligned} T_P \uparrow 0 &= \emptyset \\ T_P \uparrow (i+1) &= T_P(T_P \uparrow i), \quad \text{for } i > 0 \\ T_P \uparrow \omega &= \bigcup_{i \geq 0} T_P \uparrow i. \end{aligned}$$

In the case of Datalog¬, this simple operational semantics can be slightly modified to realize to the so-called *inflationary negation*: the required change is to accumulate the powers of T_P as follows:

$$T_P \uparrow (i+1) \quad = \quad T_P \uparrow i \cup T_P(T_P \uparrow i), \quad \text{for } i > 0.$$

This fixpoint procedure is therefore monotonic only w.r.t. the positive knowledge, and computes, in general, non-minimal models.

The fixpoint (iterative) extensions of FO consist of augmenting the relational calculus with fixpoint operators, which provide recursion. The *inflationary fixpoint* operator IFP is defined as follows. Let Φ be a FO formula where the n-ary relation symbol S occurs. Then $IFP(\Phi, S)$ denotes an n-ary relation, whose extension is the limit of the sequence $J_0, \ldots, J_k, \ldots$, defined as follows (given a database extension, or instance, I):

- $J_0 = I(S)$, where $I(S)$ denotes the extension of S in I, and
- $J_{k+1} = J_k \cup \Phi(I[J_k/S])$, for $k > 0$, where $\Phi(I[J_k/S])$ denotes the evaluation of the query Φ on I where S is assigned to J_k.

Notice that IFP converges in polynomial time on all input databases. A *partial* fixpoint operator PFP can also be defined, which gives raise to possibly infinite computations: PFP is not considered in this paper. The first-order logic augmented with IFP is called *inflationary fixpoint logic* and is denoted by *FO+IFP*. The queries computed by *FO+IFP* are the so-called *fixpoint queries*, for which various equivalent definitions exist in the literature [7, 15].

Close connections exist between the fixpoint FO extensions and the Datalog extensions [6]: Datalog¬ expresses exactly the fixpoint queries, i.e. it is equivalent to *FO+IFP*. This implies that Datalog¬ is strictly more expressive than Datalog with stratified negation, as the latter is known to be strictly included in *FO+IFP*.

Finally, the complexity measures are functions of the size of the input database. For Turing Machine complexity class C there is a corresponding complexity class of (non-deterministic) queries *(N)DB-C*. In particular, the class of (non-deterministic) database queries that can be computed by a (non-deterministic) Turing Machine in polynomial time is denoted by *(N)DB-PTIME*. It is conjectured that no deterministic language exists, capable of expressing all queries in *DB-PTIME*.

2. Non-deterministic extensions of logic database languages

In this section, several mechanisms for dealing with non-determinism in logic database languages are briefly surveyed. In particular, we present a non-deterministic construct for the fixpoint extensions of FO, a non-deterministic operational semantics for Datalog¬ (*à la production systems*), and a non-deterministic mechanism for pure Datalog. The first two classes of proposals are due to Abiteboul and Vianu [3, 5, 6], whereas the third class of proposals is due to Krishnamurthy and Naqvi [18] and Giannotti, Pedreschi, Saccà and Zaniolo [20, 13].

2.1 The witness operator

A non-deterministic extension of FO is achieved by introducing the so-called *witness* operator [3, 5, 6]. Informally, given a formula (query) $\Phi(X)$, the witness operator W_X applied to $\Phi(X)$ chooses an arbitrary X that makes Φ true. The extension of the inflationary fixpoint logic *FO+IFP* with the witness operator is denoted by *FO+IFP+W*.

Let us define more precisely the semantics of W. Notice that, in presence of non-determinism, we have a *set* of possible interpretations for a given formula in *FO+IFP+W*, or equivalently, a set of possible sets of answers to a given query. Consider a formula $W_X(\Phi(X, Y))$, where Y is the vector of

variables other than X that occur free in Φ. Then I is an interpretation of $W_X(\Phi(X,Y))$ iff, for some interpretation J of $\Phi(X,Y)$ such that $I \subseteq J$:

- for each Y such that $\langle X,Y\rangle \in J$ for some X, there is a *unique* X_Y such that $\langle X_Y,Y\rangle \in I$.

