Semantics and Expressive Power of Non-Deterministic Constructs in Deductive Databases

Fosca Giannotti[1], **Dino Pedreschi**[2] and **Carlo Zaniolo**[3]

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

[2] Dipartimento di Informatica, Univ. Pisa, Corso Italia 40, 56125 Pisa, Italy
pedre@di.unipi.it

[3] Computer Science Dept., UCLA, USA
zaniolo@cs.ucla.edu

Abstract

Non-deterministic extensions are needed in logic-based languages, such as First-Order relational languages and Datalog, to enhance their expressive power and support the efficient formulation of low-complexity problems and database queries. In this paper, we study the semantics and expressive power of the various non-deterministic constructs proposed in the past, including various versions of the *choice* operator and the *witness* operator. The paper develops a model-theoretic semantics, a fixpoint semantics, and an operational semantics for these constructs, and characterizes their power of expressing deterministic and non-deterministic queries. The paper presents various soundness and completeness results and establishes an expressiveness hierarchy that correlates the various operators with each other, and with other constructs, such as negation and fixpoint.

Keywords and phrases. Deductive databases, logic–based languages, non-determinism, expressive power, operational semantics, declarative semantics, fixpoint.

ACM-CR Subject Classification. D.1.6, I.2.3, H.2.3, F.1.3, F.3.2, F.3.3

1 Introduction

Two main classes of logic-based languages have been extensively investigated as the theoretical basis for relational database languages and their generalizations. One is the class of first-order FO languages, which are based on the relational algebra. The other is the class of Datalog languages, which are endowed with an elegant semantics based on notions such as minimal model and least fixpoint. In many respects, these two lines of research on database languages have faced similar problems and evolved along parallel and closely related directions.

A common focus of research has been on constructs that enable the expression of $PTIME$ algorithms. Many $PTIME$ problems that could not be expressed in relational algebra (e.g., transitive closures) can be expressed using recursive Datalog rules, or through the addition of a fixpoint operator in FO. But even after the introduction of a fixpoint operator, many $PTIME$ queries, such as the parity query, cannot be expressed unless a total order on the database is assumed, thus compromising the data-independence principle of *genericity* [7, 5]. Symmetrically, the parity query, and many others low-complexity queries, cannot be expressed in Datalog with stratified negation. However, non-deterministic constructs provide a very effective vehicle for achieving $PTIME$; this observation led to the parallel but independent introduction of a *witness* operator for FO languages and a *choice* operator for Datalog.

The work of Abiteboul and Vianu [1, 2, 3, 4] brought into focus the need for having non-deterministic operators in such languages in addition to recursion or fixpoint. Therefore, they proposed the non-deterministic construct called the *witness* [2, 3, 4] for the fixpoint extensions of FO. They also proposed a non-deterministic procedural semantics for Datalog$\neg$ (*à la production systems*), giving rise to the class of *N_Datalog* languages. The referenced work also characterized the expressive power of languages with these constructs for both deterministic and non-deterministic queries.

Concurrently, the quest for enhancing the expressiveness of Datalog led to the introduction of the *choice* construct in the logic database language $\mathcal{LDL}$ [27]. This proposal extends the procedural (bottom-up) semantics of deductive databases in such a way that a subset of query answer is chosen, on the basis of a functional dependency constraint. Successive studies showed that several improvements were needed. Therefore, the original proposal by Krishnamurthy and Naqvi [26] was later revised by Saccà and Zaniolo [29], and refined in Giannotti, Pedreschi, Saccà and Zaniolo [16]. These studies also exposed the close relationship connecting nonmonotonic reasoning with non-deterministic constructs, leading to the definition of a stable-model semantics for choice. While the declarative semantics of *choice* is based on *stable model* semantics, choice is amenable to efficient implementations, and it is actually supported in the logic database languages $\mathcal{LDL}$ and $\mathcal{LDL}$++ [27, 9].

So far, no comprehensive study had been produced, which analyzes and compares non-deterministic constructs such as choice and witness operators. In this paper, therefore, these operators are systematically studied, with the aim of:

- clarifying the relationships among their procedural, declarative and fixpoint semantics, and
- comparing their power of expressing deterministic and non-deterministic queries.

More precisely we will compare:

- Datalog with *static choice*, i.e., the choice construct in [26], both without negation, and with stratified negation.
- Datalog with *lazy dynamic choice*, i.e., the choice construct in [29], both without negation, and with stratified negation.

- Datalog with *eager dynamic choice*, i.e., the choice construct in [16].
- FO + W(itness) and inflationary fixpoint IFP.
- The positive existential calculus FO^+ + W(itness) and inflationary fixpoint IFP.

This analysis will be performed for both *deterministic* and *non-deterministic queries.*

A *non-deterministic* query is one which has more than one acceptable answer. A relevant example of such a query is the construction of an arbitrary total ordering relation for the constants in the universe of interest. For a universe with n constants, there are $n!$ acceptable answers that can be returned for such a query.

A *deterministic* query is one which admits only one correct answer. Without non-deterministic constructs, Datalog can only compute deterministic queries. On the other hand, a Datalog program with non-deterministic constructs might produce a deterministic query. In fact, non-deterministic operators are also essential for expressing deterministic low complexity queries, such as determining the parity or the cardinality (or other set-aggregation functions) of a given relation.

We establish a hierarchy of increasing expressiveness. First, we prove that Datalog with static choice has the same expressiveness as the non-deterministic fixpoint extension of the positive fragment of FO. Second, we recall a result from [12, 17] and observe that Datalog with eager dynamic choice has the same expressiveness as the non-deterministic fixpoint extension of FO—a language which is known to express exactly the non-deterministic polynomial-time queries *NDB-PTIME.* Third, we show that Datalog with lazy dynamic choice exhibits an intermediate expressiveness.

In addition to the systematic analysis of the expressive power of non-deterministic constructs, this paper presents a novel characterization of non-deterministic bottom-up evaluation of dynamic choice programs, by introducing an immediate consequence operator which unifies the procedural, declarative and fixpoint-based semantics for choice. In particular, we prove soundness and completeness of the procedural semantics with respect to the intended declarative semantics, based on stable models. This allows us to prove the monotonic behavior of programs with lazy dynamic choice.

The outline of the paper is as follows. In Section 2 we provide some background on Datalog semantics, relational calculus and query complexity. In Section 3 we briefly review the non-deterministic *witness* operator for relational calculus, while in Section 4 we review the *static choice* construct for Datalog. Section 5 presents a first batch of expressiveness characterizations, pointing out the similarity between witness and static choice. The *dynamic* choice construct and its declarative semantics based on stable models is reviewed in Section 6, while its expressiveness is discussed in section 7. In Section 8, the fixpoint-based semantics of dynamic choice is formalized and shown equivalent to the declarative semantics, and the two different versions of the dynamic choice, *lazy* and *eager*, are characterized. Finally, Section 9 presents the expressiveness characterization of lazy and eager dynamic choice, together with the overall expressive power hierarchy of the non-deterministic extensions of Datalog.

2 Background

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 [5, 23, 30, 10, 13]. The basic semantics of Datalog consists in evaluating "in parallel" all applicable instantiations of the rules. This is formalized using the *immediate consequence operator* T_P associated to a Datalog program P, which is a map over (Herbrand) interpretations defined as follows:

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

The upward powers of T_P starting from an interpretation I are defined as follows:

$$\begin{array}{rcll} T_P \uparrow 0(I) & = & I & \\ T_P \uparrow (i+1)(I) & = & T_P(T_P \uparrow i(I)), & \text{for } i \geq 0 \\ T_P \uparrow \omega(I) & = & \bigsqcup_{i \geq 0} T_P \uparrow i(I). & \end{array}$$

The least model M_P of a positive program P is equal to $T_P \uparrow \omega(\emptyset)$ which we will also abbreviate to $T_P \uparrow \omega$.

The *inflationary* version of the T_P operator is defined as follows:

$$T'_P(I) \ = \ T_P(I)\ \cup I.$$

For programs without negation, $T_P \uparrow \omega = T'_P \uparrow \omega = M_P$. This equality no longer holds for the language Datalog¬ which allows the use of negation in the bodies of rules. In this case, many alternative semantics can be adopted. One possibility is the so called *inflationary semantics* for Datalog¬ which adopts $T'_P \uparrow \omega$ as the meaning of a program P. Other alternatives are well-founded semantics, and stable model semantics. In this paper, we use the *stable model* semantics of Datalog¬ programs, a concept originating from autoepistemic logic, which was applied to the study of negation in Horn clause languages by Gelfond and Lifschitz [14]. To define the notion of a stable model we need to introduce a transformation which, given an interpretation I, maps a Datalog¬ program P into a positive Datalog program $\mathit{ground}_I(P)$ obtained from $\mathit{ground}(P)$ by

- dropping all clauses with a negative literal $\neg A$ in the body with $A \in I$, and
- dropping all negative literals in the body of the remaining clauses.

