Baltzer Journals AMAI, Vol. 19, No. 3-4, 1997

Programming with Non-determinism in Deductive Databases *

Fosca Giannotti[1], Sergio Greco[2], Domenico Saccà[2] and Carlo Zaniolo[3]

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

[2] *DEIS, Univ. della Calabria, 87030 Rende, Italy*
E-mail: lastname@si.deis.unical.it

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

While non-determinism has long been established as a key concept in logic programming, its importance in the context of deductive databases was recognized only recently. This paper provides an overview of recent results on this topic with the aim of providing an introduction to the theory and practice of non-determinism in deductive databases. In particular we

(i) recall the main results linking non-deterministic constructs in database languages to the theory of data complexity and the expressibility hierarchy of query languages

(ii) provide a reasoned introduction to effective programming with non-deterministic constructs

(iii) compare the usage of non-deterministic constructs in languages such as $\mathcal{LDL}$++ to that of traditional logic programming languages

(iv) discuss the link between the semantics of logic programs with non-deterministic constructs and the stable-model semantics of logic programs with negation.

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

1 – Introduction

Much of recent database research is inspired by the need for languages more powerful than traditional query languages such as SQL or QUEL. In particular, Deductive Databases focus on the use of an efficient support for logic-based rule languages such as Datalog and its extensions. Extensions are needed since Dat-

*This work was supported in part by ECUS033 and NSF Grant IRI-9314905

alog, in its most basic form that supports recursion but no negation or function symbols, can only express a proper subset of monotonic polynomial-time queries [5].

A first step toward increasing the expressive power of Datalog consists of allowing limited forms of negation in the bodies of the rules. This leads to *stratified* Datalog, denoted $\text{Datalog}^{\neg s}$, where negation cannot occur in recursive definitions [6, 10, 42]. $\text{Datalog}^{\neg s}$ has an intuitive declarative semantics that is also conducive to efficient implementation. Unfortunately, the expressive power gained with stratified negation is limited: many polynomial-time queries cannot be expressed by $\text{Datalog}^{\neg s}$. For instance, the well-known simple query which asks whether a domain has even cardinality cannot be expressed in $\text{Datalog}^{\neg s}$ [4, 3].

Several approaches can be followed to increase the power of the language beyond $\text{Datalog}^{\neg s}$, including the three discussed next. A first approach consists in assuming that the underlying universe is totally ordered; since this assumption violates the data-independence preserving condition of *genericity* [3], this approach will not be further pursued in this paper. A second approach consists in allowing more general uses of negation by supporting more powerful semantics that allow non-stratified negation. Take, for instance the *stable model* semantics for negation [20]; this semantics captures both the non-monotonic and the non-deterministic aspects of Datalog programs, and its power is such that $\mathcal{NP}$-hard problems can be easily expressed using a few rules [37]. Unfortunately, stable model semantics comes with serious drawbacks, such as (i) lack of universality, whereby not every program has a stable model, and (ii) computational cost, whereby deciding whether a given program has some stable model is $\mathcal{NP}$-complete, and computing any such model is $\mathcal{NP}$-hard [38, 39]. Thus, while stable models can provide a desirable framework for an abstract semantics (and in fact, they will be used in this paper in such a role) they do not supply a good basis for concrete semantics.

The third approach consists in using non-deterministic constructs, such as the *choice* construct, along with stratified negation. In this paper, we concentrate on this third approach, and show that non-deterministic constructs provide a desirable venue, from both a semantic and computational viewpoint, to extending Datalog's power to $PTIME$ [3] and beyond. In this paper, we overview and present in a coherent framework previously reported results with the aim of making them more accessible to a wider audience.

From a pragmatic point of view, non-deterministic constructs in languages are typically used in two different ways. One is when a committed choice is made, never to be revised. The other is when the correct choice is not known, i.e., that which will satisfy a given objective—so the initial choice will be later revisited and revised until the desired objective is reached, or no alternative choice remains. The former notion (often called, "don't care") is related to the fact that some answer to a query can be deliberately discarded. This is accomplished by *pruning* *operators* in logic programming and by *choice operators* in deductive databases. The latter notion, named "don't know", is related to the fact that queries to a logic program may have many alternative answers, corresponding to alternative ways of proving the theorem represented by the query. In Prolog-like languages, this is accomplished by means of backtracking mechanisms, whereas in Datalog-like languages, "don't know" nondeterminism is usually concealed by the default "all solutions" behavior of the bottom-up evaluation.

In most of this paper, dealing with polynomial algorithms, we study the committed-choice/"don't care" interpretation. In the final part of the paper, dealing with exponential computations, we study the "don't know" interpretation.

The rest of the paper is organized as follows. After the preliminary definitions in the next section, Section 3 discusses non-deterministic constructs based on functional dependencies, and their limitations. Section 4, introduces the notion of dynamic choice which ensures the ability of expressing all polynomial-time queries. A comparative introduction to programming with non-determinism in different logic-based languages is given in Sections 5 and 6. In Section 7, we turn to the operational semantics of choice and to two specializations of such semantics, called respectively eager choice and greedy choice. These are two candidates of great practical interest, because they extend even further the expressive power and pruning ability of dynamic choice. In Section 8, we study the problem of using choice to express non-polynomial time algorithms. We therefore introduce two extensions to choice programs that subsume the power of unrestricted negation under stable model semantics and provide a better basis for programming; one such extension uses XY-stratified $\mathrm{Datalog}_{1S}$ programs, and the other uses quantified query goals.

2 – Preliminary Definitions

We assume the reader is familiar with the concepts of relational databases and of the Datalog language [26, 41] as well as of logic programming [31].