Intuitively, one "witness" X_Y is arbitrarily chosen for each Y satisfying $\exists X.\Phi(X,Y)$. Alternatively, the meaning of W can be also described in terms of functional dependencies: the interpretation I is a maximal subset of J satisfying the functional dependency $Y \rightarrow X$.

Example 1 Consider a binary relation E such that $E(P,S)$ represents the fact that professor P is an eligible advisor of student S. Then the formula $W_P(E(P,S))$ realizes the non-deterministic query of assigning exactly one advisor to each student.

It should be noted that the witness operator is added to FO independently from the fixpoint operator. Accordingly, the fixpoint computation and the non-deterministic choices do not interfere, in the sense the non-deterministic choices of the witnesses are performed w.r.t. the current fixpoint approximation, without memory of the choices that were previously operated. In other words, the witness operator performs choices *locally* to a given step of the fixpoint computation.

From the viewpoint of the expressive power, the relevance of $FO{+}IFP{+}W$ is due to the following result of Abiteboul and Vianu [5]:

Theorem 2 *A query is in NDB-PTIME iff it is expressed in* $FO{+}IFP{+}W$. □

An analogous result of the same authors shows that $FO{+}PFP{+}W$, i.e. FO augmented with the partial fixpoint and the witness operators, expresses exactly the queries in *NDB-PSPACE*.

2.2 N_Datalog

A natural form of non-determinism for Datalog programs is obtained by relaxing the constraint that, at each step of the fixpoint computation, all applicable rules are executed. Thus, a non-deterministic operational semantics is obtained by firing, at each step, one (instance of an) applicable rule, based on a non-deterministic choice. This policy directly mirrors the behavior of rule-based (or production) systems, such as OPS5 or KEE. Notice that such an execution policy yields the same results as the usual Datalog fixpoint computation in absence of negation, as, in pure Datalog, an applicable rule remains applicable as new facts are inferred.

Abiteboul and Vianu [5] proposed to adopt the mentioned non-deterministic operational semantics for *N_Datalog*$\neg$, an extension of pure Datalog which allows the use of negation in clause bodies, and multiple atoms in clause heads. Thus, an N_Datalog program is a finite set of rules of the form

$$A_1, \ldots, A_k \leftarrow L_1, \ldots, L_m$$

($k \geq 1, m \geq 0$), where each A_j is an atom and each L_i is a literal, i.e. an atom or its negation.

To define the non-deterministic operational semantics, the notion of *immediate successor* of an interpretation (i.e. a set of facts) I w.r.t. a rule r is introduced. Let $r' = A_1, \ldots, A_k \leftarrow L_1, \ldots, L_m$ be a ground instance of an N_Datalog$\neg$ rule r such that all literals $L_1, \ldots, L_m$ in the body of r' are true in I. Then the interpretation $J = I \cup \{A_1, \ldots, A_k\}$ is called an *immediate successor of* I *using* r. We then define a computation of an N_Datalog$\neg$ program P starting from an initial interpretation I_0 as a maximal sequence $I_0, \ldots, I_n, \ldots$ of interpretations such that, for $k \geq 0$, I_{k+1} is an immediate successor of I_k using some rule from P.

It is worth observing that such an operational semantics is inflationary, and thus computations are always finite (and, again, convergent in polynomial time).

Example 3 The following Datalog¬ program takes as input a binary relation G representing an undirected graph g, and computes (into the relation DG) an arbitrary orientation of g:

$$DG(X,Y) \leftarrow G(X,Y), G(Y,X), \neg DG(Y,X).$$

From the viewpoint of the expressive power, N_Datalog¬ is strictly included in *NDB-PTIME*. In fact, it is possible to show that such a language cannot express the query $P - \pi_1(Q)$, where P is a unary relation and Q a binary one. Thus, it is needed to extend N_Datalog¬ in order to capture all the queries in *NDB-PTIME*. Two possible approaches of remedying this problem are the following. One is allowing universal quantification in clause bodies: the resulting language is denoted N_Datalog¬∀. The second is violating the *data independence* principle, and allowing the use of *ordered* databases. In both cases we obtain languages that capture *NDB-PTIME*, and that are therefore equivalent to *FO+IFP+W*. This result is due to Abiteboul and Vianu [6].