Next, an interpretation M is a stable model for a Datalog¬ program P iff M is the least model of the program $\mathit{ground}_M(P)$. In general, Datalog¬ programs may have zero, one or many stable models. We shall see how multiplicity of stable models can be exploited to give a declarative account of non-determinism.

A word on the terminology and notation used in the fixpoint extension of the relational calculus—or first-order logic (FO) interpretation of the relational data model—is in order. 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 obtained as the limit of the sequence $J_0, \ldots, J_k, \ldots$, defined as follows (given an interpretation, or *database instance*, I):

- $J_0 = I(S)$, where $I(S)$ denotes the set of tuples of S in the interpretation 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.

It is important to observe that IFP converges in polynomial time on all input databases. First-order logic augmented with IFP is called *inflationary fixpoint logic* and is denoted by *FO+IFP*. The queries computed by the language *FO+IFP* are the so-called *fixpoint queries*, for which various equivalent definitions exist in the literature [7, 20].

Close connections exist between the fixpoint FO extensions and the Datalog extensions [4]: Datalog¬ under inflationary fixpoint semantics expresses exactly the fixpoint queries, i.e., it is equivalent to *FO+IFP*. This implies that Datalog¬ under the inflationary fixpoint semantics is strictly more expressive than Datalog with stratified negation, as the latter is known to be strictly included in *FO+IFP* [25]. Moreover, Datalog is equivalent to $FO^+ + IFP$, where FO^+ denotes the positive existential calculus, namely the negation-free, purely existential fragment of FO [8].

Following [4, 5], the complexity measures are functions of the size of the input database. We denote by *NDB-PTIME* the class of (non-deterministic) database queries that can be computed in polynomial time by a non-deterministic Turing transducer such that, for each input, each branch of the transducer's computation halts in an accepting state. It is important to avoid confusing the class *NDB-PTIME* of non-deterministic queries with the *DB-NP* of deterministic queries computed by non-deterministic devices. Intuitively speaking, any query in *NDB-PTIME* can be evaluated by means of *don't care* non-determinism: at any choice point, any non-deterministic choice will lead to some acceptable outcome—so no backtracking is needed. Analogously, *DB-PTIME* is the class of (deterministic) database queries that can be computed in polynomial time by a deterministic Turing transducer. A question, among others, that remains unsolved is whether a deterministic language exists, capable of expressing exactly the queries in *DB-PTIME*.

3 The Witness Operator

A non-deterministic extension of FO proposed by Abiteboul and Vianu is based on the *witness* operator [3, 4].

Let R be a relation (instance) over some attribute set $U = X \cup Y$, where X and Y are disjoint. Then, the witness operator, $W_X(R)$, selects one tuple $t \in R$ for each $t[X] \in \Pi_X(R)$. Therefore, W_X selects a *maximal* subset $R' \subseteq R$ satisfying the functional dependency $Y \to X$.

Example 3.1 Our TinyCollege database contains the two binary relations `student` and `professor`. The first column in `student` contains the student name and the second column gives his/her major. The first column in `professor` contains his/her name and the second column his/her area of research. In fact, let us say that our database consists only of the following facts:

$$\begin{array}{ll} \texttt{student(marc,ee).} & \texttt{professor(ohm,ee).} \\ \texttt{student(bianca,ee).} & \texttt{professor(bell,ee).} \end{array} \tag{1}$$

Next, assume that we want to write an expert system that assigns advisors to all students according to some simple rules. A first rule is that the major of a student must match his/her advisor's area. Then, eligible advisors can be expressed by the *FO* equivalent of:

$$\texttt{elig\_adv} = \pi_{\$1,\$4}(\texttt{student} \bowtie_{\$2=\$2} \texttt{professor}) \tag{2}$$

Thus for the previous database, we obtain:

```
elig_adv(marc,ohm).
elig_adv(marc,bell).
elig_adv(bianca,ohm).
elig_adv(bianca,bell).
```

But, the second rule is that *a student can only have one advisor.* Thus we must now use the operator $W_{\$2}$ to express the fact that, in the actual advisor relation, the second column (professor name) is functionally dependent on the first one (student name). Thus, the assignment of an actual advisor might be performed through the following relational algebra+witness (FO+W) expression:

$$\texttt{actul\_adv} = \texttt{W}_{\$2}\ \ (\pi_{\$1,\$4}(\texttt{student} \bowtie_{\$2=\$2} \texttt{professor})) \tag{3}$$

The result of executing this expression is non-deterministic. It yields a relation consisting of one of the four following sets of tuples: $\{(\texttt{marc},\texttt{ohm}),(\texttt{bianca},\texttt{ohm})\}$ $\{(\texttt{marc},\texttt{bell}),(\texttt{bianca},\texttt{ohm})\}$, $\{(\texttt{marc},\texttt{ohm}),(\texttt{bianca},\texttt{bell})\}$ $\{(\texttt{marc},\texttt{bell}),(\texttt{bianca},\texttt{bell})\}$. □

4 Datalog and Static Choice

The choice construct was first proposed by Krishnamurthy and Naqvi in [26]. 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.

In Datalog, eligible advisors of Example 3.1 can be computed by the following rule:

```
elig_adv(S,P)  ←  student(S,Major), professor(P,Major).
```

Then, using $\mathcal{LDL}$'s static choice construct [27], the goal `choice((S),(P))` can be added to the rule forcing the selection of a unique advisor for each student. The goal `choice((S),(P))` denotes that the functional dependency $S \rightarrow P$ must hold in `actual_adv`:

Example 4.1 Computation of unique advisor by choice rules.

```
actual_adv(S,P)  ←  student(S,Major), professor(P,Major),       (4)
                    choice((S),(P)).
```

The above program has the following four choice models:

$$\begin{aligned}
M_1 &= \{\texttt{actual\_adv(marc, ohm)}, \texttt{actual\_adv(bianca, ohm)}\} \cup DB,\\
M_2 &= \{\texttt{actual\_adv(marc, bell)}, \texttt{actual\_adv(bianca, ohm)}\} \cup DB,\\
M_3 &= \{\texttt{actual\_adv(marc, ohm)}, \texttt{actual\_adv(bianca, bell)}\} \cup DB,\\
M_4 &= \{\texttt{actual\_adv(marc, bell)}, \texttt{actual\_adv(bianca, bell)}\} \cup DB.
\end{aligned}$$

where DB denotes the set of atoms in the database, i.e.,

$$\begin{aligned} DB = \{ \; & \texttt{student(marc, ee)}, \texttt{student(bianca, ee)},\\ & \texttt{professor(ohm, ee)}, \texttt{professor(bell, ee)}\} \end{aligned} \qquad (5)$$

Thus, the choice rule above produces the same result as the relational expression with witness of Equation 3. □

Static choice can therefore be viewed as a construct for declaring and enforcing FDs in Datalog. In more operational terms, the programmer can also view choice as an actual table used to memorize every pair (S, P) produced by a rule such as that of Example 4.1; the table is used to compare each new pair against the old ones and discard those whose addition would violate the FD constraint. Facts produced by other rules with the same predicated name, `actual_adv`, are not compared against the old values in the table, since they are not subject to the FD constraint declared by the choice goal in Example 4.1.

We now explain the semantics of the static choice construct. We shall see that the qualification *static* for this choice operator stems from the observation that the choice is applied once and for all, after a preliminary fixpoint computation. 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 in Example 4.1. Basically, any choice model can be constructed in two steps. First, by using the rule (4) for `actual_adv` where the choice goal has been dropped, the set of all possible `actual_adv` facts is computed. Then, the basic operation of enforcing the FD constraints is performed, by selecting a maximal subset of the previous `actual_adv` facts that satisfies the FD $S \to P$.

For the sake of simplicity, assume that P contains 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 dropping all *choice* goals. Positive programs admit least models, and therefore we can consider the set M_P, denoting the least model of the positive program $PV(P)$. Next, we can construct the set C_P of `choice` facts which are related to instances of $B(Z)$ which are true in the least model, formally defined as follows:

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

where $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ are vectors of values such that $\mathbf{z} \supseteq \mathbf{x} \cup \mathbf{y}$, since $Z \supseteq X \cup Y$. Finally, we consider a maximal subset C'_P of C_P satisfying the FD $X \to Y$. Under these hypotheses, a choice model of P is defined as the least model of the program $P \cup C'_P$, i.e., program P extended with the set of `choice` facts specified by C'_P. Observe that the program $P \cup C'_P$ admits a least model as it is a positive program.

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 choice made.

Because of its static nature, this form of choice becomes ineffective with recursive rules, in the sense that the combined use of recursion and static choice does not buy extra expressiveness. The following section, devoted to illustrate the close relationships between the witness operator and the static choice, also clarifies the problem of static choice within recursion. The solution to this problem, which is the subject of Section 6, consists of defining a new declarative semantics for the choice construct.

5 Expressiveness Characterizations I

The similarities between the witness operator (in the context of FO) and the static choice operator (in the context of Datalog) become obvious once we concentrate on FO^+—the positive existential fragment of FO where negation and universal quantification is not used. These similarities will allow us to extend the natural equivalence, expressed by the following folk theorem [8, 5]:

Theorem 5.1 *A query is expressible in Datalog iff it is expressible in* $FO^+ + IFP$. □

Furthermore, safe FO^+ formulas, or non recursive Datalog programs, are equivalent to expressions in the monotonic fragment of relational algebra (i.e., RA without negation or division) [30].

