Expressive Power of Non-Deterministic operators for logic-based languages

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

[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

[3] Computer Science Dept., UCLA, USA
e-mail: zaniolo@cs.ucla.edu

Abstract

Non-deterministic operators are needed in First-Order relational languages and Datalog to extend the expressive power of such languages and support efficient formulations of low-complexity problems. In this paper, we study the operators proposed in the literature, including witness, lazy choice and dynamic choice, and compare their power of expressing deterministic and non-deterministic queries. We obtain a simple hierarchy that relates these operators with each other and with other constructs, such as negation and fixpoint.

1 Introduction

Two main classes of logic-based languages have been extensively studied in the literature as the theoretical basis for relational database languages and their extensions. One is the class of first-order *FO* languages, which are based on relational algebra, or more precisely, the the first-order logic interpretation of 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. A first area of investigation has been the study of extensions that support the formulation of transitive closures and recursive queries that cannot be expressed in relational algebra. This has led to the definition of queries expressible in *FO* plus a fixpoint operator (fixpoint queries), Datalog with inflationary negation, and related concept that will be discussed later in this paper.

More recently the work of Abiteboul and Vianu [2, 3, 4, 5] has brought into focus the the need for having non-deterministic operators in such languages in addition to recursion and fixpoint. Therefore, they proposed the a non-deterministic construct called the *witness* [2, 3, 4, 5] for the fixpoint extensions of *FO*. They also proposed non-deterministic operational semantics for Datalog¬ (*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.

A *non-deterministic* query is one where there exist more than one acceptable answer. A very important 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 where there is only one correct answer. Non-deterministic operators are also essential for deterministic queries in as much as they are critical for expressing low complexity problems, such as determining the parity or the cardinality (or other set-aggregation functions) of a given relation. In this paper, we study the expressive power of languages for both deterministic queries and non-deterministic queries. This need has led to the introduction of the *choice* construct in $\mathcal{LDL}$. The original proposal by Krishnamurthy and Naqvi [19] was later revised by Saccà and Zaniolo [21] and refined in Giannotti, Pedreschi, Saccà and Zaniolo [14]. While the declarative semantics of *choice models* is based on and *stable models* semantics, it leads to efficient implementations, and it is actually supported in logic database language $\mathcal{LDL}$ and $\mathcal{LDL}$++ [20, 8].

The objective of this paper is to provide a characterization of the expressive power of various forms of non-deterministic constructs. Thus, in addition to FO+W(itness), we study the following three languages:

- Datalog with *static choice*, i.e. the choice construct in [19],
- Datalog with *lazy dynamic choice*, i.e. the choice construct in [21],
- Datalog with *eager dynamic choice*, i.e. the choice construct in [14].

We show that these languages establish to a hierarchy of increasing expressiveness which we correlate to the non-deterministic extensions FO. First, we prove that Datalog with static choice has the same expressiveness as the non-deterministic fixpoint extension of PEC, where PEC denotes the positive fragment of FO. Second, we recall a result from [11] 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.

2 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, 22, 9, 12]. The basic semantics of Datalog, consists in 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 upward powers of T_P are defined as follows:

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

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

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

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

For programs without negation, $T_P \uparrow (\emptyset) = T_P \uparrow \omega(\emptyset) = M_P$. This equality no longer holds for the language Datalog¬ which allows the the use of negation in the bodies of rules. Therefore, alternative semantics have been proposed; in particular, the *inflationary semantics* for *Datalog¬* which adopts $T'_P \uparrow \omega(\emptyset)$ as the meaning of a program P.

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. 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 there various exist equivalent definitions in the literature [6, 15].

Close connections exist between the fixpoint FO extensions and the Datalog extensions [5]: 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*. Moreover, Datalog is equivalent to *PEC+IFP*, where PEC, the positive existential calculus, denotes the negation-free, existential fragment of FO [7].

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*. A question 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 is achieved by introducing the so-called *witness* operator [2, 4, 5]. 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 3.1 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 nondeterministic query of assigning exactly one advisor to each student. □

From the viewpoint of the expressive power, the relevance of *FO+IFP+W* is due to the following result of Abiteboul and Vianu [4]:

Theorem 3.2 *A query is in NDB-PTIME iff it is expressed in FO+IFP+W.* □

4 The family of choice operators

The choice construct captures non-determinism while preserving the model-theoretic semantics of Datalog. Chocie is amenable to efficient implementation, has been adopted in the logic database language $\mathcal{LDL}$ and $\mathcal{LDL}$++ [20, 8] and it has been used widely in applications developed in these languages. The construct was introduced by Krishnamurthy and Naqvi [19], and later refined by Saccà and Zaniolo [21] and Giannotti, Pedreschi, Saccà and Zaniolo [14]; these improvements were motivated by the realization that different versions of the construct behave very differently in terms of expressive power.