Theorem 4 *A query is in NDB-PTIME iff it is expressed in N_Datalog¬∀ or, equivalently, in N_Datalog¬ over ordered databases.* □

An analogous result of the same authors shows that N_Datalog¬*, i.e. N_Datalog¬ augmented with negation in rule heads (interpreted as deletion of facts), expresses exactly the queries in *NDB-PSPACE*.

2.3 The family of choice operators

The proposals discussed in the previous sections 2.1 and 2.2 suffer from the lack of a declarative, model-theoretic semantics, which seriously compromises their logic connotation. Another approach was started by Krishnamurthy and Naqvi [18], and later refined by Saccà and Zaniolo [20] and Giannotti, Pedreschi, Saccà and Zaniolo [13]. The proposals described in this section are based on a non deterministic *choice* construct for Datalog, which, in all cases, was designed on the basis of a declarative semantics—*choice models* in [18], and *stable models* in [20, 13]. Moreover, the choice construct can be efficiently implemented, and it is actually adopted in the logic database language $\mathcal{LDL}$[19, 8]. On the other hand, an expressiveness characterization for these proposals is lacking, which allows to compare the choice construct with the previously discussed proposals. The rest of this section surveys the original proposal and two refinements, which improve from several viewpoints.

Static choice The choice construct was first proposed by Krishnamurthy and Naqvi in [18]. According to their proposal, special goals, of the form $choice((X),(Y))$, are allowed in Datalog rules to denote the functional dependency (FD) $X \rightarrow Y$. The meaning of such programs is defined by its *choice models*, as discussed next.

Example 5 Consider the following Datalog program with choice.

$a\_st(St, Crs) \leftarrow takes(St, Crs), choice((Crs),(St)).$
$takes(andy, engl).$
$takes(ann, math).$
$takes(mark, engl).$
$takes(mark, math).$

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.

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 is a *choice rule*, while a rule without choice predicates is 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 choice program 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).

For the sake of simplicity, assume that P contain only one choice rule r, as follows:

$$r : A \leftarrow B(Z), choice((X),(Y)).$$

where $B(Z)$ denotes the conjunction of all the non-choice goals of r, and Z is the vector of variables occurring in the body of r (hence $Z \supseteq X \cup Y$.) The positive version of P, denoted by $PV(P)$, is the positive program obtained from P by eliminating all *choice* goals. Let M_P be the least model of the positive program $PV(P)$, and consider the set C_P defined as follows:

$$C_P = \{\ choice((x),(y)) \mid M_P \models B(z)\}$$

Consider next a maximal subset C'_P of C_P satisfying the FD $X \rightarrow Y$. With this preparation, a choice model of P is defined as the least model of the program $P \cup C'_P$.

Thus, computing with the static choice entails three stages of a bottom-up procedure. In the first stage, the saturation of $PV(P)$ is computed, ignoring choice goals. In the second stage, an extension of the choice predicates is computed by non-deterministically selecting a maximal subset of the corresponding query which satisfies the given FD. Finally, a new saturation is performed using the original program P together with the selected choice atoms, in order to propagate the effects of the operated choice.

The qualification *static* for this choice operator stems from the observation that the choice is operated once and forall, after a preliminary fixpoint computation. Because of its static nature, this form of choice cannot be safely used within recursive rules. As observed in [13], the choice models semantics fails when mixed with recursion, in the sense that the delivered results do not comply with any declarative reading. Moreover, the procedure for computing choice models is extremely inefficient, as operating the choices only after a general saturation phase is wasteful—a more efficient procedure should instead operate choices as soon as possible, in order to reduce the amount of work for future saturations. Finally, due to the impossibility of being adopted within recursion, the static choice has a limited expressive power. To remedy these drawbacks, some refinements of the static choice have been proposed, which are discussed next.

Model-theoretical choice An alternative approach to define a declarative semantics for the choice construct was proposed by Saccà and Zaniolo [20]. According to this proposal, programs with choice

are transformed into programs with negation which exhibit a multiplicity of stable models. [1] Each stable model corresponds to an alternative set of answers for the original program. Following [20], therefore, given a choice program P, we introduce the *stable version* of P, denoted by $SV(P)$, defined as the program with negation obtained from P by the following two transformation steps:

1. Consider a choice rule of P, say

   $$r : A \leftarrow B(Z), choice((X),(Y)).$$

   where $B(Z)$ denotes the conjunction of all the non-choice goals of r, and Z is the vector of variables occurring in the body of r, and replace the body of r with the atom $chosen(Z)$:

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

2. add the new rule:

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

3. add the new rule:

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

   where Z' is a list of variables obtained from Z by replacing variable Y by the fresh variable Y'.