Positive Existential Calculus and Witness The witness construct, when added to languages which support negation, enhances expressiveness, inasmuch as it allows to compute new *deterministic* and non-deterministic queries. Also, it has been recently shown that the addition of W to FO (without recursion) allows to compute deterministic queries not expressible in FO.

However, when added to FO^+, W does not extend the power of FO^+ in expressing deterministic queries. As discussed next, this observation hold both for FO^+ and $FO^+ + IFP$.

Theorem 5.2 *A deterministic query is expressible in FO^+ iff it is expressible in $FO^+ + W$.*

Proof. Consider a deterministic query Φ in $FO^+ + W$, and let I be the (unique) interpretation which satisfies Φ. Consider next the (deterministic) query Φ' in FO^+, obtained from Φ by removing any occurrence of a witness operator, and let J be the interpretation which satisfies Φ'. Clearly, we have $I \subseteq J$, by the definition of W. We now prove that indeed $I = J$.

Assume by contradiction that there exists a tuple α such that $\alpha \in J \setminus I$. By the definition of W, $I \cup \{\alpha\} \not\models FD$, where FD denotes the set of functional dependency constraints of the W operators in Φ. Next, consider $K \subset I$ such that $K \cup \{\alpha\} \models FD$. Observe that such a K exists: it can be constructed by removing from I the tuples which, together with α, violate the FD's. As a consequence of the fact that $FO^+ + W$ is monotonic, there exists a maximal interpretation I' satisfying FD such that $K \cup \{\alpha\} \subseteq I' \subseteq J$. To conclude the proof, it suffices to observe that, by construction, $I \neq I'$, and both I and I' satisfy Φ, which contradicts the fact that Φ is a deterministic query. $\Box$

Next, we observe that the above argument can be literally repeated in the case of the fixpoint extension of FO^+, thus obtaining the following result.

Theorem 5.3 *A deterministic query is expressible in $FO^+ + IFP$ iff it is expressible in $FO^+ + IFP + W$.* $\Box$

In summary, when added to negation-free deterministic query languages, the witness operator's only contribution is to allow the expression of non-deterministic queries, with no benefit w.r.t. deterministic ones.

A similar, but more far-reaching result is that $FO^+ + IFP + W$ is equivalent to its sublanguage L *consisting of the formulas where W does not occur within the fixpoint IFP operator.* In other words, in absence of negation, W has not effect within recursion—the only meaningful use of W is *after* a fixpoint computation.

Theorem 5.4 *A query is expressible in $FO^+ + IFP + W$ iff it is expressible in L.*

Proof. It suffices to show that any interpretation of the formula $IFP(\Phi, S)$, where Φ is a formula in $FO^+ + W$ and S is a relation occurring in Φ, coincides with the (unique) interpretation of $IFP(\Phi', S)$, where Φ' is the FO^+ formula obtained by dropping all occurrences of W in Φ. To this end, we observe that the monotonicity of FO^+ implies that the extension of Φ' at stage n of the fixpoint computation is included in the extension of Φ' at any later stage. As a consequence of the fact that the FD constraints of the witness operators are not maintained through the fixpoint computation, we have that eventually all candidate witnesses are selected. $\Box$

The above result is based on the observation that the witness does not enforce the FD as a global constraint upon the fixpoint iteration, but rather as as a local constraint on each step of the iteration. Therefore, the witness operator has no effect in presence of monotonicity. The absence of negation implies that no alternative is lost because of previously made choices, and therefore every alternative is eventually selected in the fixpoint computation.

Datalog with Static Choice Datalog programs with static choice are evaluated by first disregarding the choice construct, and then selecting some subset of the answer which satisfies the FD's. In other words, the choice is performed at the end of recursion. Say that we use the term *non-recursive choice* to denote the situation where the choice construct only appears in non-recursive rules. Then, we can state the following property:

Theorem 5.5 *A query is expressible in Datalog with static choice iff it is expressible in Datalog with non-recursive static choice.* □

This observation, together with Theorem 5.1, allows us to derive immediately an interesting corollary of Theorem 5.4, namely that Datalog with static choice and $FO^+ + IFP + W$ are equivalent query languages.

Theorem 5.6 *A query is expressible in Datalog with static choice iff it is expressible in* $FO^+ + IFP + W$. □

Theorem 5.6 extends theorem 5.1 and also confirms the affinity between the witness operator and the static choice operator—an affinity that can be traced back to their common reliance on the concept of functional dependencies as the basis for their semantics. These two constructs also share common limitations w.r.t. expressive power, which in turn can be traced back to Theorem 5.4 and Theorem 5.5, where it is shown that the usage of choice in (positive) recursive rules does not yield more expressive power than its usage in non-recursive ones. The dynamic choice operator, discussed in the next section, was indeed introduced to correct this weakness, and produced a significant augmentation of expressive power.

6 Datalog and the Dynamic Choice

A new semantics for the choice operator was proposed by Saccà and Zaniolo [29]. As discussed next, the new formulation called *dynamic choice* avoids the problems pertaining to the use of choice within recursive rules [16], that afflicted the version proposed in [26].

Dynamic choice is no longer defined using FD directly, rather choice programs 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 [29], therefore, we introduce the *stable version* of a choice program P, denoted by $SV(P)$, defined as the program with negation obtained from P as follows.

Definition 6.1 The *stable version* $SV(P)$ of a choice program P is obtained by the following transformation. Consider a choice rule r in P:

$$r : A \leftarrow B(Z),\ \mathit{choice}((X_1),(Y_1)),\ ...,\ \mathit{choice}((X_k),(Y_k)).$$

where:

(i) $B(Z)$ denotes the conjunction of all the non-choice goals of r, where Z is the vector of variables occurring in B, and

(ii) for $i \in [1,k]$, X_i, Y_i, denote vectors of variables from Z such that $X_i \cap Y_i = \emptyset$.

Then the original program P is transformed as follows:

1. replace r with a rule r' obtained by substituting the choice goals with the atom $chosen_r(W)$:
$$r' : A \leftarrow B(Z),\ chosen_r(W).$$
where, $W \subseteq Z$ is the list of all variables appearing in choice goals;

2. add the new rule:
$$chosen_r(W) \leftarrow B(Z),\ \neg diffChoice_r(W).$$

3. add, for each choice atom $choice((X_i),(Y_i))$ $(1 \leq i \leq n)$, the new rule:
$$diffChoice_r(W) \leftarrow chosen_r(W'),\ Y_i \neq Y_i'.$$
where (i) the list of variables W' is derived from W by replacing each variable $V \in Y_i$ with a new variable $V' \in Y_i'$ (i.e., by priming those variables), and (ii) $Y_i \neq Y_i'$ is a shorthand for the disjunction of inequalities $V_1 \neq V_1' \ \vee\ \ldots\ \vee\ V_{k_i} \neq V_{k_i}'$, where $Y_i = \langle V_1, \ldots, V_{k_i}\rangle$ (and $Y_i' = \langle V_1', \ldots, V_{k_i}'\rangle$.) □

Example 6.2 The stable version of the rule in Example 4.1 is the following, whose declarative reading is "a student can be assigned a professor provided that a different professor has not been assigned":

```
actual_adv(S,P)   ←  student(S,Major), professor(P,Major), chosen(S,P).
chosen(S,P)       ←  student(S,Major), professor(P,Major), ¬diffChoice(S,P).
diffChoice(S,P)   ←  chosen(S,P′), P ≠ P′.
```

(6)

□

Therefore, the pair of rules `chosen` and `diffchoice` provide an intuitive logical translation of FD constraints; this declarative formulation is semantically well-formed under stable model semantics. In fact, the program $SV(P)$ generated by this transformation has the following properties [29, 16]:

- $SV(P)$ has one or more stable models
- The `chosen` atoms in each stable model of $SV(P)$ obey the FDs defined by the choice goals.

The stable models of $SV(P)$ are called *dynamic choice* models for P.

When the choice rules are non-recursive (i.e., no choice rule is used in the definition of a recursive predicate), the semantics of static choice and dynamic choice coincide [29]. For instance, the program defined by the rules of Example 6.2 with the database defined by the facts of Example 4.1 has four stable models, corresponding to the four choice models of the program of Example 4.1 under the static choice semantics.

The significance of dynamic choice is two-fold. On the theory side, it establishes a clear relation between nonmonotonicity and non-determinism in logic languages; on the practical side it is critical in a deductive database system such as $\mathcal{LDL}$++, for expressing queries such as user-defined aggregates and depth-first graph traversals. These issues are best illustrated by simple examples where choice is used to modify and prune the transitive closure computation in a directed graph. For instance, the following program generates a spanning tree, rooted in a node `a` for a graph, where an arc from `X` to `Y` is represented by facts `g(X, Y)`:

Example 6.3 Spanning tree rooted in node `a`

$$\begin{array}{lll} \mathtt{st(nil,a).} & & \\ \mathtt{st(X,Y)} & \leftarrow & \mathtt{st(\_,X),\ g(X,Y), Y \neq a, choice((Y),(X)).} \end{array} \tag{7}$$

This program is basically the computation of the transitive closure of a graph constrained by addition of the choice goal and the pruning induced by the FD constraints. Thus, the condition $\mathtt{Y} \neq \mathtt{a}$ ensures that the in-degree for the node a is one, while the condition `choice((Y), (X))` ensures that the in-degree of all nodes generated by the recursive rule is also one.

