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 of First-Order relational languages and Datalog are needed to enhance the expressive power of such languages, and support efficient formulations of low-complexity problems. In this paper, we study semantics and expressiveness of the various non deterministic constructs proposed in the past, including various versions of the *choice* operator and the *witness* operator. We begin by formalizing declarative and fixpoint semantics, and operational semantics of these constructs, and analyze their power of expressing deterministic and non-deterministic queries. The paper presents various soundness and completeness results and an expressiveness hierarchy that relates the various operators with each other and other constructs, such as negation and fixpoint.

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

1 Introduction: Da Rivedere

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 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 concepts 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 non-deterministic construct called the *witness* [2, 3, 4, 5] for the fixpoint extensions of *FO*. They also proposed non-deterministic procedural 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 which admits 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.

Clearly, a *deterministic* query is one which admits 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.

The quest for enhanced expressiveness has led to the introduction of the *choice* construct in the logic database language $\mathcal{LDL}$. The basic idea is to extend the procedural (bottom-up) semantics of deductive databases in such a way that a subset of the answer to a query is chosen, on the basis of a functional dependency constraint. However, several different interpretations of the choice construct are possible. The original proposal by Krishnamurthy and Naqvi [20] was later revised by Saccà and Zaniolo [22], and refined in Giannotti, Pedreschi, Saccà and Zaniolo [15]. While the declarative semantics of *choice models* is based on *stable model* semantics, it leads to efficient implementations, and it is actually supported in the logic database language $\mathcal{LDL}$ and $\mathcal{LDL}$++ [21, 9].

So far, no comprehensive study is available which compares the various choice operators with each other and with other construct, such as the witness, from the point of view of semantics and expressiveness.

In this paper, the various *choice* operators proposed in the literature are systematically studied, with the aim of:

- clarifying the relationships among their procedural, declarative and fixpoint semantics, and
- comparing their relative power of expressing deterministic and non-deterministic queries, also with respect to the *witness* operator.

More precisely, in addition to *FO*+W(itness), we consider the following three languages:

- Datalog with *static choice*, i.e. the choice construct in [20],
- Datalog with *lazy dynamic choice*, i.e. the choice construct in [22],
- Datalog with *eager dynamic choice*, i.e. the choice construct in [15].

On the one hand, we provide a thorough formalization of non deterministic bottom-up evaluation of dynamic choice programs, by introducing an immediate consequence operator which

allows us to relate procedural, declarative and fixpoint-based semantics. In particular, we obtain soundness and completeness of the procedural semantics with respect to the intended declarative semantics, based on stable models. Also, we show that the proposed procedural semantics can be justified in terms of (minimal) fixpoints. The specific properties of lazy and eager dynamic choice are also studied, such as the ability of computing non monotonic queries.

On the other hand, we provide a thorough characterization of the expressive power of various forms of non-deterministic constructs. We establish 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 [12] 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.

The plan of the paper 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 various forms of non deterministic *choice* operators for Datalog and the associated programming styles. In Section 5 the results concerning the semantics characterizations of the choice operators are presented, whereas in Section 6 the results concerning the expressiveness characterizations of the choice and witness operators are presented.

2 Background

We assume that the reader is familiar with the relational model and associated algebra, the relational calculus (i.e. the *first-order queries*, denote Datalog [18, 23, 10, 13]. The basic semantics of Datalog, consists in evaluating l applicable instantiations of the rules. This is formalized using the cons r T_P associated to a Datalog program P, which is a map over (Herbrand) ined as follows:

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

The upward powers of T_P starting from an inte s follows:

$$\begin{aligned} T_P \uparrow 0(I) &= I \\ T_P \uparrow (i+1)(I) &= T_P(\\ T_P \uparrow \omega(I) &= \bigcup \end{aligned}$$

The least model M_P of a positive progra ch 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$\neg$ 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$\neg$ which adopts $T'_P \uparrow \omega$ as the meaning of a program P. In this paper, we use *stable model* semantics of Datalo$\neg$ 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$\neg$ program P into a positive Datalog program $ground_I(P)$ obtained from $ground(P)$ 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$\neg$ program P iff M is the least model of the program $ground_M(P)$. In general, Datalog$\neg$ 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, whose instance is the limit of the sequence $J_0, \ldots, J_k, \ldots$, defined as follows (given a database 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.