The transformation directly generalizes to FD involving vectors of variables, and to multiple choice goals. 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 coincides with the set of stable models of $SV(P)$ on common predicate symbols [20].

Example 6 The following is the stable version of Example 3.

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

This programs admits four distinct stable models, corresponding to the four choice models of Example 3.

It should be remarked that, in choice programs, negation is only used to assign a declarative semantics to the choice construct. In other words, choice programs are *positive* Datalog programs augmented with choice goals.

This new characterization of choice overcomes the cited deficiencies of static choice of Krishnamurthy and Naqvi [18]. Indeed, the new formulation correctly supports the use of choice within recursive rules, avoiding the semantical anomalies of the static choice [13]. Moreover, it can be efficiently implemented by a straightforward fixpoint procedure which allows to interleave non-deterministic choices and ordinary rule applications in the bottom-up computation (the so-called *stable backtracking fixpoint* [20]). Nevertheless, the expressiveness of this form of choice can be considerably enhanced by adopting a particular instance of the cited fixpoint procedure.

[1] Stable models semantics is a concept originating from autoepistemic logic, which was applied to the study of negation in Horn clause languages by Gelfond and Lifschitz [12].

Dynamic choice We now introduce a particular operational semantics for the choice construct, following the presentation of Giannotti, Pedreschi, Saccà and Zaniolo [13]. This operational semantics is an instance of the general bottom-up procedure of Saccà and Zaniolo [20] for computing stable models, and is obtained by adopting a particular policy of interleaving non-deterministic choices and the ordinary fixpoint computation. The resulting procedure is referred to as DCF for *dynamic choice fixpoint*, and the associated form of choice construct is referred to as *dynamic choice*.

The DCF procedure, and thus the dynamic choice construct, reflects the intuition that choices should be operated as soon as possible during the fixpoint computation. This design principle has two relevant consequences. First, a higher degree of efficiency is achieved, as early choices have the effect of reducing the number of inferred facts at the intermediate stages of the fixpoint computation, and possibly of anticipating its termination. Second, a higher degree of expressiveness is achieved: the next sections of this paper are devoted to this point. For instance, we will show how the dynamic choice construct is expressive enough to capture various forms of negation for Datalog.

Informally, the DCF procedure 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 (original) 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 ground instance r of a *chosen* rule in **C**, repeat:
   (a) execute r;
   (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. Notice that the execution the $diffchoice$ rules shrinks the set of enabled *choice* rules.

In other words, when DCF is in the choice mode, all the choices that are compatible with the functional dependency are operated, before DCF switches to the saturation mode again.

The following code formalizes the 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.

The DCF procedure is correct with respect to the stable choice model semantics of the program, in the sense that the result of DCF is a stable choice model of the 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 any stable choice model of a program, but only some *preferred* ones. Therefore, dynamic choice is sound, although not complete, w.r.t. stable model semantics. The main interest for the dynamic choice construct lies in the fact that it is highly expressive—it allows to compute efficiently some relevant *deterministic* problems which cannot be expresses by deterministic, such as negation and ordering. These and other issues are addressed in the rest of this paper.

3. Computing negation with the choice operator

A remarkable example taken from [13] is the realization of a form of negation, which can be used to model stratified and inflationary negation for Datalog. The following choice program defines relation NOT_P as the complement of a relation P with respect to a universal relation U. We assume here that both P and U are extensional relations, although this constraint will be soon relaxed.