Given a *logic program* P, let U_P, B_P and $ground(P)$ denote respectively the Herbrand universe, the Herbrand Base and the ground instantiation of P. An *interpretation* for P is any subset of B_P and it is a *model* for P if it makes true all rules in $ground(P)$.

The semantics of a *positive* (i.e., negation-free) logic program P is given by the minimum model that coincides with $T_P^{\omega}(\emptyset)$; i.e., the power to the first ordinal of the *immediate transformation* T_P.

For a program with negation, we will assume in this paper, that its intended semantics is defined by the set of its total stable models [20]. We recall that, given a logic program P and an interpretation M of P, M is a (*total*) *stable model* of P if M is the minimal model of the program $gl(P,M)$ defined as follows: $gl(P,M)$ is obtained from the ground instance of P by (i) deleting all rules which have some negative literal $\neg b$ in their bodies with $b \in M$, and (ii) removing all negative literals in the bodies of the remaining rules.

Existence and unicity of stable models is not guaranteed. However, positive programs have a unique stable model that coincides with the minimum model. Another interesting class of programs for which a stable model always exists and is unique is *stratified* programs, briefly discussed next. Given a program P with predicate symbols p and q, we write $p \rightarrow q$ if there exists a rule where q occurs in the head and p in the body or there exists a predicate s such that $p \rightarrow s$ and $s \rightarrow q$. If $p \rightarrow q$ then we say that q *depends on* p; also we say that q *depends on* any rule where p occurs in the head. A program is *stratified* if there exists no rule r where a predicate p occurs in a negative literal in the body, q occurs in the head and $q \rightarrow p$, i.e., there is no recursion through negation.

A Datalog$^\neg$ program P is a function-free logic program whose predicates are partitioned into *extensional* and *intensional* predicates (EDB and IDB predicates, respectively). Positive and stratified Datalog$^\neg$ program are simply called Datalog and Datalog$^{\neg s}$ program, respectively. A Datalog$^\neg$ program P has associated a relational database scheme $DS_P = \{r \mid r \text{ is an EDB predicate symbol of } P\}$. EDB predicates are seen as database relation schemes and never occur in rule heads, since they are solely defined by the database facts.

The stable model semantics of a program P applied to a database D is given by the stable models for $P \cup D$. Given a stable model M for $P \cup D$ and a predicate symbol r, $M(r)$ denotes the relation on r defined as $\{t | r(t) \in M\}$.

The notion of *data complexity* [44]) is defined naturally by viewing the program P as a function computed on the database (which is thus viewed as the input variable). In the general case, finding a stable model for a given program is $\mathcal{NP}$-hard, and deciding whether a program admits some stable model is $\mathcal{NP}$-complete [39]. However, we will deal with particular classes of programs where the existence of a stable model is a known a-priori. For instance if a Datalog$^\neg$ program P is positive or stratified then there always exists a unique stable model for P applied to a database D, and this can also be computed in time polynomial in the size of D.

A *non-deterministic Datalog$^\neg$ query* Q is a pair $(g(X), P)$ where P is a Datalog$^\neg$ program, g is a predicate symbol in P, say with arity $n \geq 0$, and X is a list of n variables; the atom $g(X)$ is called a query-goal. Given a database D, the set of *answers* of Q on D, denoted by $Q(D)$, is the set of relations on g denoted as $A_g = \{M(g) | M \text{ is a stable model of } P \cup D\}$. A non-deterministic answer to such a query, or equivalently the answer to such a query under a non-deterministic interpretation, means returning a single relation in $Q(D)$; this corresponds to determining one stable model.

The query is called *deterministic* or *non-deterministic* according to whether the mapping is single-valued or multivalued. While the fact that there exists only one stable model implies that the query is deterministic, the opposite is not true. In fact for a query goal $g(X)$, we might have multiple stable models all sharing the same values of the g-predicates.

3 – Non-determinism and Functional Dependencies

Early attempts at introducing non-determinisms rule-based languages are the Witness operator [2] and the Static Choice operator [30]. Although these two were defined quite independently they share profound similarities, beginning with the usage of the notion of Functional Dependencies (FDs) for their definitions.

Witness Operator

An important non-deterministic extension of FO uses the *witness* operator [2, 3]. Let J be a relation (instance) over some attribute set $U = X \cup Y$, where X and Y are disjoint. Then, the witness operator, $W_X(J)$, selects one tuple $\langle X, Y\rangle \in J$ for each $\langle X\rangle \in \pi_X(J)$. Therefore, W_X selects a *maximal* subset $I \subseteq J$ satisfying the functional dependency $X \to Y$.

Example 1

Consider the binary relations STUDENT and PROFESSOR. The first column in student contains the student name and the second column gives his/her major. The first column in `professor` contains his name and the second column his/her department. In fact, we might have a toy database consisting of the following facts:

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

Now, the rule is that the major of a student must match his/her advisor's area. Then eligible advisors can be computed as follows:

$$\text{ELIG\_ADV} = \pi_{\$1,\$3}(\text{STUDENT} \bowtie_{\$2=\$2} \text{PROFESSOR}) \quad (2)$$

Thus for the previous database, we obtain:

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

But, since *a student can only have one advisor* we must use the operator

$W_{\$2}$ to express the fact that, in the actual advisor relation, the second column (professor name) must be FD from the first one (student name). Thus the assignment of an actual advisor might be performed through the following relational algebra+witness (FO+W) expression:

$$\text{ACTUL\_ADV} = \mathrm{W}_{\$2}\ (\pi_{\$1,\$3}(\text{STUDENT} \bowtie_{\$2=\$2} \text{PROFESSOR})) \tag{3}$$

The result of execution of this FO+W expression is non-deterministic. It can either give a singleton relation containing the tuple (`marc`, `ohm`) or that containing the tuple (`marc`, `bell`). □

Static Choice

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

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

Then, using $\mathcal{LDL}$'s static choice construct [32], the goal `choice((S),(P))` can be added to the rule forcing the selection of a unique advisor for each student. The goal `choice((S),(P))` denotes that the functional dependency `S` → `P` must hold in `actual_adv`:

Example 2
Computation of unique advisor by choice rules.