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 there exist various equivalent definitions in the literature [7, 16].

Close connections exist between the fixpoint FO extensions and the Datalog extensions [5]: Datalog$\neg$ under inflationary fixpoint semantics expresses exactly the fixpoint queries, i.e. it is equivalent to *FO+IFP*. This implies that Datalog$\neg$ 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* [8].

Finally, the complexity measures are functions of the size of the input database. For Turing Machine complexity class C there is a corresponding complexity class of (non-deterministic) queries *(N)DB-C*. In particular, the class of (non-deterministic) database queries that can be computed by a (non-deterministic) Turing Machine in polynomial time is denoted by *(N)DB-PTIME*. It is 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. 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* 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 X and Y are disjoint vectors of variables representing a partition of the free variables of the formula 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 non-deterministic 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.* □

It should be remarked that the interpretation of the W operator and that of the fixpoint *IFP* operator are given orthogonally. As a consequence, when W is used within an *IFP* formula, the functional dependency constraint is only enforced at each stage of the fixpoint computation, but need not to hold in the final outcome of the query. This observation marks the main difference between W and the different forms of the choice construct, discussed in the next section.

4 The Choice Operators

In this section we provide an informal description of the various forms of choice operators, together with some example of the style of programming supported by Datalog augmented with choice. The choice constructs were designed for supporting non-determinism while

preserving the model-theoretic semantics of Datalog. Another design principle underlying the choice constructs is that they were supposed to be amenable to efficient implementation. In fact, all the forms of choices described in this paper have been adopted in the various versions of the logic database languages $\mathcal{LDL}$ and $\mathcal{LDL}$++ [21, 9] and have been widely used in applications developed in these languages. The construct was introduced in [20], and later refined in [22] and [15]; these improvements were motivated by the realization that the different versions of the construct are quite different from the point of view of the expressive power.

4.1 Static Choice

The choice construct was first proposed by Krishnamurthy and Naqvi in [20]. 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 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 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 $X \rightarrow Y$. Under these hypotheses, 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 for all, after a preliminary fixpoint computation. Because of its static nature, this form of choice becomes ineffective with recursive rules.

4.2 Dynamic Choice

A new semantics for the choice operator was was proposed by Saccà and Zaniolo [22]. The new formulation of choice called *Dynamic Choice* avoids the semantical anomalies of the version proposed [20]. As discussed in details later, anonomalies occur when choice is used within recursive rules [15].

Furthermore, 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 [22], 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 4.2 The *stable version* $SV(P)$ of a choice program P is obtained by the following transformation. For any choice rule r of P

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

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:

1. replace r with a rule r' obtained by replacing the body of r with the atom $chosen_r(Z)$:

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

2. add the new rule:

$$chosen_r(Z) \leftarrow B(Z), \neg diffChoice_r(Z).$$

3. add the new rule:

$$diffChoice_r(Z) \leftarrow chosen_r(Z'), Y \neq Y'.$$

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

An interpretation I is called a *stable choice model* of P iff I is a stable model of $SV(P)$. □

The transformation directly generalizes to FDs involving vectors of variables, and to rules with 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 (in the sense of the static choice) coincides with the set of stable choice models of P on common predicate symbols [22].

Example 4.3 The following is the stable version of Example 3.

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

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.

An operational semantics for this form of choice is defined by the *dynamic choice fixpoint* (DCF) algorithm [15]. 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.

The qualification *dynamic* for this form of choice stems from the fact that the choice are interleaved during the fixpoint computation, and not postponed till the end as in the static choice.

5 Static Choice versus Dynamic Choice: Lazy and Eager

Let us now discuss the issues that have motivated the introduction of dynamic choice and its further refinements into a lazy and eager versions. These are issues of efficiency and expressive power that have emerged while implementing and using choice as a non-deterministic pruning operator in the $\mathcal{LDL}$++ system.

An interesting example consitsts on studying how choice can be used to modify and prune the transitive closure and graph reachability programs. For instance, the following program that generates a spanning tree, rooted in a node a for a graph, where an arc from X to Y with cost C is represented by facts g(X, Y, C):

Example 5.1 Spanning tree

```
st(nil, a).
st(X, Y)  ←  st(_, X),  g(X, Y), choice((Y), (X)).
```

(This program assumes that there is no loop arc g(a, a), otherwise a condition $X \neq Y$ must be added.)