Definition 7 The choice program $NOT[P, U]$ consists of the following rules:

$$NOT\_P(X) \leftarrow COMP\_P(X, 1).$$

$$COMP\_P(X, I) \leftarrow TAG\_P(X, I), choice((X), (I)).$$

$$TAG\_P(nil, 0).$$
$$TAG\_P(X, 0) \leftarrow P(X).$$
$$TAG\_P(X, 1) \leftarrow U(X), COMP\_P(\_, 0).$$

where *nil* is a new constant, which does not occur in the EDB. □

According to the specified operational semantics of the dynamic choice, we obtain a set of answers where $COMP\_P(x, 1)$ holds if and only if x is not in the extension of P. This behavior is due to the fact that the extension of $COMP\_P$ is taken as a subset of the relation TAG_P which obeys the FD $(X \rightarrow I)$, and that the dynamic choice operates early choices which binds to 0 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 1. The fact rule $TAG\_P(nil, 0)$ is needed to cope with the case that relation P is empty.

More precisely, in the first saturation phase the facts and $TAG\_P(x, 0)$ are inferred, for x in the extension of relation P. In the following choice phase the facts $chosen(x, 0)$ are chosen, again for x in the extension of P, as all possible choices are operated. In the second saturation phase the facts $COMP\_P(x, 0)$ are inferred for x in the extension of P, and the facts $TAG\_P(x, 1)$ for all x in U. In the following choice phase the facts $chosen(x, 1)$ are chosen in a maximal way to satisfy the FD, i.e. for x *not* in the extension of P, as all x in P have been chosen with tag 0 already. In the third saturation step the extension of NOT_P becomes the complement of P with respect to U.

The above argument is actually a sketch of the proof of the following result, which states the correctness of the program of Def. 7.

Proposition 8 *Let P and U be n-ary EDB relations. Then program $NOT[P,U]$ has a unique stable choice model M_{NOT}, and $M_{NOT} \models NOT\_P(x)$ iff $x \in U \setminus P$.* □

Essentially, this example shows how the dynamic choice offers a flexible mechanism for handling the control needed to emulate the difference between two relations. It is shown in [10] that the above program can be refined in order to realize more powerful forms of negation, such as stratified and inflationary negation. This goal is achieved by suitably emulating the extra control needed to handle program strata and fixpoint approximations, respectively.

4. Ordering with the choice operator

It has been pointed out in the literature that a tight connection exists between non-determinism and ordered databases [16, 6]. On one hand, consider the case that a query Q relies on the ordering in which elements are stored in the database: when abstracted at the conceptual level, where physical details are unrelevant, Q exhibits a non-deterministic behavior. On the other hand, it is often possible to emulate ordering using non-deterministic mechanisms.

The following choice program $ORD[U]$ exploits the dynamic choice to compute an arbitrary ordering of the elements of an EDB relation U.

Definition 9 The choice program $ORD[U]$ consists of the following rules:

$$SUCC(min, Y) \leftarrow \quad U(Y), choice((), (Y)).$$

$$SUCC(X, Y) \leftarrow \quad SUCC(\_, X), U(Y), SUCC(min, Z), \\ X \neq Y,\ Y \neq Z,\ choice((X), (Y)), choice((Y), (X)).$$

where *min* is a new constant, which does not occur in the EDB. □

According to the specified operational semantics of the dynamic choice, we obtain a set of answers where the extension of relation $SUCC$ is a total, strict ordering over the input relation U. The first clause of program $ORD[U]$ starts the computation, by selecting an arbitrary element from U as the successor of *min*, i.e., as the actual minimum element of U. The second clause selects from U the successor y of an element x which has been already placed in order. The constraints in the body of the second clause enforce irreflexivity. In particular:

- $x \neq y$ prevents immediate cycles (e.g., $SUCC(a, a)$),
- $y \neq z$ prevents cycles with the minimum element z,
- the choice goals establish the bijection $x \leftrightarrow y$ which prevents the other possible cycles; also, y is uniquely determined by x.

The above argument is actually a sketch of the proof of the following result, which states the correctness of the program of Def. 9.

Proposition 10 *Let U be an EDB relation. Then, in any stable choice model M_{ORD} of program $ORD[U]$, the (transitive closure of the) relation SUCC is an irreflexive total ordering over U.* □

This application brings further evidence to the effectiveness of the dynamic choice as a control mechanism. It also suggests that the dynamic choice is highly expressive, as languages over ordered domains are known to be strictly more expressive than languages over unordered domains [16]. Indeed, the fact that dynamic choice can express ordering is essential in the proof of the main result of this paper.