Consider now the following graph:

```
g(a, b).
g(a, c).
g(b, c).
g(c, b).
```

If we consider the stable version of the spanning tree program, it is easy to see that it has 3 stable models, as follows:

```
st(nil, a);  st(a, b);  st(a, c)
st(nil, a);  st(a, b);  st(b, c)
st(nil, a);  st(a, c);  st(c, b)
```

□

Under the original, static version of choice (illustrated in Section 4), the semantics prescribes (i) the computation of the model of the program $PV(P)$ obtained from P by eliminating the choice goals and (ii) the extraction of a subset of this model to satisfy the FDs of the choice goals. Therefore, in addition of the above three models the static choice will also accept the solution:

```
st(nil, a);  st(b, c);  st(c, b)
```

This extra solution does not correspond to a tree but rather to an unsupported cycle $\mathtt{a} \leftrightarrows \mathtt{b}$ in the graph. In fact, this is an example of the greater expressive power and improved efficiency available through the use of dynamic choice, due to its ability in pruning and controlling the computation of a recursive predicate. This ability is lost with static choice, which can only perform a postselection on the nodes generated by the transitive closure. As a result, dynamic choice is both more efficient and more powerful than static choice. The gains in efficiency, due to the early pruning performed in the course of the computation, can be underscored even further by the following example—which is indeed an excursion out of pure Datalog.

Example 6.4 Shortest path

$$\begin{array}{lcl} \mathtt{st(a,0).} & & \\ \mathtt{st(Y,s(J))} & \leftarrow & \mathtt{st(X,J),\ g(X,Y),choice((X),(J)).} \end{array} \tag{8}$$

On the graph $\mathtt{g(a,b),g(b,b)}$ this program has an infinite number of static-choice models, each requiring an infinite computation. But, this program has only one dynamic-choice model, namely: $\mathtt{g(a,b),g(b,b),st(a,0),st(b,s(0))}$. □

7 Expressiveness Characterizations II

We show in this section that dynamic choice represents an improvement in expressive power, in that it is strictly more expressive than static choice. First, we observe that, trivially, Datalog with dynamic choice is at least as expressive as Datalog with static choice, since every query expressible in the latter language can be computed by a separate use of recursion and choice. Thus we have the following result:

Theorem 7.1 *If a query is expressible in Datalog with static choice then it is expressible in Datalog with dynamic choice.* □

To prove the strict inclusion, we consider the following program using dynamic choice that orders the elements of an EDB relation `u` in an arbitrary way.

Example 7.2 The program `ORD`[`u`] for ordering the elements of `u`:

$$\begin{array}{lll} r_1: & \mathtt{succ(min,min).} & \\ r_2: & \mathtt{succ(X,Y)} \quad \leftarrow & \mathtt{succ(\_,X),\ u(Y),} \\ & & \mathtt{choice((X),(Y)),choice((Y),(X)).} \end{array}$$

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

According to the semantics of dynamic choice, the binary relation `succ` defines a total, strict ordering over the input relation `u`. Viewing `succ` as a graph, we see that the first clause of program `ORD`[`u`] starts the fixpoint computation, by simply adding a loop arc on the distinguished element `min`. Consider now the second rule, and say that this, in the second stage of the computation, adds an arc (`min`, `a`) to `succ`, where `a` is an element in `u`. No other arc can be added at this stage, because it would violate the constraints imposed by choice. Likewise, at the third step, a new element from `u` will become the unique successor of `a`, and so on. Since the two choice goals enforce acyclicity (for the elements added by the second rule), at the end of the computation `succ` contains a simple path touching all elements in `u`.

With relation `succ(X,Y)` defining the immediate successor `Y` of an element `X`, the less-than relation $<$ can be constructed as the transitive closure of `succ` (also we eliminate the distinguished element `min`):

$$\begin{array}{lcl} \mathtt{X} < \mathtt{Z} & \leftarrow & \mathtt{succ(X,Z),\ u(X),\ u(Z).} \\ \mathtt{X} < \mathtt{Z} & \leftarrow & \mathtt{X} < \mathtt{Y},\ \mathtt{succ(Y,Z),\ u(Z).} \end{array} \tag{9}$$

From this, the definition of inequality follows:

$$\begin{array}{lll} \mathtt{X} \neq \mathtt{Y} & \leftarrow & \mathtt{X} < \mathtt{Y}. \\ \mathtt{X} \neq \mathtt{Y} & \leftarrow & \mathtt{Y} < \mathtt{X}. \end{array} \tag{10}$$

It is known that the inequality query cannot be expressed in $FO^+ + IFP$ (see, e.g., [5]), so by Theorem 5.3 we conclude that the inequality query cannot be expressed in $FO^+ + IFP + W$ either, as it is a deterministic query. But the expressive power of Datalog with static choice is, by Theorem 5.6 equivalent to $FO^+ + IFP + W$, so we have the following:

Theorem 7.3 *The inequality query is not expressible in Datalog with static choice.* □

Therefore, Datalog with dynamic choice is strictly more expressive than Datalog with static choice. Even so, Datalog with dynamic choice cannot express express all $PTIME$ queries, inasmuch as, as proven later, it cannot express nonmonotonic queries. However, all non-deterministic polynomial-time queries become expressible provided that stratified negation is supported by the language: thus Datalog$^{\neg s}$ with dynamic choice can express all *NDB-PTIME* queries. In fact we have the following result [19]:

Theorem 7.4
A query is expressible in Datalog$^{\neg s}$ with dynamic choice iff it is expressible in $FO+IFP+W$.
□

As a consequence of the key result by Abiteboul and Vianu that $FO + IFP + W$ expresses precisely *NDB-PTIME* [3, 4], Theorem 7.4 implies that also Datalog$^{\neg s}$ with dynamic choice is a precise characterization of *NDB-PTIME*. Therefore, all deterministic *DB-PTIME* queries can be expressed in this language. This conclusion is consistent with the fact that Datalog$^{\neg s}$ expresses all $PTIME$ queries when a total order exists on the database [21, 31]: in fact, Example 7.2 illustrates how to generate such an order using dynamic choice.

8 Constructive Semantics

This section is devoted at providing a constructive characterization of dynamic choice. Towards this goal, we define a general operator Ψ_P for non-deterministic bottom-up computations of a choice program P, and study its properties. The main result, stated in Corollary 8.8, is that the proposed constructive semantics coincide with the stable choice model semantics discussed in the previous section.

Two alternative semantics for dynamic choice, called *lazy* and *eager* dynamic choice, are then identified as instances of the general operator Ψ_P, in such a way that their relevant properties are directly derived. The *lazy* operator yields a constructive formalization of the dynamic choice construct discussed in the previous section.

We develop our constructive semantics of choice programs with reference to proper Datalog (choice) programs, i.e., function-free programs over a finite universe. However, all results presented in this section also hold in the more general case of arbitrary choice programs with function symbols, i.e., programs over infinite universes. This issue is addresses in the final part of this section, and requires the development of a notion of fairness of non-deterministic computations.

Given a choice program P, we denote by T_{P_C} the immediate consequence operator associated with the *chosen* rules in the stable version $SV(P)$ of P, i.e., the rules introduced at step 2 of the transformation of Definition 6.1. Analogously, we denote by T_{P_H} the immediate consequence operator associated with the non-*chosen* rules in $SV(P)$. Therefore, we have that, for any interpretation I:

$$T_{SV(P)}(I) = T_{P_H}(I) \ \cup \ T_{P_C}(I).$$

Moreover, when we refer to an interpretation of a choice program, we actually mean an interpretation of its stable version. Given a choice program P and an interpretation I, we write $I \models FD_P$ if, for any choice rule r of P, the set of $chosen_r$ atoms of I satisfies the FD constraint specified by the choices in rule r. We are now ready to introduce a general operator for non-deterministic fixpoint computations of choice programs.

Definition 8.1 Given a choice program P, its *non-deterministic immediate consequence operator* Ψ_P is a map from interpretations to sets of interpretations defined as follows:

$$\Psi_P(I) \ = \ \{J \ \cup \ T_{P_H} \uparrow \omega(J) \ \mid \ J \in \Gamma_P(I)\}$$

where

$$\Gamma_P(I) \ = \ \{I \cup \Delta I \ \mid \ \emptyset \subset \Delta I \subseteq T_{P_C}(I) \setminus I \ \wedge \ \Delta I \models FD_P \ \} \ \cup \ \{I \ \mid \ T_{P_C}(I) \setminus I = \emptyset\}.$$

□

Informally, the Ψ_P operator, when applied to an interpretation I, behaves as follows:

- first, T_{P_C} is used to derive from I all the possible choices;
- second, Γ_P is used to construct the set of all possible ways of augmenting I with a new set of *admissible* choices ΔI, which do not violate the FD of choice rules; if such ΔI is empty, then the singleton set $\{I\}$ is returned;
- finally, for each set in $\Gamma_P(I)$, a saturation using T_{P_H} is performed, in order to derive all consequences of the operated choices.