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

This rule produces the same result as the relational expression with witness of Equation 3: It produces a relation consisting of either the only fact `actual_adv(marc,ohm)` or of the fact `actual_adv(marc,bell)`. □

Static choice can therefore be viewed as a construct for declaring and enforcing FDs in Datalog. Observe that Datalog$^{\neg_s}$, i.e., Datalog with stratified negation, cannot express a non-deterministic query; e.g., it cannot select an arbitrary advisor for a student out of a set of several eligible advisors.

The original definition of choice [30, 32] relies on the re-writing of the program into one where the original choice rule is broken in three parts. For instance, the program of Example 2 is basically rewritten as follows (using a mixed algebra+rules notation where CHOSEN and EXTCHOICE denote the binary relations corresponding to the binary predicates `chosen` and `extchoice`, respectively):

Example 3
The definition of static choice applied to Example 2.

$$\begin{array}{lcl} \texttt{actual\_adv(S,P)} & \leftarrow & \texttt{student(S,Major), professor(P,Major),} \\ & & \texttt{chosen(S,P).} \end{array} \quad (5)$$

$$\textsc{chosen} = \mathrm{W}_{\$2}(\textsc{extchoice}) \quad (6)$$

$$\texttt{extchoice(S,P)} \leftarrow \texttt{student(S,Major), professor(P,Major).} \quad (7)$$

□

Thus the computation proceeds in three steps. In the first one, the choice rules without the choice goals are used to compute `extchoice`, where all possible alternatives are collected. Then, the FDs are enforced, by the witness operator (although an equivalent operator was used in [30, 32]), and, finally, the answers are computed. Thus, the enforcement of FDs is made a-posteriori, in a static fashion, after all the alternatives have been generated. From a computational viewpoint, it is preferable to enforce the FD constraint in a dynamic fashion, while the tuples are still being generated, by using `chosen` as an actual table where pairs (S, P) are memorized as they are produced. The table `chosen` is used to compare each new pair (S, P) generated against the previously generated ones, stored in `chosen`; if the addition of the new pair would violate the FD constraints the new pair is discarded, otherwise it is added to `chosen`. Facts produced by other rules with the same predicated name, `actual_adv`, are not compared against the old values in the table, since they are not subject to the FD constraint declared by the choice goal in Example 2; thus choice has a local scope inasmuch as this is restricted to a single rule. This more dynamic interpretation, produced the notion of dynamic choice, which also achieves more expressive power, and a more declarative semantics using the notion of stable models.

Non-determinism in Deterministic Queries

A programmer posing the query "Find an advisor for Marc" expects either one of the following two answers: `P = bell` or `P = ohm`. We thus have a *non-deterministic* program and a non-deterministic query, inasmuch as different answers are acceptable and expected for different executions of the same query on a database.

However, it is often the case that, while non-deterministic constructs are used in the rules, the same answer will always be returned for every execution of a query on a given database; then the query is deterministic.

Take for instance the following program which determines whether there are more boys than girls in a database containing the unary relations `boy` and `girl`:

Example 4
Are there more boys than girls in our database?

$$
\begin{array}{lcl}
\texttt{match(Bname, Gname)} & \leftarrow & \texttt{boy(Bname), girl(Gname).} \\
 & & \texttt{choice((Bname), (Gname)), choice((Gname), (Bname)).} \\
\texttt{matched\_boys(Bname)} & \leftarrow & \texttt{match(Bname, Gname).} \\
\texttt{moreboys} & \leftarrow & \texttt{boy(Bname)}, \neg\texttt{matched\_boys(Bname).}
\end{array}
$$

□

The first rule pairs-up boys and girls in a random fashion. Irrespectively of the matching, however, the name of some boy not appearing among the matched ones denotes that there are more boys than girls. Consider now a query G : ?`moreboys`. Since this is a *boolean query* (no variable in the query-goal) it either returns `true` or `false`. Moreover, for a given database of boys and girls, this query will deterministically return the same result, independent of the particular run. Therefore G is a deterministic query.

Observe that other queries on the same program are not necessarily deterministic. Take for instance the query goal `match(tom, mary)`. This always evaluates to `false` for a database where there is no `boy(tom)` or no `girl(mary)`, and it always evaluates to true when these two people are the only ones in the database. However, the query can return true or false on different runs for databases where there are several boys, including `tom`, and several girls, including `mary`; thus this is a non-deterministic query.

It is not known how to express the query `?moreboys` in Datalog$^{\neg s}$; it is actually conjectured that it cannot be done [3]. Therefore, this example illustrates how the most frequent use of non-deterministic constructs is in formulating deterministic queries that cannot be expressed (or is impractical to express) using only deterministic constructs. The sections that follow elaborate on this aspect.

4 – Dynamic Choice

The introduction of static choice in Datalog without negation does not extend the power of the language for deterministic queries [17]. We also conjecture that, although static choice adds some expressive power to Datalog$^{\neg s}$—as illustrate Example 4—Datalog$^{\neg s}$ with static choice cannot express all deterministic polynomial-time queries.

One possible approach to extend the language to express $PTIME$ consists in using more powerful forms of negation, such as negation under well-founded semantics [11]. The approach proposed in [36, 21] is instead based on the observation that the limitations of static choice follow from its inability to contribute to recursive predicates. This observation led to the introduction of *dynamic choice* [36, 21], which delivers major gains in expressive power without computational drawbacks, within the desirable framework of stratified negation.