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. Consider now the following graph:

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

If we consider the equivalent program under dynamic choice it is easy to see that the program 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 version of choice (which we will call static to distinguish to new one called dynamic), the semantics call for (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 satisfies the FDs of the choice goals.

Therefore, in addition of the above three models the static choice will also accect the solution:

$$\texttt{st(nil, a), st(b, c), st(c, b)}$$

This solution not longer corresponds to a tree but rather to an unsupported cycle 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 pruning and controlling the computation of a recursive predicate. This ability is lost static choice, which, because of its semantics, 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.

Example 5.2 Spanning tree

```
st(nil, 0).
st(Y, s(J))  ←  st(X, J),  g(X, Y), choice((X), (J)).
```

On the graph $g(a, b), g(b, b)$ this program has an infinite number of static-choice models, each requiring an infinite computation. Because of the pruning, the program has only one model, computed in two steps. □

In terms of improved expressive power, it is easy to prove that for each program with the static choice construct there is an equivalent dynamic choice program (hint: move the choice construct after recursion), the opposite is not true. For instance the inequality construct can be expressed by a positive datalog program and dynamic choice, but cannot be expresseda by a positive datalog program with static choice. (See section).

The typical situation that a designer has to face is a tradeoff between increased expressive power and increased computational complexity of various constructs. In moving from static to dynamic choice we were instead faced with an unusual win/win situation, where greater expressive power was achieved along with improved efficiency.

Furtermore this unusual situation of "inverted tradeoff curve" occured again in the transition to eager dynamic choice on which most of the paper is spent. In a nutshell though the situation whether once the node is reached all the available choices compatible with the FD constraints should be exercised in a

.....Perhaps move this...

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

$$R_1 : \ SUCC(min, min).$$

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

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 $SUCC$ 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 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:

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

This is a first example of a deterministic query which can be expressed using the form of non determinism provided by the dynamic choice.

6 Semantics Characterizations

After the informal presentation of the various forms of choice constructs, this section is aimed at providing a formalization of dynamic choice. The main tool is a notion of *non deterministic immediate consequence operator*, which yields directly the notions of a computation and a fixpoint. We define a general operator Ψ_P for non deterministic bottom-up computations of a choice program P, and study its properties from three different points of view: procedural, fixpoint and declarative semantics. Characterization results which relate the three semantics are established. The lazy and eager dynamic choice semantics are next identified as instances of the general operator Ψ_P, in such a way that their relevant properties are directly derived.

For the sake of generality, we do not confine ourselves to the case of Datalog choice programs, and study arbitrary choice programs, which admit the use of function symbols. The results hold in this general sense, although it is needed to introduce a notion of fairness with respect to choices, in order to prevent that infinite computations do not converge to a fixpoint.

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 4.2. 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 6.1 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:

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

where either $\alpha = n$ (> 0) or $\alpha = \omega$, and

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

□

Informally, the Ψ_P operator, when applied to an interpretation I, first uses T_{P_C} to derive all the admissible choices, and then uses T_{P_H} on all possible admissible subsets of the derived choices, in order to derive all consequences of the operated choices. Clearly, non determinism is due to fact that there are many subsets of admissible choices. 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 definition of Ψ_P is parametric with respect an ordinal α, which is either finite or ω. The selection of α gives rise to a family of operators. Again, the need for introducing the ordinal parameter is due to the general case of arbitrary programs with function symbols. If $\alpha = \omega$ is chosen in the definition of Ψ_P, then the computation step from I_n to I_{n+1} is not effectively computable in the general case of logic programs, although effectiveness is still guaranteed for Datalog programs, since saturation via T_{P_H} is obtained at some finite stage.

In the definition of $\Gamma_P(I)$, two cases are specified, in such a way that ΔI is not empty if $T_{P_C}(I) \setminus I$ is not empty, thus reflecting the fact that, if there are possible new choices, then at least one has to be made.

The Ψ_P operator formalizes a single step of an ideal bottom-up 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 6.2 Given a choice program P, a *(non deterministic) computation* of 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 $n \geq 0$.