5. Emulating N_Datalog with the choice operator

The aim of this section is to present a general transformation algorithm which allows to emulate the control needed to handle the non-deterministic semantics of N_Datalog¬. Ordering over a relation of a suitable cardinality is exploited to emulate the level of the fixpoint iteration of the N_Datalog¬ computation.

Definition 11 (*Transformation*) Let *Prog* be a N_Datalog¬ program. Let δ be the finite set of distinct constants occurring in *Prog*, and L the cardinality of δ. Let l be the number of distinct relations of *Prog*, and l_1 be the maximal arity of the relations in *Prog*. As a consequence, L^{l*l_1} is an upper bound for number of instances which are derivable from *Prog*. Given a set of variables $\{V_1, \ldots, V_{l*l_1}\}$, all variables occurring in the heads of a rule in *Prog* can be renamed using variables from this set.

Prog' is a choice program obtained from *Prog* according to the following steps:

1. Add the following facts:

   $$LEVEL(min).$$
   $$UNIV(a_1, \ldots, a_{l*l_1}).$$

   for $a_j \in \delta, j = 1, \ldots, l * l_1$, together with the rules of the program $ORD[UNIV]$ as in Def. 9 as in Here, *min* is an array (of proper arity) of new constants.

2. Add the following rules defining the complement of a relation P respect another relation U. Such rules extend the program of Def. 7 to deal with the level of the fixpoint iteration. The notation $NOT[P, U](x, n)$ is used in the following to refer to the following program.

   $$NOT\_P(X, N) \leftarrow COMP\_P(X, 1, N).$$

   $$COMP\_P(X, I, N) \leftarrow TAG\_P(X, I, N), choice((X), (I)).$$

   $$TAG\_P(nil, 0, N) \leftarrow LEVEL(N).$$
   $$TAG\_P(X, 0, N) \leftarrow P(X), LEVEL(N).$$
   $$TAG\_P(X, 1, N) \leftarrow U(X), COMP\_P(\_, 0, N).$$

   where *nil* is a new constant.

3. For each rule R_i of *Prog*:

   $$A_0(X_0), \ldots, A_m(X_m) \leftarrow P_1(Y_1), \ldots, P_k(Y_k), \neg Q_1(Z_1), \ldots, \neg Q_h(Z_h).$$

   with $h, k, m \geq 0$, add the following rules:

$$NEW(U, N, i) \leftarrow P_1(Y_1), \ldots, P_k(Y_k), NOT[Q_1, UNIV](Z_1, N), \ldots, NOT[Q_h, UNIV](Z_h, N), NOT[A_0, UNIV](X_0, N).$$

$$\vdots$$

$$NEW(U, N, i) \leftarrow P_1(Y_1), \ldots, P_k(Y_k), NOT[Q_1, UNIV](Z_1, N), \ldots, NOT[Q_h, UNIV](Z_h, N), NOT[A_m, UNIV](X_m, N).$$

Here, i is a constant identifying the rule R_i, and U is an array of terms of arity $l * l_1$, which contains all variables in the head of the original rule; all the extra arguments of U are filled in with a new constant ∂.

4. Add the rule:

$$INSTANCE(Z, N, I) \leftarrow NEW(U, N, I), choice((N), (U, I)).$$
$$LEVEL(N_1) \leftarrow INSTANCE(\_, N, \_), SUCC(N, N_1).$$

where I is a variable denoting a generic rule from $Prog$, and Z is the set of variables $\{V_1, \ldots, V_{l*l_1}\}$.

5. Replace rule R_i with the following rules:

$$A_0(X_0) \leftarrow INSTANCE(Z, N, i).$$
$$\vdots$$
$$A_m(X_m) \leftarrow INSTANCE(Z, N, i).$$

□

Before analyzing the transformation, let us recall the behavior of the non-deterministic semantics: at each fixpoint iteration a new instance is computed by choosing only one instantiation among the possible ones of the single rule chosen among the firable ones. At step 3 of the transformation the predicate $NEW(\_, i)$ collects *only* all the new instances derived using the rule R_i. In fact the meta_predicate $NOT[A_j, UNIV]$) ensures that instances for the predicate A_i occurring in the head of the rule have not been computed yet. At step 4 only one rule and only one instance of the selected rule are selected. At step 5 the predicates of the head of the selected rule are inferred. At this stage, a new value for $LEVEL$ can be inferred which will possibly fire rules of the meta-predicate $NOT[\_, UNIV]$.