Dynamic choice is no longer defined directly using FDs; rather choice programs are transformed into programs with negation which exhibit a multiplicity of stable models, each obeying the FDs defined by the choice goals. Each stable model corresponds to an alternative set of answers for the original program; each such a model is called a *choice model* for the original program. Following [36, 21], 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 1
The *stable version* $SV(P)$ of a choice program P is obtained by the following transformation. Consider a choice rule r in P:

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

where,
(i) $B(Z)$ denotes the conjunction of all the non-choice goals of r, and
(ii) $X_i,\ Y_i,\ Z,\ 1 \leq i \leq k$, denote vectors of variables occurring in the body of r such that $X_i \cap Y_i = \emptyset$ and $X_i, Y_i \subseteq Z$.

Then the original program P is transformed as follows:

1. replace r with a rule r' obtained by substituting the choice goals with the atom $chosen_r(W)$:

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

   where, $W \subseteq Z$ is the list of all variables appearing in choice atoms;

2. add the new rule:

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

3. add, for each choice atom $choice((X_i),(Y_i))$ $(1 \leq i \leq n)$, the new rule:

   $$diffChoice_r(W) \leftarrow chosen_r(W'),\ Y_i \neq Y_i'.$$

   where (i) the list of variables W' is derived from W by replacing each $A \in Y_i$ with a new variable $A' \in Y_i'$ (i.e., by priming those variables), and (ii) $Y_i \neq Y_i'$ is true if $A \neq A'$, for some variable $A \in Y_i$ and its primed counterpart $A' \in Y_i'$. □

Example 5
The stable version of the rule in Example 2 is (a student can be assigned a professor provided that a different professor has not been assigned):

$$
\begin{array}{lll}
\texttt{actual\_adv(S,P)} & \leftarrow & \texttt{student(S,Majr)}, \texttt{professor(P,Majr)}, \texttt{chosen(S,P)}. \\
\texttt{chosen(S,P)} & \leftarrow & \texttt{student(S,Majr)}, \texttt{professor(P,Majr)}, \neg\texttt{diffChoice(S,P)}. \\
\texttt{diffChoice(S,P)} & \leftarrow & \texttt{chosen(S,P}'\texttt{)}, \texttt{P} \neq \texttt{P}'.
\end{array}
$$

□

In general, the program $SV(P)$ generated by this transformation has the following properties [36, 21].

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

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

When the choice rules are non-recursive, the semantics of static choice and dynamic choice coincide [36]; in other words `chosen` and `diffChoice` of Example 5 operate as $\mathrm{W}_{\$2}$ in Example 3. For instance, the program consisting of the rules of Example 5 and the facts of Example 1 has two stable models, one where `marc` has `ohm` as advisor and one where he has `bell` instead. However, in recursive rules dynamic choice is more powerful than static choice inasmuch as the new definition embeds a memorization capability that is not present in static choice (or witness operator). In particular, the problem of Example 6, i.e., create a linear order on the elements of a given domain cannot be expressed by static choice.

Example 6
Ordering a domain. Constructing a total order for a domain D whose elements are represented by facts of the form `d(x)` (`root` is a distinguished new symbol).

$$
\begin{array}{lll}
\texttt{ordered-d(root,root)}. & & \\
\texttt{ordered-d(X,Y)} & \leftarrow & \texttt{ordered\_d(\_,X)},\ \texttt{d(Y)}, \\
 & & \texttt{choice((X),(Y))},\ \texttt{choice((Y),(X))}.
\end{array}
\tag{8}
$$

The choice goals here ensure that once an arc `(X,Y)` is generated, this is the only arc leaving the source node `X` and the only arc entering the sink node `Y`. □

Example 7
The arcs and the bijective FDs resulting from the application of the program in Example 6 to the facts `d(b), d(c), d(a).`

`ordered-d(root,root).`	$FDs:$
`ordered-d(root,b).`	$root \leftrightarrow b$
`ordered-d(b,c).`	$b \leftrightarrow c$
`ordered-d(c,a).`	$c \leftrightarrow a$

Assuming that the nodes are chosen in the order listed above, then, after that the tuple (`b,c`) has been inserted, the choice of the last node `a` is the only remaining possibility, since e.g., choosing `b` or `c` would violate the previous choices. After the last tuple (`c,a`) is inserted there is no further possible choice. □

The ability of chaining-up the elements in a relation, by using choice in recursion, extends the expressive power of Datalog. To prove this statement for positive Datalog programs, it suffices to observe that the inequality relationship $\neq$ among elements of the domain relation `d(X)` can be trivially defined on the chain representation of Example 6. Indeed, inequality is a deterministic query that cannot be expressed on an unordered domain using positive Datalog [29]. Furthermore, the inequality query cannot be expressed in Datalog with static choice or witness [17]. In fact, as per the static approach to the FD enforcement illustrated by the Example 3, if the choice constructs in Example 6 were instead interpreted under the static-choice semantics, then additional choice models would arise, including, e.g., one with the following atoms: `ordered-d(root, b), ordered-d(a, c), ordered-d(c,a).` Therefore, while the FDs are kept in these models, the chain property is lost.

Even more dramatic is the increased expressive power achievable by dynamic choice used in conjunction with stratified negation. Example 6 illustrates the ability of generating a total order for the elements in a domain by means of dynamic choice. By generating a total order on all the data items in the database, and using stratified negation is then possible to express all the deterministic $PTIME$ queries [3]. The next section is devoted to the discussion of such queries.

5 – Applications of Choice

Inasmuch as choice is a new construct, there is a need for more practice and understanding on how it can be effectively used in writing actual programs. The selected examples that follow are meant to address this need, and further elaborate on the expressive power issues previously described.