A computation $\langle I_n \rangle_{n \geq 0}$ is *fair* iff for any *chosen* atom H such that $H \in T_{P_C}(I_n)$ for infinitely many $n \geq 0$, then $H \in I_m$ for some $m \geq 0$. □

According to the above definition, computations are always infinite. In our framework, finite computations are viewed as infinite computations which reach a fixpoint at a finite stage.

The fairness requirement characterizes those computations which eventually make all admissible choices. Conversely, an unfair computation is one which never selects a choice which is infinitely often compatible with the FD's.

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

Lemma 6.3 *Let $\langle I_n \rangle$ $(n \geq 0)$ be a non deterministic computation of 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, for any $y' \neq y$, $chosen_r(x, y') \notin I_n$. 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 $lim_{n \geq 0} \langle I_n \rangle = \bigcup_{n < \omega} I_n$.

We now aim at relating limits of fair computations and fixpoints of Ψ_P, in order to justify the fact that non deterministic (fair) computations are in fact fixpoint computations. We need here a unconventional notion of fixpoint, due to the fact that Ψ_P maps interpretations to sets of interpretations.

Definition 6.4 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 6.5
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:

$$T_{SV(P)}(I) = I \quad \text{iff} \quad T_{P_C}(I) = I_C \text{ and } T_{P_H}(I) = I_H, \tag{1}$$

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 $T_{SV(P)}$. (1) implies that $T_{P_C}(I) \setminus I = \emptyset$, and therefore $\Gamma_P(I) = \{I\}$. Also, (1) 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 (1). □

The next result shows that the limits of fair computations are *minimal* fixpoints of Ψ_P.

Theorem 6.6 *Let M be the limit of a fair computation $\langle I_n \rangle$ $(n \geq 0)$. Then M is a minimal fixpoint of Ψ_P.*

Proof. We first show that M is a fixpoint of Ψ_P. Assume by contradiction that $\Psi_P(M) \neq \{M\}$. This, together with Theorem 6.3(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

$$M \models B \tag{2}$$
$$M \not\models H \tag{3}$$
$$J \models H. \tag{4}$$

By (2) we have that for some $k > 0$, $I_k \models B$, i.e., Two cases then arise.

Case 1. r is not a *chosen* clause.
Then we conclude that $I_{k+1} \models H$ by the definition of Ψ_P, and therefore we get a contradiction of (3), as $I_{k+1} \subseteq M$ by the fact M is the limit of the computation.

Case 2. r is a *chosen* clause.
Then, for any $n \geq k$, $I_n \cup \{H\} \models FD_P$, otherwise (4) is contradicted. Therefore, by the fairness requirement, $I_m \models H$ for some $m \geq k$, which contradicts (3).

This concludes the proof that M is a fixpoint of Ψ_P. We now show that M is a minimal fixpoint. Assume by contradiction that there exists a fixpoint M' of Ψ_P such that $M' \subset M$, and consider an atom $H \in M \setminus M'$. 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 clause $H \leftarrow B$ of $SV(P)$ such that $I_{n-1} \models B$. Two cases arise.

Case 1. H is not a *chosen* atom.
As a consequence of the fact that M' is a fixpoint and $M' \not\models H$, we have that, for some atom A of B, $A \in M \setminus M'$.

Case 2. H is a *chosen* atom.
Let $H = chosen_r(t)$. Hence the body B of the clause is $C, \neg diffChoice_r(t)$. As a consequence of Theorem 6.3(1) and (2), $M \models FD_P$, which implies that $M \not\models diffChoice_r(t)$. By the fact that $M' \subseteq M$ we have that also $M' \not\models diffChoice_r(t)$. As a consequence of the fact that M' is a fixpoint and $M' \not\models chosen_r(t)$, we have that, for some atom A of C, $A \in M \setminus M'$.

In both cases, we found that the existence of an atom in $I_n \setminus M'$ implies the existence of another atom in $I_{n-1} \setminus M'$. We can therefore repeat the construction of cases 1 and 2 $n-1$ times, until we obtain the existence of an atom in $I_0 \setminus M' = \emptyset$, which is a contradiction. □

The next result shows that non deterministic computations are stable choice models, thus providing a notion of soundness w.r.t. the stable model semantics of choice programs.