It is worth remarking that in the non-deterministic semantics also the EDB facts are derived one at a time, so the transformation considers them as IDB rules with an equality between variables and constants the body.

The above argument is a rough sketch of the proof of the correctness of the transformation. To formalize this statement we need the following definition.

Definition 12 Let P be a N_Datalog$\neg$ and P' a choice program. P and P' are *semantically equivalent* with respect to common predicate symbols if

- for each model M of P there exists a stable choice model M' of P' which coincides with M over common predicates, and $M \models \neg R(x)$ iff $M' \models NOT\_R(x)$ for each relation R occurring in P.
- for each stable choice model M' of P' there exists a model M of P which coincides with M' over common predicates, and $M \models \neg R(x)$ iff $M' \models NOT\_R(x)$ for each relation R occurring in P.

□

The above definition takes into account the fact that negative information is represented in choice programs with the NOT_R predicates.

Theorem 13 *Let P be a N_Datalog¬ and let P' a choice program obtained from P applying the transformation of Def. 11. Then P and P' are semantically equivalent.* □

Example 14 We show the transformation on a simple N_Datalog¬ program:

$$
\begin{array}{ll}
R_1: & P(x), Q(y) \leftarrow \neg R(a), S(x), T(y). \\
R_2: & S(x) \leftarrow T(x). \\
R_3: & T(a).
\end{array}
$$

The following are the relevant rules of the corresponding choice program:

$$
\begin{array}{lll}
LEVEL(min). & & \\
UNIV(a). & & \\
NEW(X,Y,N,1) & \leftarrow & NOT[R,UNIV](a,N), S(X), T(Y), \\
 & & NOT[P,UNIV](X,N) \\
NEW(X,Y,N,1) & \leftarrow & NOT[R,UNIV](a,N), S(X), T(Y), \\
 & & NOT[Q,UNIV](Y,N) \\
NEW(X,\partial,N,2) & \leftarrow & T(X), NOT[S,UNIV](X,N) \\
NEW(a,\partial,N,3) & \leftarrow & NOT[T,UNIV](a,N) \\
INSTANCE(X,N,I) & \leftarrow & NEW(X,Y,N,I), choice((N),(X,Y,I)). \\
P(X) & \leftarrow & INSTANCE(X,Y,N,1). \\
Q(Y) & \leftarrow & INSTANCE(X,Y,N,1). \\
S(X) & \leftarrow & INSTANCE(X,Y,N,2). \\
T(X) & \leftarrow & INSTANCE(X,Y,N,3). \\
LEVEL(N_1) & \leftarrow & INSTANCE(\_,\_,N,\_), SUCC(N,N_1).
\end{array}
$$

6. Datalog with dynamic choice computes *NDB_PTIME*

We are now in the position of summing up the results of the previous sections in the main result of this paper. It is stated by the following

Theorem 15 *A query is in NDB-PTIME iff it is expressed in Datalog with dynamic choice.*

Proof. The *only if* part follows from the following facts:

- Datalog with dynamic choice emulates N_Datalog¬ (Theorem 13),
- Datalog with dynamic choice expresses ordering (Proposition 10), and
- N_Datalog¬ over ordered domains expresses *NDB-PTIME* (Theorem 4).

The *if* part follows from the observation that Datalog with dynamic choice is an inflationary language, as operated choices are never retracted. □

Theorem 15 defines precisely the expressive power of Datalog augmented with the dynamic choice construct. As a consequence, we obtain that such a language embodies a simple, declarative and efficiently implementable characterization of *NDB-PTIME*, thus improving over previous results.

From a more pragmatical viewpoint, these results indicate that dynamic choice is a flexible mechanism for explicitly handling the control in the fixpoint computation. A natural parallel here is with the *cut* control mechanism of Prolog, which is however much more difficult to be explained in declarative terms [14]. Also, it is natural to ask ourselves whether the dynamic choice provides us with the basis for constructing *bottom-up meta-interpreters*, capable of turning logic database programs into efficient systems by exploiting a customized computation strategy. Another open problem is whether it is realistic to implement negation and ordering by choice in a real language.