Example 8
A main extension of the relational model supports nested relations. This extension is easily accomplished in a logic-based framework using functors to define complex objects such as lists. Thus,

$$\texttt{dept\_list(shoe,[adams,jones,zong,smith])} \tag{9}$$

denotes that `[adams,jones, zong, smith]` is the list of employees working in the shoe department. Since we are dealing with sets, databases obtained by exchanging the order of employee names in the list are viewed as equivalent.

If we are restricted to Datalog proper, then lists are not allowed, and we might instead use the following representation, where the list is kept as a chain connecting the employees:

$$\begin{array}{l}\texttt{dept\_chain(shoe,adams).}\\ \texttt{emp\_chain(adams,jones).}\\ \texttt{emp\_chain(jones,zong).}\\ \texttt{emp\_chain(zong,smith).}\end{array} \tag{10}$$

□

From either lists or chains, it is simple to derive a set of 1NF facts representing the same information. For the chain representation of sets, for instance, we can use the following (transitive closure) rules:

Example 9
Normalizing relations.

```
flat(Dept,Emp)   ←  dept_chain(Dept,Emp).
flat(Dept,Emp2)  ←  flat(Dept,Emp1), emp_chain(Emp1,Emp2).
```

□

These rules applied to the previous `dept_chain` and `emp_chain` yield:

$$\begin{array}{l}\texttt{flat(shoe,adams).}\\ \texttt{flat(shoe,jones).}\\ \texttt{flat(shoe,zong).}\\ \texttt{flat(shoe,smith).}\end{array} \tag{11}$$

In summary, there exist simple recursive rules that transform a nested-relation representation to a flat (1NF) representation. The inverse transformation, however, reveals the limitation of stratified Datalog. It is not possible to translate flat relations into nested ones, unless we assume that there is an underlying order in the database or we use choice.

If choice is allowed, we can write the rules of Example 10 using recursive rules with choice and stratified negation. This program is similar to that of Example 6.

Example 10
De-normalizing relations.

```
echain(root, root)  ←  flat(shoe, _).
echain(Emp1, Emp2)  ←  echain(_, Emp1), flat(shoe, Emp2),
                       choice((Emp1), (Emp2)), choice((Emp2), (Emp1)).
empchain(Em1, Em2)  ←  echain(Em1, Em2), Em1 ≠ root.
depchain(shoe, Em)  ←  flat(shoe, Em), echain(root, Em), Em ≠ root.
```

□

Observe that we obtain the original nested-relation representation of Equation 11, modulo the renaming of `empchain` into `emp_chain`, of `depchain` into `dep_chain`, and, possibly, a different order for the employee-names in the chain.

Various deterministic queries which could not be expressed on flat relations are easily expressed on the the chain-based representation of Example 10. Aggregates represent a good example of such deterministic queries; the count of elements in a chain (i.e., the chain's length), or the sum of these elements, adds up to the same value irrespectively of the order in which the elements are arranged in the chain.

We will now express the aggregate that determines whether a given set has even cardinality; unlike many other aggregates, this does not require the use of arithmetic. Parity can be computed using the following program expressed in Stratified Datalog with dynamic choice, denoted Datalog$^{\neg s,c}$:

Example 11
The parity aggregate.

```
even(root).
odd(Y)       ←  even(X), ordered-d(X, Y).
even(Y)      ←  odd(X), ordered-d(X, Y).
succ(X)      ←  ordered-d(X, Y).
evenchain    ←  even(X), ¬succ(X).
```

□

The parity aggregate on a relation such as a flat relation (11) cannot be computed by Datalog$^{\neg s}$ [3]. Thus, dynamic choice properly extends the expressive power of Datalog$^{\neg s}$. In fact, as shown in [23], Datalog$^{\neg s,c}$ can express all deterministic $PTIME$ queries. To understand why this property holds, observe that the linking of the employees in the shoe department into a chain defines a total order on these atoms. By the same token, we can define a total order on all the constants in the database, thus producing the desired conclusion [3].

Static choice cannot order the elements of a relation into a chain. As explained, in more details in [21], the extended power of dynamic choice over static choice follows from its ability of (i) memorizing previous choices and (ii) applying the FD constraints dynamically during the recursive computation, rather than, statically, at the end.

Furthermore, while Datalog$^{\neg s,c}$ supports the expression of all deterministic queries in $PTIME$ [23], we conjecture that the parity query and many other low-complexity queries cannot be expressed in stratified Datalog with static choice.

The results presented above ensure the theoretical soundness Datalog$^{\neg s,c}$ as a language for database programming. However, more practical questions regarding the usability and the effectiveness of such a language can only be answered by writing actual programs. In the rest of this section, therefore, we express basic graph algorithms, such as depth first traversal with the help of choice, while, in the following section, we compare the uses of choice in Datalog against the uses of other non-deterministic constructs (such as Prolog's cut) in other logic-based languages.

Example 12

Spanning tree. The following program computes the spanning tree starting from the source node a for a graph where an arc from node b to d is represented by the database fact g(b, d).

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

Observe, that when an arc st(b, d) is added by the second rule, an entry is made in the chosen made to ensure that no st(X, d), can later be added if $\mathtt{X} \neq \mathtt{b}$. However, there is no entry in the chosen table for arc st(root, a), which is produced by the first rule. Thus the condition $\mathtt{Y} \neq \mathtt{a}$ must be added explicitly to the second rule, to prevent it from generating arcs leading back to a. These combined conditions ensure that the st derives a spanning tree rooted in a. □

A more complex algorithm is needed if, rather than any trees, one wants a tree generated by a depth-first traversal of a graph. This fundamental algorithm is at core of many other graph algorithms [1]. The following program generates a *depth-first, preorder* traversal of a directed graph.