Theorem 6.7 (Soundness)
Any limit of a non deterministic fair computation of a choice program P is a stable choice model of P.

Proof. Let M 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'. Clearly, by Theorem 6.6, M is a model of P'. To show that M is the least model of P' we provide a construction similar to that of the proof of Theorem 6.6. Assume by contradiction that 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 6.3(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 individuating an atom A, inferred by a unit clause $A \leftarrow$ from $SV(P)$, such that $J \not\models A$. This contradicts the fact that J is a model of P'. □

Theorem 6.8 (Completeness)
Let P be a choice program, and assume $\alpha = \omega$ in the definition of Ψ_P. Then any stable choice model of P is the limit of a fair computation of P.

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

$$M \models FD_P. \tag{5}$$

To prove (5), 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. $M_{n+1} = T_{P'_H} \uparrow \omega(T_{P'_C}(M_n) \cup M_n), \quad$ 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 (5), 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 fair computation of P. □

It is easy to see that, for α finite, completeness does not hold. Therefore, a transfinite computation is required in the evaluation of a single step of Ψ_P in order to compute any stable model. However, this limitation only applies to arbitrary choice programs with function symbols, and not to Datalog programs, as in the latter case the saturation of T_{P_H} can be effectively computed.

Summarizing, we have shown that stable choice models and limits of fair computations coincide, and that they are minimal fixpoints. Therefore, we have that two semantics of choice programs coincide, namely the procedural and declarative interpretation. Such a coincidence does not hold with the fixpoint semantics, in that there are programs with minimal fixpoints which are not stable choice model or equivalently limits of computations. As an example, consider the following program P:

$$p(a) \leftarrow p(a).$$
$$p(b).$$
$$q(X) \leftarrow p(X), choice((), (X)).$$

and its stable version $SV(P)$:

$$p(a) \leftarrow p(a).$$
$$p(b).$$
$$q(X) \leftarrow p(X), chosen(X).$$
$$chosen(X) \leftarrow p(X), \neg diffChoice(X).$$
$$diffChoice(X) \leftarrow chosen(X'), X \neq X'.$$

It is readily checked that Ψ_P has two minimal fixpoints:

$$M_1 = \{p(b), q(b), chosen(b), diffChoice(a)\}$$
$$M_2 = \{p(b), p(a), q(a), chosen(a), diffChoice(b)\}$$

but M_2 is not a stable choice model of P, as well as it 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 procedural semantics and fixpoint based semantics supports the choice of stable model semantics for choice programs. In fact, the only other semantics which allows multiple models is the completion semantics, which coincides with the fixpoint semantics discussed in this section. The simple counter example exhibited above shows that completion semantics allows more models than those computable with the intended procedural semantics, and therefore it is not adequate.

As a second remark, it is natural to ask ourselves whether reasonably large classes of choice programs exist for which stable model and 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. The interest about this possibility is that we would obtain simpler means for reasoning about choice programs in a declarative way. However, this forms material for further research.

6.1 Eager Dynamic Choice

We are now in the position of introducing two natural instances of the Ψ_P operator which formally describe two distinct choice constructs. The first one corresponds to the dynamic choice construct discussed earlier, and the second one correspond to a new construct, called *eager* dynamic choice. In order to avoid confusion, the first construct is referred to as *lazy* dynamic choice. In both cases, the ordinal α is chosen to be ω, whereas the choice of certain ΔI's in the definition of Γ_P is what makes the difference. In the case of lazy choice, only singleton subsets of the admissible choices are selected as ΔI, whereas in the case of eager choice, only maximal subsets of the admissible choices which satisfies the FD's are selected.

Definition 6.9
(1) The *lazy* operator Ψ_P^L is defined as the instance of Ψ_P where $\alpha = \omega$ and ΔI is a singleton:

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

where

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

(2) The *eager* operator Ψ_P^E is defined as the instance of Ψ_P where $\alpha = \omega$ and ΔI is maximal:

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

where

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

□

Computations according to the Ψ_P^L and the Ψ_P^E operators formalize respectively the lazy and eager dynamic choice procedures. The latter, proposed in [12], can be described as follows, by modifying that in Section 4.2 in such a way that a *maximal* set of choices compatible with the FD's is made whenever the algorithm is in the choice mode.