Clearly, non-determinism is due to the Γ_P operator, which returns a multiplicity of sets of admissible choices, which satisfy the functional dependency constraints placed by choice rules. Notice that, at this stage, we are not specific on what kind of subsets of choices are selected. For instance, it is possible that two such subsets are included in each other.

Observe that the usage of the ordinal ω in the above definition is not problematic, since saturation via T_{P_H} is is guaranteed to occur at some finite stage for Datalog programs. More precisely, we have that $T_{P_H} \uparrow \omega(J) = T_{P_H} \uparrow k(J)$ for some finite k $(< \omega.)$

In the definition of $\Gamma_P(I)$, ΔI is not empty if $T_{P_C}(I) \setminus I$ is not empty; this reflects the fact that, if there are admissible new choices, then at least one such choice has to be made. As a final comment to Definition 8.1, we observe that in the definition of Γ_P, the check for functional dependencies can be specified simply as $\Delta I \models FD_P$, instead of $(I \cup \Delta I) \models FD_P$; the reason is that for each single tuple $t \in \Delta I$, $I \cup \{t\} \models FD_P$ $diffChoice$ rules in $SV(P)$ disallow such a conflict. It is therefore sufficient to check that each pair in ΔI of such rules do not conflict with each other with respect to the given FDs.

The Ψ_P operator formalizes a single step of an ideal bottom-up non-deterministic computation of a choice program. Instead of defining the powers of Ψ_P, it is technically more convenient to define directly the notion of a non-deterministic computation based on the Ψ_P operator.

Definition 8.2 Let P be a choice program. A *(non-deterministic) computation based on* Ψ_P is a sequence $\langle I_n \rangle_{n \geq 0}$ of interpretations such that:

i. $I_0 = \emptyset$,

ii. $I_{n+1} \in \Psi_P(I_n)$, for $n \geq 0$.

□

The following result points out some basic properties of non-deterministic computations, namely that they are inflationary, and they preserve the FD's.

Lemma 8.3 *Let $\langle I_n \rangle$ $(n \geq 0)$ be a non-deterministic Ψ_P-based computation, for a choice program P. Then, for $n \geq 0$*

(1) $I_n \subseteq I_{n+1}$,

(2) $I_n \models FD_P$.

Proof. (1) follows from the fact that, for any pair of interpretations I and J such that $J \in \Psi_P(I)$, we have $I \subseteq J$, which is a direct consequence of the definition of Ψ_P.

The proof of (2) is by induction on n. In the base case, we have $\emptyset \models FD_P$, which holds trivially.

In the induction case, two subcases arise. If $T_{P_C}(I_n) \setminus I_n$ is empty, then we have $I_{n+1} = I_n$ by the definition of Ψ_P, and therefore the thesis directly follows from the induction hypothesis that $I_n \models FD_P$. If $T_{P_C}(I_n) \setminus I_n$ is not empty, we have to prove that $I_n \cup \Delta I \models FD_P$ for any non empty subset ΔI of $T_{P_C}(I_n) \setminus I_n$. Consider a *chosen*-atom $chosen_r(x, y) \in T_{P_C}(I_n) \setminus I_n$. Consider the rule $chosen_r(x, y) \leftarrow B(x, y), \neg diffChoice_r(x, y)$ from $SV(P)$: the fact that $chosen_r(x, y) \in T_{P_C}(I_n)$ implies that the body of such a rule holds in I_n, and in particular $diffChoice_r(x, y) \notin I_n$. As a consequence of the facts that $SV(P)$ contains any instance of the *diffChoice* rule $diffChoice_r(X, Y) \leftarrow chosen_r(X, Y'), Y \neq Y'$, and that I_n is saturated with respect to such rule, we obtain that $chosen_r(x, y') \notin I_n$, for every $y' \neq y$, Therefore, the choice $chosen_r(x, y)$ does not violate the FD's, i.e., $I \cup \{chosen_r(x, y)\} \models FD_P$. To conclude the proof, it suffices to observe that the above reasoning can be repeated for any *chosen* atom in ΔI, and that $\Delta I \models FD_P$ by the definition of Ψ_P. □

Given a computation $\langle I_n \rangle$ $(n \geq 0)$, we define its (ω) limit $lim_{n \geq 0} \langle I_n \rangle = \bigcup_{n<\omega} I_n$. It should be noted that, in our case of function-free Datalog programs, the limit of a computation is reached at some finite stage. Therefore, we have that, for some $\bar{k} < \omega$, $lim_{n\geq 0}\langle I_n\rangle = I_{\bar{k}}$, and $I_n = I_{\bar{k}}$ for every $n \geq \bar{k}$.

Our next concern is to correlate the limits of computations with the fixpoints of Ψ_P, in order to justify the fact that non-deterministic computations are in fact fixpoint computations. We need here an unconventional notion of fixpoint, due to the fact that Ψ_P maps interpretations to sets of interpretations (a multivalued mapping):

Definition 8.4 An interpretation I is a fixpoint of Ψ_P if $\Psi_P(I) = \{I\}$. □

In other words, I is a fixpoint of Ψ_P if the operator behaves deterministically on I, and I itself is obtained as a result. In order to justify the above notion of fixpoint of a non-deterministic operator, we observe here that this notion coincides with that of a fixpoint of the ordinary immediate consequence operator $T_{SV(P)}$ associated with the stable version $SV(P)$ of a choice program P.

Lemma 8.5 *Let P be a choice program. Then M is a fixpoint of Ψ_P iff M is a fixpoint of $T_{SV(P)}$.*

Proof. First, observe that, for any interpretation I:

$$T_{SV(P)}(I) = I \quad \text{iff} \quad T_{P_C}(I) = I_C \text{ and } T_{P_H}(I) = I_H, \tag{11}$$

where I_C denote the set of *chosen* atoms in I and I_H denote the set of non-*chosen* atoms in I. Consider a fixpoint I of $T_{SV(P)}$. Now, (11) implies that $T_{P_C}(I) \setminus I = \emptyset$, and therefore $\Gamma_P(I) = \{I\}$. Also, (11) implies $T_{P_H} \uparrow \alpha(I) \ \cup \ I = I$. Therefore $\Psi_P(I) = \{I\}$, so I is a fixpoint of Ψ_P.

Conversely, assume that $\Psi_P(I) = \{I\}$. Then clearly $T_{P_H}(I) = I_H$ and $T_{P_C}(I) = I_C$ by the definition of Ψ_P, so the fact that I is a fixpoint of $T_{SV(P)}$ follows from (11). □

The next two results show that the limit of a non-deterministic computation is a dynamic choice model and vice versa, thus providing a notion of soundness and completeness w.r.t. the stable model semantics of choice programs.

Theorem 8.6 (Soundness) *Let P be a choice program. If M is the limit of a non-deterministic Ψ_P-based computation, then M is a dynamic choice model for P.*

Proof. Let $M = I_{\bar{k}}$ be the limit of a computation $\langle I_n\rangle_{n\geq 0}$, and consider the reduced program $P' = ground_M(SV(P))$. We have to show that M is the least model of P'. This is established by the following Claims 1 and 2.

Claim 1. *M is a model of P'.*

To this end, we now prove that M is a fixpoint of Ψ_P which, together with Lemma 8.5, implies that M is a fixpoint of $T_{SV(P)}$, and *a fortiori* a model of P'. Assume by contradiction that $\Psi_P(M) \neq \{M\}$. This, together with Theorem 8.3(1) implies that there exists $J \in \Psi_P(M)$ with $M \subset J$. Therefore, for some ground instance $H \leftarrow B$ of a clause r from $SV(P)$, we have

$$M \models B \tag{12}$$
$$M \not\models H \tag{13}$$
$$J \models H. \tag{14}$$

By (12), we have that $I_{\bar{k}} \models H$. Two cases then arise.

Case 1. r is not a *chosen* clause.
Then we conclude that $I_{\bar{k}+1} \models H$ by the definition of Ψ_P, which, together with the fact that $M = I_{\bar{k}}$ is the limit of the computation, contradicts (13).

Case 2. r is a *chosen* clause.
Then $I_{\bar{k}} \cup \{H\} \models FD_P$, otherwise (14) is contradicted. By Definition 8.1, a non-empty ΔI exists ($= \{H\}$), and therefore each interpretation $K \in \Psi_P(I_{\bar{k}})$ strictly includes $I_{\bar{k}}$. This contradicts the fact that $M = I_{\bar{k}}$ is the limit of the computation, as any possible choice for $I_{\bar{k}+1}$ is such that $I_{\bar{k}} \subset I_{\bar{k}+1}$.