Example 13
Depth-First Preorder Traversal of a directed graph. We are given a directed graph whose arcs are stored by means of facts of the form $\mathtt{g(x,y)}$ and an initial node $\mathtt{a}$. The first set of rules orders the immediate successors of a node $\mathtt{X}$ into a chain:

$$
\begin{array}{lcl}
\mathtt{next(N,start,start)} & \leftarrow & \mathtt{g(N,\_)}. \\
\mathtt{next(N,Y,Z)} & \leftarrow & \mathtt{next(N,\_,Y),\ g(N,Z),} \\
 & & \mathtt{choice((N,Y),(Z)),\ choice((N,Z),(Y)).} \\
\mathtt{first(N,Y)} & \leftarrow & \mathtt{next(N,X,Y),\ X = start,\ Y \neq start.} \\
\mathtt{last(N,Z)} & \leftarrow & \mathtt{next(N,Y,Z),\ \neg next(N,Z,\_).} \\
\mathtt{sterile(Y)} & \leftarrow & \mathtt{g(X,Y),\ \neg g(Y,Z).}
\end{array}
$$

The second set of rules does the actual traversal, starting from a root node $\mathtt{a}$, and proceeding downward. The first three rules defining **explored** arcs perform the downward traversal beginning from $\mathtt{a}$. During this traversal, an arc is classified as a tree-arc, when its end-node is not the end-node of a previously classified tree arc—a constraint enforced by the choice goal in the fourth rule below. Thus, downward traversal develops through the visitation of the first child of the last-developed tree arc (second rule), and then the visitation of the next sibling of a solved arc (third rule). The last three rules add an arc to set of solved arcs, when (i) this is a sterile tree arc, (ii) its end-node has been previously reached (by via another tree arc—the complementary condition of that used in the fourth rule) and (iii) the last child of the arc has been solved. At each step, only one of the recursive rules below is applicable.

$$
\begin{array}{lcl}
\mathtt{explored(a,Y)} & \leftarrow & \mathtt{first(a,Y).} \\
\mathtt{explored(Y,Z)} & \leftarrow & \mathtt{tree(X,Y),\ first(Y,Z).} \\
\mathtt{explored(Y,Z)} & \leftarrow & \mathtt{solved(Y,X),\ next(Y,X,Z).} \\
\mathtt{tree(X,Y)} & \leftarrow & \mathtt{explored(X,Y),\ choice((Y),(X)).} \\
\mathtt{solved(X,Y)} & \leftarrow & \mathtt{tree(X,Y),\ sterile(Y).} \\
\mathtt{solved(X,Y)} & \leftarrow & \mathtt{explored(X,Y),\ tree(X1,Y),\ X \neq X1.} \\
\mathtt{solved(W,Y)} & \leftarrow & \mathtt{last(Y,X),\ solved(Y,X),\ tree(W,Y).}
\end{array}
\tag{12}
$$

We next turn to the problem of detecting forward, back and cross arcs [1]. The transitive closure predicate $\mathtt{treach(y,x)}$ denotes the fact $\mathtt{y}$ is reachable from $\mathtt{x}$, using tree-arcs. Thus, an arc $\mathtt{explored(x,y)}$ is (i) a forward arc if $\mathtt{treach(x,y)}$ is true but $\mathtt{tree(x,y)}$ is not, (ii) a back arc if $\mathtt{treach(y,x)}$ is true, and (iii) a cross arc if it is neither a tree arc, nor a forward arc, nor a back arc.

$$\begin{array}{lcl} \texttt{treach(X,Y)} & \leftarrow & \texttt{tree(X,Y)} \\ \texttt{treach(X,Z)} & \leftarrow & \texttt{tree(X,Y), treach(Y,Z).} \\ \texttt{forward(X,Y)} & \leftarrow & \texttt{explored(X,Y), }\neg\texttt{tree(X,Y), treach(X,Y).} \\ \texttt{back(X,Y)} & \leftarrow & \texttt{explored(X,Y), treach(Y,X).} \\ \texttt{cross(X,Y)} & \leftarrow & \texttt{explored(X,Y), }\neg\texttt{tree(X,Y),} \\ & & \neg\texttt{forward(X,Y), }\neg\texttt{back(X,Y).} \end{array} \qquad (13)$$

□

A list of the nodes of resulting tree in pre-order traversal is also easy to compute. If one wants to compute such a list while computing the tree itself, then the rules in (12) will have to be modified with the addition of the following rules:

$$\begin{array}{lcl} \texttt{chain(root,a).} & & \\ \texttt{chain(Y,Z)} & \leftarrow & \texttt{chain(\_,Y), tree(\_,Z),} \\ & & \texttt{choice((Y),(Z)),choice((Z),(Y)).} \end{array} \qquad (14)$$

Thus, the pre-order traversal list is kept as a chain where a new chain-link `chain(X,Y)` is added after the addition of next tree arc. The two choice conditions in the rule ensure that `Z` is the end-node of the last-generated tree arc, rather than a previous one. Moreover to ensure that `Z` is the end-node of the last-generated tree arc, rather than one that will be generated later on, we must also replace every `tree(X,Y)` goal in (12) with the pair `tree(X,Y),chain(_,Y)`. Thus the second and sixth rule in (8) are modified—the last rule need not be changed.

While a detailed performance analysis of this algorithm is outside the scope of this paper, it should be noted that the optimal $O(e)$ time complexity (e the number of arcs reachable from `a` [1]) is obtainable with suitable storage structures[1].

6 – Comparison with other non-deterministic constructs

Inasmuch as non-deterministic constructs, such as Prolog's cut, are cornerstones of logic programming languages, it is useful to compare them to choice.