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 is somewhat simpler to implement and more expressive than lazy dynamic choice, as it will be shown later. The following example from [15] 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 6.10 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 sketched 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 any x in the extension of relation P. In the following choice phase the facts $chosen(x, 0)$ are chosen, again for any 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 any x in the extension of P, and the facts $TAG\_P(x, 1)$ for any 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 any x *not* in the extension of P, as any x in P has 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 [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 non-monotonic deterministic queries. On the other hand, we shall see later how any deterministic query specified using the lazy dynamic choice is indeed monotonic.

Some remarkable differences among the lazy and eager choice construct can be enlightened using their formalization in terms of the associated operators Ψ_P^L and Ψ_P^E (and their relation with the general operator Ψ_P).

Firstly, the Ψ_P operator, as well as the lazy operator Ψ_P^L obey a property of non deterministic monotonicity. The next result shows that computations starting from larger input databases yield larger output, in a sense made precise below.

Lemma 6.11 *Let P and P' choice programs such that P' is equal to P except that it has a larger EDB. Then for any limit I of a computation of P there is a limit J of a computation of P' such that $I \subseteq J$.*

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 $n \geq 0$:

$$I \cup (J_n \setminus I_n) \models FD_P, \tag{6}$$

$$I \cap (J_n \setminus I_n) = \emptyset. \tag{7}$$

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

$$\Delta I_n \subseteq T_{P'_C}(J_n) \setminus J_n. \tag{8}$$

Due to (7) 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 (7), as $chosen(t) \in \Delta I_n$ and, a fortiori, in I.

Next, as a consequence of (8) 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 (7) and (7), and put

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

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

A similar construction can be adopted in the case of the lazy operator Ψ_P^L. □

As a consequence of this Lemma, we have have that on *deterministic* queries, i.e., those queries for which exactly one computation exists, the general and the lazy operator behave monotonically in the standard sense. In other words, these operators allow to compute only monotonic deterministic queries. On the contrary, the eager operator Ψ_P^E 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 6.11 is no longer possible. An example of a non monotonic deterministic query computed by means of the eager operator is the negation/complement query in Section 4.3.

Another difference concerns fairness. Any eager computation is fair, as a consequence of the commitment to maximal sets of admissible choices. On the contrary, lazy computations are not necessarily fair, and therefore lazy computations must be monitored by appropriate schedulers to ensure that computations are not terminated until a complete stable choice model is reached.

Finally, it is readily checked that the procedural semantics based on the lazy operator is complete in the sense of Theorem 6.8, since any computation can be emulated by one where choices are made one at a time. On the other hand, the procedural semantics based on the eager operator is not complete. Again, an illuminating example is the negation query of Section 4.3. Stable choice models of such program exist where the relation NOT_P is not the set difference of relations P and U: these solutions correspond to computations where non maximal sets of choices were made at some steps.

7 Expressiveness Characterizations

In this section, we study the expressiveness of the various static and dynamic choice operators. We show how these operators give raise to a hierarchy of increasingly more expressive query languages. In order to achieve a complete characterization, we also relate this hierarchy with the query languages obtained by adding the witness operator to the relational calculus *FO* and its positive existential fragment *PEC*. Therefore, we refer to the query languages $PEC+IFP+W$ and $FO+IFP+W$. The latter is already known to express exactly *NDB-PTIME*. On the other hand, we needed a finer tuned characterization of $PEC+IFP+W$. ¿From our study, new results concerning $PEC+IFP+W$ emerged, which are reported next.

7.1 Positive Existential Calculus and Witness

The witness construct, when added to languages which support negation, strictly enhances expressiveness, in the sense that it makes 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*, hence all polynomial deterministic queries. Also, it has been recently shown that the addition of W to *FO* (without recursion) allows to compute certain deterministic queries not expressible in *FO*. It is therefore interesting to ask ourselves whether similar properties hold for *PEC*—the positive fragment of *FO* where negation and universal quantification are dropped.