Claim 2. *M is the least model of P'.*

To this end, assume by contradiction that $J (\subset M)$ is a model of P', and consider an atom $H \in M \setminus J$. Let n (> 1) be the stage of the computation at which H is inferred, i.e., such that $I_n \models H$ and $I_{n-1} \not\models H$, and consider the ground clause $H \leftarrow B$ of $SV(P)$ such that $I_{n-1} \models B$, and the corresponding clause $H \leftarrow B'$ of P'. Two cases arise.

Case 1. H is not a chosen atom.
As $J \not\models H$, we have that $J \not\models A$, for some atom A of B'.

Case 2. H is a chosen atom.
Let $H = chosen_r(t)$. As a consequence of Theorem 8.3(2), $M \models FD_P$, which implies that $M \not\models diffChoice_r(t)$. Therefore, all atoms in the body B' of the clause from P' are not *chosen* or *diffChoice*, by the definition of $SV(P)$ and the stability transformation. Again, as $J \not\models H$, we have that $J \not\models A$, for some atom A of B.

We can therefore repeat the construction of cases 1 and 2 at most $n-1$ times before finding an atom A, inferred by a unit clause $A \leftarrow$ from P', such that $J \not\models A$. This contradicts the fact that J is a model of P'. □

Theorem 8.7 (Completeness) *Let P be a choice program. If M is a dynamic choice model for P, then there exists a computation based on Ψ_P, which has M as its limit.*

Proof. Let M be a stable model of $SV(P)$. i.e., the least model of $P' = ground_M(SV(P))$. First, observe that

$$M \quad \models \quad FD_P. \tag{15}$$

To prove (15), assume by contradiction that $M \models choice_r(t) \wedge choice_r(t')$ such that $choice_r(t)$ and $choice_r(t')$ violate the FD's. Then, by the definition of $SV(P)$, we have that $M \models diffChoice_r(t') \wedge diffChoice_r(t)$. By the stability transformation, there is no clause in P' with either $choice_r(t)$ or $choice_r(t')$ in the head, thus contradicting that M is the least model of P'.

Next, let P'_H be the program consisting of the non-*chosen* rules of P', and P'_C be the program consisting of the *chosen* rules of P'. We can define the following sequence of interpretations $\langle M_n \rangle_{n \geq 0}$:

i. $M_0 = \emptyset$,

ii. $M_{n+1} = T_{P'_H} \uparrow \omega(T_{P'_C}(M_n) \cup M_n)$, for $n > 0$.

Clearly, $M = T_{P'} \uparrow \omega = \bigcup_{n \geq 0} M_n$, since P' is a definite program, and therefore the powers of $T_{P'_H}$ and $T_{P'_C}$ can be arbitrarily interleaved to obtain $T_{P'} \uparrow \omega$. Therefore, by (15), we get $M_n \models FD_P$ for $n \geq 0$. As a consequence, $T_{P'_C}(M_n)$ is a subset of $T_{P_C}(M_n)$ which satisfies the FD's, and therefore it can be selected as a ΔI in the definition of Ψ_P. To conclude the proof, it suffices to notice that, by construction, $\langle M_n \rangle$ $(n \geq 0)$ is a computation based on Ψ_P. □

As a consequence of Theorems 8.6 and 8.7, we obtain the following corollary:

Corollary 8.8 (Characterization) *Let P be a choice program, and M an interpretation. Then M is a dynamic choice model for P iff M is the limit of a computation based on Ψ_P.* □

Next, we can show that the limits of computations are *minimal* fixpoints of Ψ_P. This result, stated in the next theorem, is a direct consequence of the Soundness Theorem 8.6, Lemma 8.5, and the fact that every stable model of a program P is a minimal fixpoint of T_P.

Theorem 8.9 *Let P be a choice program, and M be the limit of a computation $\langle I_n \rangle$, $(n \geq 0)$, based on Ψ_P. Then M is a minimal fixpoint of Ψ_P.* □

In summary, the notions of dynamic choice models and of limits of Ψ_P-based computations coincide, and these are minimal fixpoints. Therefore, the declarative semantics of choice programs and their procedural (fixpoint) semantics coincide. Such an equivalence does not extend to minimal-fixpoint semantics, since there are programs with minimal fixpoints which are not dynamic choice models. As an example, consider the following program P:

```
p(a) ← p(a).
p(b).
q(X) ← p(X), choice((), (X)).
```

and its stable version $SV(P)$:

```
p(a) ← p(a).
p(b).
q(X) ← p(X), chosen(X).
chosen(X) ← p(X), ¬diffChoice(X).
diffChoice(X) ← chosen(X′), X ≠ X′.
```

It is readily checked that Ψ_P has two minimal fixpoints:

$$\begin{aligned} M_1 &= \{\texttt{p(b)}, \texttt{q(b)}, \texttt{chosen(b)}, \texttt{diffChoice(a)}\} \\ M_2 &= \{\texttt{p(b)}, \texttt{p(a)}, \texttt{q(a)}, \texttt{chosen(a)}, \texttt{diffChoice(b)}\} \end{aligned}$$

but M_2 is not a dynamic choice model of P, and cannot be obtained as the limit of any computation. In fact, the atom `chosen(a)` in M_2 is supported by the fact `p(a)` which cannot be computed by a bottom up computation starting with the empty set.

Two remarks arise from this discussion. First, the non-coincidence of the procedural (fixpoint-based) semantics and the minimal-fixpoint semantics underscores the suitability of stable models as the formal basis for the non-deterministic semantics of choice programs. For instance, completion semantics also allow multiple models, but these coincide with the minimal-fixpoint semantics of the programs. Therefore, for the simple example shown above, the completion semantics is undesirable, since it allows more models than those computable with the intended procedural semantics.

As a second remark, it is natural to ask ourselves whether reasonably large classes of choice programs exist for which the stable model and the minimal-fixpoint (or completion) semantics do coincide. We believe that the answer is affirmative, and that a promising way to go is to extend to choice programs the notion of acceptable programs from [6], as for this class of programs we have coincidence between stable model and completion semantics. This represents a subject for further research.

8.1 Lazy Versus Eager Dynamic Choice

The non-deterministic Ψ_P operator generates a family of different operators, obtained by suitably restricting the set Γ_P. We now introduce two specializations of the Ψ_P, and provide their characterizations. The first one is the *lazy* version of dynamic choice, while the second is called *eager* dynamic choice. Lazy choice is obtained by letting the ΔI's in the definition of Γ_P be minimal, while greedy choice is obtained by only taking maximal ΔI's, as follows:

Definition 8.10
(1) The *lazy* operator Ψ_P^L is defined as the instance of Ψ_P where ΔI is a singleton:

$$\Psi_P^L(I) \quad = \quad \{J \cup T_{P_H} \uparrow \omega(J) \ \mid \ J \in \Gamma_P(I)\}$$

where

$$\Gamma_P(I) \quad = \quad \{I \cup \{H\} \ \mid \ \{H\} \subseteq T_{P_C}(I) \setminus I\} \ \cup \ \{I \ \mid \ T_{P_C}(I) \setminus I = \emptyset\}.$$

(2) The *eager* operator Ψ_P^E is defined as the instance of Ψ_P where ΔI is maximal:

$$\Psi_P^E(I) \quad = \quad \{J \cup T_{P_H} \uparrow \omega(J) \ \mid \ J \in \Gamma_P(I)\}$$

where

$$\begin{aligned}\Gamma_P(I) \quad = \quad & \{I \cup \Delta I \ \mid \ \emptyset \subset \Delta I \subseteq T_{P_C}(I) \setminus I \ \wedge \ \Delta I \models FD_P \ \wedge \Delta I \text{ is maximal } \} \\ & \cup \{I \ \mid \ T_{P_C}(I) \setminus I = \emptyset\}.\end{aligned}$$

□

Lazy choice and eager choice exhibit very different properties. In particular, in the next section, we will show that the latter is more powerful than the former. Moreover, the lazy operator is non-deterministically complete in the sense of Theorem 8.7; in fact every choice computation can be emulated by one where choices are made one-at-a-time. Thus we can state the following:

Lemma 8.11 (Completeness) *Let P be a choice program. Then, any dynamic choice model of P is the limit of a computation based on Ψ^L_P.* □

On the other hand, the procedural semantics based on the eager operator is not complete, and some of the possible solutions might in fact be lost. Take for instance Example 6.3. Under eager semantics both $\mathtt{st(a,b)}$ and $\mathtt{st(a,c)}$ are produced in the first step of the recursive computation. Thus, only the first of the three solutions in Example 6.3 is obtained under eager choice. Therefore, under static choice, lazy choice and eager choice this program has, respectively, four choice models, three choice models and one choice model. In fact, stricter non-deterministic semantics here yields a greater expressive power.