The effect of top-down pruning operators is to cut off portion of the search space of the program. Examples of such operators are Prolog's *cut*, !, the *committed choice* operator, |, of the parallel logic language PARLOG, and the *one-of* construct proposed in [15]. A thorough comparison of top-down pruning operators and dynamic choice is provided in [16]. That paper presents a compilation of

[1] For instance, one can assume that the seminaive fixpoint computation is used and that tuples generated so far are kept in hash-tables accessible in constant time.

top-down pruning operators into Datalog$^{\neg s,c}$ which provides a declarative characterization for top-down pruning operators. The paper uses a refinement of the technique used in [36], for transforming programs with *one-of* operators into programs with negation. The stable model semantics of transformed programs model the information which can be inferred using leftmost SLD-resolution augmented with the pruning mechanism. Building on this basis, the paper shows how the behavior of several top-down pruning operators, such as the committed choice construct, and Prolog's if-then-else and the cut.

The discussion of the complex transformation proposed in [16] would take us beyond the scope of this paper; thus we will limit ourselves to underscore the main points with the help of some examples. We will see that the different operators are bound by profound similarities, e.g., the ability of pruning search and increasing expressive power, but they are also divided by important syntactic and semantic differences, including:

- Bottom-up versus Top-down interpretation
- Predicate-based versus rule-based scope
- Valued-Based versus occurrence-based semantics
- Memoing versus memory-less capability.

Let us begin from the first point with the help of two examples which compute the *nondeterministic merge* of two lists in Parlog (Example 14) and Datalog with lists (Example 15).

Example 14
A Parlog program that merges the employees in the shoe department with those in the toy department.

```
shoe_toy_emp(W)          ←  dept_list(shoe, W1), dept_list(toy, W2),
                            merge(W1, W2, W).
merge([X|Xs], Ys, [X|Zs]) ←  | merge(Xs, Ys, Zs).
merge(Xs, [Y|Ys], [Y|Zs]) ←  | merge(Xs, Ys, Zs).
merge([ ], Ys, Ys).
merge(Xs, [ ], Xs).
```

The bar symbol, |, denotes the *committed choice* construct; this construct forces a unique, non-deterministic choice among all rules whose heads match with the goal. Let the query goal be: `shoe_toy_emp(Zs)`. At a generic step of the recursion, if both the second and third rule unify with the goal, then only one is chosen. Thus either `X` or `Y` will be added to the output list `Zs`. □

Notice that, if we replace the committed choice operator by Prolog's cut, the program of Example 14 becomes deterministic since the ordering of the clauses is relevant; thus, the Prolog program would compute the *append* of the two lists.

A choice program computing a nondeterministic *merge* of the list of employees in the `shoe` department (Equation 9) with list of employees in the `toy` department is as follows:

Example 15
A Datalog program computing nondeterministic *merge*.

$$
\begin{array}{lcl}
r_1 : \texttt{mergelist(Xs,Ys,[\,])} & \leftarrow & \texttt{dept\_list(shoe,Xs), dept\_list(toy,Ys).} \\
r_2 : \texttt{merge(Xs,Ys,[X|Zs])} & \leftarrow & \texttt{mergelist([X|Xs],Ys,Zs).} \\
r_3 : \texttt{merge(Xs,Ys,[Y|Zs])} & \leftarrow & \texttt{mergelist(Xs,[Y|Ys],Zs).} \\
r_4 : \texttt{mergelist(Xs,Ys,[Z|Zs])} & \leftarrow & \texttt{merge(Xs,Ys,[Z|Zs]), choice((Xs,Ys),Z).} \\
r_5 : \texttt{shoe\_toy\_emp\_list(Zs)} & \leftarrow & \texttt{merge([\,],[\,],Zs).}
\end{array}
$$

The rule r_1 is the activating point for the bottom-up evaluation. So the first iteration produces `mergelist(l1,l2,[])`, where `l1` and `l2` stand for the actual input lists; this fire the rules r_2 and r_3 which in turn fires rule r_4. The two predicates `merge` and `mergelist` recursively interact exchanging their roles as producer and consumer as follows. The rules defining `merge` produce the heads of the

two lists; only one such head is nondeterministically selected (*consumed*) by the rule defining `mergelist`. This last result is then fed back into `merge` and the cycle repeats. Finally, when both lists are fully consumed, the predicate `shoe_toy_emp_list` contains the output list. □

This example provides an opportunity for several observations. The first observation is that, in program of the Example 15, lists can be replaced by chains as it was done in previous examples. A second observation is that the pruning that follows from a choice goal is local to the rule which contains the goal; thus choice is similar to the one-of operator in this respect. On the other hand, the scope of Parlog's committed choice is global; i.e., it extends to all the rules defining the same predicate. In the case of Prolog, the scope of the cut extends to the rules and goals following the cut.

Actually, versions of the choice having global scope have also been proposed [35]. Semantically, there is not a profound difference between the different versions, which are also equivalent in terms of expressive power. In fact a program under choice with local semantics can be easily translated into an equivalent program with choice under global semantics. Nevertheless the different versions can simplify the program in certain situations, and complicate it in others. For instance, the use of global (i.e., predicate based) scope for choice would avoid the need to introduce the additional `mergelist` predicate in Example 14.

Let us consider now the important issues of (i) occurrence-based versus value-based pruning and (ii) memoing. For instance, consider the program of Example 15, where the choice rules explicitly state that the choice of the next element `Z` is a function of the two input lists[2]. In the Parlog example, a new choice is performed every time a new instance of a rule with the non-deterministic construct is used and a choice point is crossed. Thus, the declaration of resulting FDs remains implicit and there is no notion of memorizing previous choices. In many situations, including Example 15, this leads to simple programs and efficient pruning. In general, however, the occurrence-based semantics causes many problems: the two main problems are (i) loss of declarative semantics, and (ii) loss of expressive power.