Finally, we mention another direction for future work. Abiteboul and Vianu showed that certain non deterministic languages augmented with the extra possibility of performing *updates* are capable of expressing *NDB-PSPACE*, i.e., the non deterministic space-polynomial queries [6]. We conjecture that a similar result holds when augmenting Datalog with dynamic choice and an update construct, such as that of $\mathcal{LDL}$.

ACKNOWLEDGMENTS

Thanks are owing to Victor Vianu, Luigi Palopoli, Carlo Zaniolo and Mimmo Saccà for their useful suggestions on the subject of this paper. In particular, we owe the ordering example to L. Palopoli.

REFERENCES

1. K. R. Apt. *Introduction to Logic Programming.* In: Handbook of Theoretical Computer Science, vol B. (Ed. J. van Leeuwen) (1990). pp. 493-574.
2. S. Abiteboul, E. Simon, V. Vianu. *Non-Deterministic Language to Express Deterministic Transformation.* Proceedings of ACM Symposium on Principles of Database Systems, 1990. pp. 218-229.
3. S. Abiteboul, V. Vianu. *Transaction Languages for Databases Update and Specification.* INRIA Technical Report n. 715 (1987).
4. S. Abiteboul, V. Vianu. *Procedural Languages for Database Queries and Updates.* Journal of Computer and System Science 41 (2) (1990).
5. S. Abiteboul, V. Vianu. *Fixpoint Extension of First Order Logic and Datalog-Like Languages.* Proc. 4th Symp on Logic in Computer Science (LICS). IEEE Computer Press (1989). pp. 71-89.
6. S. Abiteboul, V. Vianu. Non-Determinism in Logic Based Languages. Annals of Mathematics and Artificial Intelligence 3 (1991). pp. 151-186.
7. A. Chandra, D. Harel. *Structures and Complexity of Relational Queries.* Journal of Computer and System Science 25 (1982). pp. 99-128.
8. D. Chimenti, et al., *The $\mathcal{LDL}$ System Prototype.* IEEE Journal on Data and Knowledge Engineering, Vol. 2, No. 1, (1990). pp. 76-90.
9. E.F. Codd. *Relational Completeness of Database Sublanguages.* Data Base Systems, (Ed. R. Rustin), Prentice-Hall, Englewood Cliffs, NJ (1972) pp. 33-64.
10. L. Corciulo. *Non determinism in deductive databases.* Laurea Thesis. Dipartimento di Informatica, Università di Pisa. 1993 (in Italian)
11. H. Gallaire, J. Minker, J.M. Nicolas. *Logic and Databases, a Deductive Approach.* ACM Computing Surveys 16(2) (1984). pp. 153-185.

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

13. F. Giannotti, D. Pedreschi, D. Saccà, C. Zaniolo. *Non-Determinism in Deductive Databases.* Proc. Deductive and Object-oriented Databases, Second International Conference, DOOD'93, (Eds. C. Delobel, M. Kifer, Y. Masunaga), Springer-Verlag, LNCS 566, pp. 129-146, 1991.

14. F. Giannotti, D. Pedreschi, C. Zaniolo. *Declarative Semantics for Pruning Operators in Logic Programming.* Methods of Logic in Computer Science (1993) To appear.

15. Y. Gurevich, S. Shelah. *Fixed-Point Extensions of First-Order Logic.* Annals of Pure and Applicate Logic 32 (1986). pp. 265-280.

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

17. P.C. Kanellakis. *Elements of Relational Databases Theory.* In: Handbook of Theoretical Computer Science, (Ed. J. van Leeuwen) (1990). pp. 1075-1155.

18. R. Krishnamurthy, S. Naqvi. *Non-Deterministic Choice in Datalog.* Proc. 3nd Int. Conf. on Data and Knowledge Bases, Morgan Kaufmann Pub., Los Altos (1988). pp. 416-424.

19. S. Naqvi, S. Tsur. *A Logical Language for Data and Knowledge Bases.* Computer Science Press, New York (1989).

20. D. Saccà, C. Zaniolo. *Stable Models and Non-Determinism in Logic Programs with Negation.* Proc. Symp. on Principles of Database System PODS'89 (1989).

21. J.D. Ullman. *Principles of Databases and Knowledge Base System.* Volume I and II. Computer Science Press, Rockville, Md (1988).