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

Theorem 7.1 *A deterministic query is expressed in PEC iff it is expressed in $PEC + W$.*

Proof. Consider a deterministic query Φ in $PEC + W$, and let I be the (unique) database instance which satisfies Φ. Consider next the (deterministic) query Φ' in PEC, obtained from Φ by removing any occurrence of a witness operator, and let J be the database instance 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 $PEC + W$ is monotonic, there exists a maximal instance 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. □

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

Theorem 7.2 *A deterministic query is expressed in $PEC + IFP$ iff it is expressed in $PEC + IFP + W$.* □

As a conclusion, W, when added to negation-free deterministic query languages, allows to compute certain non deterministic queries, but does not improve on deterministic queries. Another, perhaps surprising, result is that $PEC + IFP + W$ is equivalent to its sub-language 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 7.3 *A query is expressed in $PEC + IFP + W$ iff it is expressed in L.*

Proof. It suffices to show that any interpretation of the formula $IFP(\Phi, S)$, where Φ is a formula in $PEC + W$ and S is a relation occurring in Φ, coincides with the (unique) interpretation of $IFP(\Phi', S)$, where Φ' is the PEC formula obtained by dropping all occurrences of W in Φ. To this end, we observe that the monotonicity of PEC 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's constraints of the witness operators are not maintained through the fixpoint computation, we have that eventually all candidate witnesses are selected. □

The above result is based on the observation that the witness does not enforce the FD constraint globally to the fixpoint computation, and therefore it has no effect in presence of monotonicity. In fact, all alternatives will be always present during the computation, and hence eventually selected, due to the semantics of witness.

7.2 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. This observation, together with the known fact that $PEC + IFP$ is equivalent to Datalog [8], [11], allows us to derive immediately an interesting corollary of Theorem 7.3, namely that Datalog with static choice and $PEC + IFP + W$ are equivalent query languages.

Theorem 7.4 *A query is expressed in Datalog with static choice iff it is expressed in* $PEC + IFP + W$. □

As shown by Definition 5.3, Datalog with dynamic choice can produce a total ordering on the universe, and, therefore, it can also express the inequality 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 7.2, 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 [19]. Therefore:

Theorem 7.5 *The ordering query is inexpressible in Datalog with static choice.* □

7.3 Datalog with Dynamic Choice

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

Theorem 7.6 *A query is expressed in* $FO + IFP + W$ *iff it is expressed in Datalog with eager dynamic choice.* □

As a consequence, we obtain that 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, which is an instance of the Ψ_P operator. As a consequence of Lemma 6.3(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 7.6, we obtain that any query expressible with lazy dynamic choice is also expressible using eager dynamic choice, as the latter express all (non deterministic) polynomial queries.

Theorem 7.7 *If a query is expressed in Datalog with lazy dynamic choice then it is expressed in Datalog with eager dynamic choice.* □

As a consequence of Lemma 6.11, 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. Therefore, eager dynamic choice is strictly more expressive than lazy dynamic choice.

Theorem 7.8 *The negation query is inexpressible in Datalog with lazy dynamic choice.* □

It is immediate to observe that, in the case of programs where all choice rules are non recursive, all three forms of choice constructs exhibit the same semantics. Moreover, by Theorems 7.3 and 7.4 we obtain that every program with static choice in recursive rules can be reduced to an equivalent program where choice is only used in non-recursive rules. Therefore:

Theorem 7.9 *If a query is expressed in Datalog with static choice then it is expressed in Datalog with lazy dynamic choice.* □

The following table summarizes the results presented in this section, which characterize the query languages associated with the various choice operators as a hierarchy of increasingly higher expressiveness.

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

It should be stressed that all extensions of Datalog considered here do not allow the use of negation—although negation can be defined using eager dynamic choice. Greco and Saccà [] showed that Datalog with stratified negation and *lazy* dynamic choice is yet another language which express exactly *NDB-PTIME*

8 Conclusions

Acknowledgements

Thanks are owing to Luca Corciulo, who helped us in clarifying the ideas expressed in this paper.

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] 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 (1982). pp. 99-128.

[8] A. Chandra, D. Harel. *Horn clause queries and generalizations.* Journal of Logic Programming 1, (19852). 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.* Proc. Deductive and Object-oriented Databases, Third International Conference, DOOD'93, 1993.

[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.* Proc. 5th Int. Conf. and Symp. on Logic Programming, MIT Press, pp. 1080-1070, 1988.

[15] 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.

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

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

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

[19] 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

[20] 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.

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

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

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