4.1 Static choice

The choice construct was first proposed by Krishnamurthy and Naqvi in [19]. 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 4.1 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 $X \rightarrow Y$. Therefore, 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 becomes ineffective with recursive rules.

4.2 Dynamic Choice

An alternative approach to define a declarative semantics for the choice construct was proposed by Saccà and Zaniolo [21]. 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 [21], 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] 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 [13].

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 FDs 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 [21].

Example 4.2 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 has four distinct stable models, corresponding to the four choice models of Example 4.1. □

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 [19]. Indeed, the new formulation correctly supports the use of choice within recursive rules, avoiding the semantical anomalies of the static choice [14].

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

Definition 4.3 The choice program $ORD[U]$ consists of the following rules:

$$R_1 : \ SUCC(min, min).$$
$$R_2 : \ SUCC(X, Y) \leftarrow SUCC(\_, X), \ U(Y), \ choice((X),(Y)), choice((Y),(X)).$$

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

According to the semantics of the dynamic choice, the binary relation $SUCC$ defines a total, strict ordering over the input relation U. Viewing 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 add an (min, a), with a an element in U to $SUCC$, the second step of the fixpoint computation. No other arc can be added at this step, because it would violate the constraints imposed by choice. Likewise, at the third step a new elment must be taken from U will become the unique successor of a, and so on. Since the two choice goals enforces 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):

$$X < Z \leftarrow SUCC(X, Z),\ U(X),\ U(Z).$$
$$X < Z \leftarrow X < Y,\ SUCC(Y, Z),\ U(Z).$$

From this, the definition of inequality follows:

$$X \neq Y \leftarrow X < Y.$$
$$X \neq Y \leftarrow Y < X.$$

An operational semantics for choice is defined by the *dynamic choice fixpoint* (DCF) algorithm [14]. 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 operates as follows:

1. find the fixpoint of the **O**-rules;
2. if there exists an enabled ground instance r of a *chosen* rule in **C** then:
   (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 single instance of the *chosen* rule is executed together with all the associated $diffChoice$ rules. It can be shown that this procedure is sound and (non-deterministically) complete w.r.t. the computation of choice models.

4.3 Lazy and Eager Choice

In the DCF procedure described so far, the chosen rules are fired lazily, i.e., only after rules have been fired and the computation terminates unless a choice rule is fired. In the eager version of the DCF algorithm proposed in [11] all choice rules that are compatible with the functional dependency are executed before the procedure switches to the saturation mode again. Thus, the eager DCF algorithm is as follows:

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

This version of choice semantics, which will be called *eager dynamic choice* is somewhat simpler to implement and, more expressive than its *lazy dynamic choice* counterpart discussed previously. The following example from [14] shows how to 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. We assume here that both P and U are extensional relations.

Definition 4.4 The choice program $NOT[P, U]$ consists of the following rules:

$$R_1: \ NOT\_P(X) \leftarrow COMP\_P(X, 1).$$

$$R_2: \ COMP\_P(X, I) \leftarrow TAG\_P(X, I), choice((X), (I)).$$

$$R_3: \ TAG\_P(nil, 0).$$
$$R_4: \ TAG\_P(X, 0) \leftarrow P(X).$$
$$R_5: \ TAG\_P(X, 1) \leftarrow U(X), COMP\_P(\_, 0).$$

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

According to the specified operational semantics of eager 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 $TAG\_P(nil, 0)$ 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.

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 [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. Thus, eager choice can establish non-monotonic deterministic mappings from the database to the answer. Deterministic mappings specified using Datalog programs with choice under the lazy semantics are instead monotonic.

5 Expressiveness characterizations

In this section, we summarize the characterization the expressiveness of the various non-deterministic operators with respect to the procedural extensions of the relational calculus FO and the positive existential calculus PEC. In particular, we refer to $PEC + IFP + W$ and $FO + IFP + W$, where the latter is already known to express exactly *NDB-PTIME*. On the other hand, we proved new results concerning $PEC + IFP + W$, which are reported next.

5.1 Positive Existential Calculus and Witness

The witness construct, when added to languages which support negation, strictly augments expressiveness, in the sense that it is possible to compute new *deterministic* queries. In the case of the fixpoint logic $FO + IFP$, adding W allows to compute all queries in *NDB-PTIME*. Also the addition of W to FO allows the language to compute several deterministic queries not computable otherwise. It is therefore interesting to ask whether similar properties hold for PEC—the positive fragment of FO where negation and inequality are not included.

The next results show that the answer is negative: W without negation does not improve expressiveness on deterministic queries (with or without fixpoint).

Theorem 5.1 *A deterministic query is expressed in PEC iff it is expressed in $PEC + W$.* □

The following result is based on the observation that the witness does not enforce the FD constraint globally to the fixpoint procedure, and therefore it has no effect in absence of negation.