Therefore, we are witnessing an interesting phenomenon with non-deterministic semantics: as the set of canonical models becomes smaller expressive power increases. This is further illustrated by the following example taken from [16], where we emulate negation using eager choice. The following choice program defines relation `not_p` as the complement of a relation `p` with respect to a universal relation `u` (for simplicity, say that both `p` and `u` are extensional relations):

Example 8.12 The choice program $\mathtt{NOT}[\mathtt{p},\mathtt{u}]$ consists of the following rules:

$$
\begin{array}{ll}
\mathtt{R}_1: & \mathtt{not\_p(X)} \leftarrow \mathtt{comp\_p(X,1)}.\\
\\
\mathtt{R}_2: & \mathtt{comp\_p(X,I)} \leftarrow \mathtt{tag\_p(X,I), choice((X),(I))}.\\
\\
\mathtt{R}_3: & \mathtt{tag\_p(nil,0)}.\\
\mathtt{R}_4: & \mathtt{tag\_p(X,0)} \leftarrow \mathtt{p(X)}.\\
\mathtt{R}_5: & \mathtt{tag\_p(X,1)} \leftarrow \mathtt{u(X), comp\_p(\_,0)}.
\end{array}
$$

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

According to the operational semantics of eager choice, we obtain a set of answers where $\mathtt{comp\_p(x,1)}$ 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 $(\mathtt{X} \rightarrow \mathtt{I})$, 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 $\mathtt{tag\_p(nil,0)}$ is needed to cope with the case that relation `p` is empty.

More precisely, in the first saturation phase, the facts $\mathtt{tag\_p(nil,0)}$ and $\mathtt{tag\_p(x,0)}$ are inferred, for every `x` in the extension of relation `p`. In the following choice phase, the facts $\mathtt{chosen(x,0)}$ are derived again for every `x` in the extension of `p`, since all possible choices are exercised. In the second saturation phase, we infer $\mathtt{comp\_p(x,0)}$ for every `x` in the extension of `p`, and $\mathtt{tag\_p(x,1)}$ for every `x` in `u`. In the following choice phase, the facts $\mathtt{chosen(x,1)}$ are chosen in a maximal way to satisfy the FD, i.e., for every `x` *not* in the extension of `p`, since every `x` in `p` has already been chosen with tag 0. In the third saturation step the extension of `not_p` becomes the complement of `p` with respect to `u`.

Essentially, this example shows that the eager dynamic choice offers a flexible mechanism for handling the control needed to emulate the difference between two relations. It is shown in

[11] 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. Thus, eager choice can compute nonmonotonic deterministic queries. On the other hand, we shall see in Section 9 that any deterministic query specified using the lazy dynamic choice is indeed monotonic.

8.2 Function Symbols and Programs with Infinite Universe

The key properties of the fixpoint semantics, discussed above, also extend to the more general case of logic programs with *function symbols*, i.e., programs whose underlying universe is infinite. For the sake of generality, and because many logic-based languages, such as $\mathcal{LDL}$, support function symbols, we briefly discuss this generalization next.

The main problem with infinite non-deterministic computations is the possibility of an *unfair* behavior w.r.t. choices: this occurs when some choice is considered infinitely often during a computation, but is never made, since other choices are always preferred. This behavior, which is not possible over finite universes, may produce limits of infinite computations, which are *not* choice models (or fixpoints), thus contradicting soundness. To solve this problem, and guarantee that infinite computations converge to a fixpoint, we next define the notion of *fairness with respect to choices.*

Definition 8.13 Let P be a choice program. A non-deterministic computation $\langle I_n \rangle_{n \geq 0}$ based on Ψ_P is *fair* iff there is no *chosen* atom H such that $H \in T_{P_C}(I_n)$ for infinitely many $n \geq 0$. $\square$

Conversely, an unfair computation is one which never selects a particular choice although this is offered an infinite number of times during the computation. Therefore, all finite computations are fair. However, for infinite computations, the fairness assumption is needed to ensure that every admissible choice is eventually selected.

It should be observed here that, while in general a computation based on Ψ_P can be either fair or unfair, depending on the choices made during the computation itself, any computation based on the eager operator Ψ_P^E is always fair, as the criterion of "ΔI maximal" entails that all possible choices are made at eachs stage. On the contrary, lazy computations are not necessarily fair, and therefore lazy computations must be scheduled or monitored explicitly to ensure that computations are not terminated until a complete dynamic choice model is reached.

It is readily checked that, that Theorem 8.7 (completeness) generalizes to programs with function symbols with no change, while Theorem 8.6 (soundness) generalizes in the case of fair computations. Thus, we obtain the following generalization of the Characterization Theorem 8.8:

Corollary 8.14 (Characterization II) *Let P be a choice program, possibly with function symbols, and M an interpretation. Then M is a dynamic choice model for P iff M is the limit of a* fair *computation based on Ψ_P.* $\square$

A final remark concerns the effectiveness of non-deterministic computations in the case of infinite universes. The point here is the definition of Ψ_P, which we recall from Definition 8.1:

$$\Psi_P(I) \quad = \quad \{J \;\cup\; T_{P_H} \uparrow \omega(J) \;\mid\; J \in \Gamma_P(I)\}$$

Due to the ω-power of T_{P_H}, a transfinite computation is required in the evaluation of a single step of Ψ_P in order to compute any stable model. While in the case of finite universes this is not a problem, in the case of infinite universes the saturation of T_{P_H} may not be effectively computed. For this reason, practical implementations of the Ψ_P operator truncate the saturation of T_{P_H} at some finite stage. Fortunately, this policy is not problematic from the point of view of soundness (the proof of Theorem 8.6 goes through, verbatim, if we replace ω with 1 in the definition of Ψ_P). However, completeness is compromised, in that this policy in general anticipates choices, and may therefore prevent to reach certain stable models.

9 The Expressive Power Hierarchy for Choice Constructs

The next result, which characterizes the expressiveness of eager dynamic choice, has been proven in [12, 17].

Theorem 9.1 *A query is expressible in $FO + IFP + W$ iff it is expressible in Datalog with eager dynamic choice.* □

Therefore, Datalog with eager choice represents a very powerful language, inasmuch as it can express all queries in *NDB-PTIME*. Datalog with lazy dynamic choice is characterized by the Ψ_P^L operator.

As a consequence of Lemma 8.3(1), any computation with the Ψ_P operator is inflationary and, a fortiori, any query supported by such a computation is polynomial. As a consequence of Theorem 9.1, we obtain that any query expressible with lazy dynamic choice is also expressible using eager dynamic choice, as the latter expresses all (non-deterministic) polynomial queries.

Theorem 9.2 *If a query is expressible in Datalog with lazy dynamic choice then it is expressible in Datalog with eager dynamic choice.* □

The inverse property does not hold since Datalog with eager dynamic choice is strictly more powerful than Datalog with lazy dynamic choice or unrestricted dynamic choice. This is due to the monotonic nature of dynamic choice, proven in Theorem 9.3. As customary, we can partition the predicates in a program P in extensional and intensional predicates. Let $edb(P)$ (resp. $idb(P)$) denote the clauses of P having extensional (resp. intensional) predicate names. Thus, $idb(P)$ contains all the rules of P, while $edb(P)$ only contain facts: the database facts.

Lemma 9.3 *Let P and P' be choice programs where $idb(P') = idb(P)$ and $edb(P') \supseteq edb(P)$. Then, for every M that is the limit of a computation of Ψ_P, there exists a computation of Ψ'_P which has as limit $M' \supseteq M$.*

Proof. Let I be the limit of a computation $\langle I_n \rangle_{n \geq 0}$ of P. We construct a computation $\langle J_n \rangle_{n \geq 0}$ of P' with limit J such that $I_n \subseteq J_n$ for any $n \geq 0$, which directly implies the thesis. The construction is by induction on n. We maintain the following invariants, for any $n \geq 0$:

$$I \;\cup\; (J_n \setminus I_n) \quad \models \quad FD_P \tag{16}$$

$$I \;\cap\; (J_n \setminus I_n) \quad = \quad \emptyset. \tag{17}$$

In other words, we require that the extra choices made in the computation over P' do not conflict with the choices made in the computation over P.

In the base case, we clearly put $J_0 = \emptyset$. In the induction case, consider $I_{n+1} \in \Psi_P(I_n)$, i.e., $I_{n+1} = T_{P_H} \uparrow \omega(I_n \cup \Delta I_n)$, for some subset ΔI_n of $T_{P_C}(I_n) \setminus I_n$ such that $\Delta I_n \models FD_P$.

We now show

$$\Delta I_n \quad \subseteq \quad T_{P'_C}(J_n) \setminus J_n. \tag{18}$$

Due to (17) and the fact that ΔI_n and I_n are disjoint, we have that ΔI_n and J_n are disjoint, so it suffices to how that $\Delta I_n \subseteq T_{P'_C}(J_n)$. Assume by contradiction that some *chosen* atom $chosen(t)$ violates this inclusion. Therefore, the clause $chosen(t) \leftarrow B(t), \neg diffChoice(t)$ is such that $I_n \models B(t), \neg diffChoice(t)$, and $J_n \not\models B(t), \neg diffChoice(t)$. By the induction hypothesis that $I_n \subseteq J_n$ we have $J_n \models B(t)$, so $J_n \not\models \neg diffChoice(t)$, i.e., $diffChoice(t) \in J_n$. Therefore, we have $chosen(t') \in J_n$ for some $t' \neq t$ such that $chosen(t) \wedge chosen(t') \not\models FD_P$. So we get a contradiction with (16), as $chosen(t) \in \Delta I_n$ and, a fortiori, in I.