Theorem 5.2 *A deterministic query is expressed in $PEC + IFP$ iff it is expressed in $PEC + IFP + W$.* □

Moreover, it is known that $PEC + IFP$ is equivalent to Datalog [7], [10].

5.2 Datalog with Static choice

As a consequence of the Theorems 5.1, 5.2, we have that $PEC + IFP + W$ is equivalent to the language of the queries $W_x.\Phi$, where Φ is a query in $PEC + IFP$. In other words, first the fixpoint query Φ is computed, and then the non deterministic choices are operated. This directly implies equivalence with Datalog with static choice.

Theorem 5.3 *A query is expressed in Datalog with static choice iff it is expressed in $PEC + IFP + W$.* □

As shown by Definition 3.3, Datalog with dynamic choice can produce a total ordering on the universe, and, therefore, it can also express query $\neq$. Now, if the total-ordering query could be expressed in Datalog with static choice, then we have to conclude that the query $\neq$ can be expressed in $PEC + IFP + W$—and, by Theorem 4.2, in $PEC + IFP$, since $\neq$ is a deterministic query. But given the equivalence of Datalog with $PEC + IFP$ this would contradict a well known result [18]. Therefore:

Theorem 5.4 *The ordering query is inexpressible in Datalog with static choice.* □

5.3 Datalog with Dynamic choice

The next result, which characterizes the expressiveness of eager dynamic choice, has been proven in [11].

Theorem 5.5 *A query is expressed in $FO+IFP+W$ iff it is expressed in Datalog with dynamic choice.* □

As a consequence, we obtain that Datalog with eager choice represents a very powerful language, inasmuch as it can express all algorithms in *NDB-PTIME*s. Datalog with lazy dynamic choice is characterized by the DCF algorithm, which computes which is inflationary in nature (atoms are added and never deleted). Thus, the DCF computation can be easily simulated using FO+IFIP+W. Thus, by Theorem 5.5, we obtain that lazy dynamic choice is at most as expressive as dynamic choice.

Theorem 5.6 *If a query is expressed in Datalog with lazy dynamic choice then it is expressed in Datalog with dynamic choice.* □

Datalog with lazy dynamic choice has a monotonic semantics, in the sense that the transformation associated with a program with lazy dynamic choice yields a larger output when applied to a larger input database. As a consequence of this fact, the negation query cannot be computed. Therefore, eager dynamic choice is strictly more expressive than lazy dynamic choice.

Theorem 5.7 *The negation query is inexpressible in Datalog with lazy dynamic choice.* □

For non-recursive programs, static choice and dynamic choice have identical semantics. Moreover, the very definition of static choice shows that every program with static choice in recursive programs can be reduced to an equivalent program where choice is only used in non-recursive rules. Therefore:

Theorem 5.8 *If a query is expressed in Datalog with static choice then it is expressed in Datalog with lazy dynamic choice.* □

Theorems 5.8 and 5.4 allow to conclude that Datalog with dynamic choice strictly includes Datalog with static choice.

5.4 Conclusions

The following table summarizes the results presented in this paper, which characterizes the various choice operators of $\mathcal{LDL}$ as a hierarchy of increasing expressiveness.

Datalog with static choice	=	$PEC + IFP + W$
$\subset$		$\subset$
Datalog with lazy dynamic choice	=	...
$\subset$		$\subset$
Datalog with eager dynamic choice	=	$FO + IFP + W$

The expressive power of non-deterministic operators for FO languages has also been characterized in this paper. Given that $PEC + IFIP$ is equivalent to Datalog, we have shown that $PEC + IFIP + W$ is equivalent to the Datalog + Static Choice, which, in turn, is less powerful than Datalog with dynamic choice.

References

[1] 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.

[2] S. Abiteboul, V. Vianu. *Transaction Languages for Databases Update and Specification*. INRIA Technical Report n. 715 (1987).

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

[4] 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.

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

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

[7] A. Chandra, D. Harel. *Horn clause queries and generalizations*. Journal of Logic Programming 1, (19852). pp. 1-15.

[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] L. Corciulo, F. Giannotti, D. Pedreschi. *Datalog with Non-deterministic Choice Computes NDB-PTIME*. Proc. Deductive and Object-oriented Databases, Third International Conference, DOOD'93, 1993.

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

[13] 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.

[14] F. Giannotti, D. Pedreschi, D. Saccà, C. Zaniolo. *Non-Determinism in Deductive Databases*. 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.

[15] Y. Gurevich, S. Shelah. *Fixed-Point Extensions of First-Order Logic*. Annals of Pure and Applied 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] P.G. Kolaitis, M.Y. Vardi. *On the expressive power of Datalog: Tools and a case study.*ACM Proc. Symp. on Principles of Database System (1990) pp.61-71

[19] 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.

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

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

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