Next, as a consequence of (18) we can choose a subset ΔJ_n of $T_{P'_C}(J_n) \setminus J_n$ such that $\Delta I_n \subseteq \Delta J_n$ in order to satisfy (16) and (17), and let J_{n+1} be:

$$J_{n+1} = T_{P'_H} \uparrow \omega(J_n \cup \Delta J_n).$$

To conclude that $I_{n+1} \subseteq J_{n+1}$ it suffices to observe that $T_{P_H}(H) \subseteq T_{P'_H}(K)$ for $H \subseteq K$. $\square$

Thus, dynamic choice programs define monotonic transformations on the underlying database: the same can also be said for the transformation defined by lazy dynamic choice rules (a proof follows immediately from the completeness Lemma 8.11, or directly from a construction similar to that used in the previous Lemma for the choice operator Ψ^L_P).

As a consequence of this Lemma, we have that on *deterministic* queries, i.e., those queries for which exactly one computation exists, the general choice operator and the lazy choice operator behave monotonically in the standard sense. In other words, these operators can compute only deterministic queries that are monotonic. On the contrary, the eager operator Ψ^E_P is not monotonic, due to the fact that the commitment to maximal sets of admissible choices reduces the non-determinism in such a way that the construction in the proof of Lemma 9.3 is no longer possible. An example of a nonmonotonic deterministic query computed by means of the eager operator is the negation/complement query in Section 4.3.

As a consequence of Lemma 9.3, Datalog with lazy dynamic choice has a monotonic semantics, in the sense that the query associated with a lazy dynamic choice program yields a larger output when applied to a larger input database. As a consequence of this fact, the negation query cannot be computed.

Theorem 9.4 *The negation query is not expressible in Datalog with lazy dynamic choice.* $\square$

Therefore, eager dynamic choice is strictly more expressive than lazy dynamic choice. Because of its expressive power and its simpler implementation eager choice is normally preferable as a practical construct. However, lazy choice find important applications in the definition of other constructs, such as monotonic aggregates [32].

10 Conclusions

This paper has elucidated the semantic and computational aspects of non-deterministic constructs of Datalog languages. From a semantic viewpoint, there is a simple connection between choice constructs and stable models of nonmonotonic logic. Furthermore we have provided a fixpoint-based operational semantics for programs with choice, where the computation of each choice model is polynomial in the size of the database.

In fact, we have studied three different versions of choice (i.e, static choice, lazy dynamic choice and eager dynamic choice) and shown that they define a strict hierarchy that parallels the expressive power hierarchy of FO languages with inflationary fixpoint and witness operator. This hierarchy, which follows from Theorems 5.6, 7.1, 7.3 ,9.2, 9.4, 9.1, and 7.4, is summarized by following table where successive rows define languages of increasingly higher expressiveness:

Datalog with static choice	$=$	$FO^+ + IFP + W$
$\subset$		$\subset$
Datalog with lazy dynamic choice	$=$	...
$\subset$		$\subset$
Datalog with eager dynamic choice	$=$	$FO + IFP + W$
Datalog$^{\neg_s}$ with lazy dynamic choice	$=$	$FO + IFP + W$

The typical situation that a language designer has to face is a tradeoff between increased expressive power and increased computational complexity of various constructs. Remarkably enough, in moving from static choice to lazy dynamic choice, and then to eager dynamic choice, we were instead faced with an unusual "win/win" situation, where greater expressive power was achieved along with improved efficiency of execution: a perfect combination, which rarely occurs.

The results obtained in this paper, however, have a significance that exceeds the purely theoretical domain. In fact, experience in writing applications in $\mathcal{LDL}$++ illustrates that such a construct can be a powerful programming tool, to express, e.g., user-defined set-aggregates or depth-first tree traversals. A broad compendium of the programming styles supported by the non-deterministic logic database languages can be found in [15], along with examples from many application domains.

Acknowledgments

Thanks are due to Luca Corciulo, who helped us in clarifying the ideas expressed in this paper. The authors would also like to thank the referees for several suggested improvements.

References

[1] S. Abiteboul, E. Simon, V. Vianu. Non-Deterministic Language to Express Deterministic Transformation. In *Proceedings of ACM Symposium on Principles of Database Systems*, 1990. pp. 218-229.

[2] S. Abiteboul, V. Vianu. Procedural Languages for Database Queries and Updates. *Journal of Computer and System Science*, 41 (2) (1990).

[3] S. Abiteboul, V. Vianu. Fixpoint Extension of First Order Logic and Datalog-Like Languages. In *Proc. 4th Symp on Logic in Computer Science* (LICS). IEEE Computer Press (1989). pp. 71-89.

[4] S. Abiteboul, V. Vianu. Non-Determinism in Logic Based Languages. *Annals of Mathematics and Artificial Intelligence*, 3 (1991). pp. 151-186.

[5] S. Abiteboul, R. Hull, V. Vianu. *Foundations of Databases.* Addison-Wesley, 1995.

[6] K.R. Apt, D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1) (1993). pp. 109-157.

[7] A. Chandra, D. Harel. Structures and Complexity of Relational Queries. *Journal of Computer and System Science*, 25(1) (1982). pp. 99-128.

[8] A. Chandra, D. Harel. Horn clause queries and generalizations. *Journal of Logic Programming*, 1, (1985). pp. 1-15.

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

[10] E.F. Codd. *Relational Completeness of Database Sublanguages.* Data Base Systems, (Ed. R. Rustin), Prentice-Hall, Englewood Cliffs, NJ (1972) pp. 33-64.

[11] L. Corciulo. *Non-determinism in Deductive Databases.* Laurea Thesis. Dipartimento di Informatica, Università di Pisa. 1993 (in Italian).

[12] L. Corciulo, F. Giannotti, D. Pedreschi. Datalog with Non-Deterministic Choice Computes *NDB-PTIME.* In: S. Ceri, T. Katsumi, and S. Tsur, eds., *Proc. of DOOD'93, Third Int. Conf. on Deductive and Object-oriented Databases*, Lecture Notes in Computer Science, Vol. 760 (Springer, Berlin, 1993) 49-65.

[13] H. Gallaire, J. Minker, J.M. Nicolas. Logic and Databases, a Deductive Approach. *ACM Computing Surveys*, 16(2) (1984), pp. 153-185.

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

[15] F. Giannotti, S. Greco, D. Saccà, C. Zaniolo. Programming with Non-determinism in Deductive Databases. In [28] (1997), pp. 97-126.

[16] F. Giannotti, D. Pedreschi, D. Saccà, C. Zaniolo. Non-Determinism in Deductive Databases. In Proc. Deductive and Object-oriented Databases, Second International Conference, DOOD'91, (Eds. C. Delobel, M. Kifer, Y. Masunaga), Springer-Verlag, LNCS 566, pp. 129-146, 1991.

[17] F. Giannotti, D. Pedreschi. Datalog with Non-Deterministic Choice Computes *NDB-PTIME. Journal of Logic Programming* 35 (1998), pp. 79-101.

[18] S. Greco, D. Saccá. "Possible is Certain" is desirable and can be expressive. In [28] (1997), pp.147-168.

[19] S. Greco, D. Saccá, C. Zaniolo. Datalog Queries with Stratified Negation and Choice: from P to D^P. In *Proc. of the Int. Conf. on Database Theory*, LNCS 893 (1995) pp.82-96.

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

[21] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68 (1986) 86-104.

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

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

[24] P.G. Kolaitis, M.Y. Vardi. On the expressive power of Datalog: Tools and a case study. In *ACM Proc. Symp. on Principles of Database System* (1990) pp.61-71.

[25] P.G. Kolaitis. The Expressive Power of Stratified Logic Programs. *Information and Computation*, 90 (1991) pp.50-66.

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

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

[28] D. Pedreschi, V.S. Subrahmanian (editors). Advances in Logic-based Database Languages. *Annals of Mathematics and Artificial Intelligence*, Volume 19(1,2), (1997).

[29] D. Saccà, C. Zaniolo. Stable Models and Non-Determinism in Logic Programs with Negation. In *Proc. Symp. on Principles of Database System* PODS'90 (ACM Press, 1990) pp. 205-217.

[30] J.D. Ullman. *Principles of Databases and Knowledge Base System.* Volume I and II. Computer Science Press, Rockville, Md (1988, 1989).

[31] M. Vardi. The complexity of relational query languages. In *Proc. ACM-SIGACT Symp. on the Theory of Computing*, (1982) pp. 137-146.

[32] C. Zaniolo and H. Wang, Logic-Based User-Defined Aggregates for the Next Generation of Database Systems, in The Logic Programming Paradigm: Current Trends and Future Directions, K.R. Apt, V. Marek, M. Truszczynski, D.S.Warren (eds.), Springer Verlag, 1999.