The loss of declarative semantics follows immediately from the fact that any occurrence-based interpretation of pruning becomes totally dependent on the execution model, and can thus change dramatically with the order of execution. To illustrate the loss of expressive power, let us return to the issue of deriving a nested-relation representation of employees working in the shoe department, from the database of flat relations. In Prolog, we write the following program:

Example 16
Building nested relations in Prolog.

```
member(X, [X|_]).
member(X, [_|L])   ←  member(X, L).
elist([])          ←  flat(shoe, _).
elist([E|Rest])    ←  elist(Rest), flat(shoe, E), new(E, Rest).
new(E, Rest)       ←  not(member(E, Rest)), !.
```

□

This program illustrates how an ordering of the elements of a set can be expressed in Prolog with stratified negation and lists. The goal ?`elist(L)`, will return lists of increasing length. Thus the complete program also have rules, here omitted for brevity, that select the list that contains all the employees working in the shoe department (this can be done by double negation) and then stop (this can be done by a cut). Now, the program in Example 16 can be translated into an equivalent choice program by replacing the goal `new(E, Rest)`, by its definition, and then replacing the cut by `choice((Rest), (E))`. Then we obtain the following program:

[2]An experienced programmer, being aware the cost involved in memorizing the two lists to support choice, will probably use a different programming technique. A natural one, consists of having an additional column indicating the depth of the recursion (an of the constructed list) and use this as the left side of the choice FD.

Example 17
A choice program equivalent to the Prolog program of Example 16.

```
member(X, [X|_]).
member(X, [_|L])    ←  member(X, L).
elist′([])          ←  flat(shoe, _).
elist′([E|Rest])    ←  elist′(Rest), flat(shoe, E),
                       ¬member(E, Rest), choice((Rest), (E)).
```

□

The choice goals in this program plays the same role as the cut in the original program, i.e., ensuring that only one value `E` is taken for each activation of the rule in the recursive loop. Likewise, the witness operator could be used here without changing the meaning of the program. Thus, we can use this example to illustrate the pruning ability shared by all nondeterministic constructs.

The list of all employees produced by these programs can be used to compute whether the shoe department has a even number of employees. Basically the list will have to be traversed to see if it contains an even number of elements: this can be done using recursive rules similar to those used in Example 11. Furthermore, observe that if the cut is eliminated from the program in Example 16 then the resulting program lists the employees in the toy department, in all possible permutations. Since all these lists have the same length, the parity query always returns the same result. Similar conclusions follow if goal `choice((Rest), (E))` is eliminated from the program in Example 17. Thus, in these examples involving lists, the cut the and choice constructs do not make much difference in terms of expressive power. However, they make great difference in terms of performance, since a program that would compute in an exponential number of steps (w.r.t. the length of the list since all permutations of the list are generated), becomes polynomial-time once the cut or an equivalent choice goal is used.

While choice was also applied to formalize Prolog's cut [16], its full potential is hardly achieved by using it to replace the cut, as done in Example 16. In fact, the memorization ability of dynamic choice allows us to write a much simpler program, as follows:

Example 18
Nesting relations by choosing and remembering.

```
elist″([]).
elist″([E1, E2|Erest])  ←  elist″([E2|Erest]), flat(shoe, E1),
                           choice((E1), (E2)), choice((E2), (E1)).
```

□

The significance of this fact, is best appreciated in the framework of Datalog, where lists or other function symbols are not allowed; therefore, lists have to be represented through chains, and the programs of Examples 16 and 18 will have to be modified to work with this representation, as it was done in Example 9 (a simple re-writing that we leave to the reader). For Example 18, the result of this translation is a stratified program with choice.

The program in Example 16 can also be translated along the same lines, but the result is a program that is no longer stratified—and will not work under Prolog's negation-by-failure semantics (the program will loop forever). On the other hand, the rules of the resulting non-stratified program could be translated into relational algebra + witness and computed under the inflationary fixpoint semantics [3].

Several conclusions emerge from the previous discussion. One is that all non-deterministic constructs improve performance by pruning search. The second is that they also increase the expressive power, although different constructs do that to a different extent. In general, expressive power of constructs must be evaluated in a minimalist's context; i.e., where each construct can be measured orthogonally and independently from others, as far as possible. For instance, we have seen that the ability of the cut of chaining-up elements in a relation holds only when lists are allowed; this capability disappears once lists or other function symbols are taken away. Likewise, the witness operator's ability of chaining-up elements in a relation and achieving $PTIME$ holds if negation is allowed in recursive rules (under inflationary semantics). But the witness operator appears to loose this ability when only stratified negation is allowed. In such a context, the memorization ability of dynamic choice sets it apart from other non-deterministic constructs, such as witness, cut and committed choice, and explains its unique ability of extending the expressive power of Datalog to $PTIME$.

7 – Fixpoint Computation and Extensions of Choice

Further extensions of choice have been proposed using its constructive characterization called the choice fixpoint.

With I an interpretation for a program P, $T_P(I)$ is defined as the set containing the heads of each rule $r \in ground(P)$ s.t. all positive goals of r are I, and none of the negated goals of r, is in I. Then, for a choice program P, let us 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 1. Analogously, we denote by T_{P_H} the immediate consequence operator associated with the non-*chosen* rules in $SV(P)$. Therefore, we have that, for any interpretation I of $SV(P)$:

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

Following [17] we can now introduce a general operator for computing non-deterministic fixpoints of choice programs.

Definition 2
Given a choice program P, its *non-deterministic immediate consequence operator* Ψ_P is a map from an interpretation of $SV(P)$ to a set of interpretations of $SV(P)$ defined as follows:

$$\Psi_P(I) = \{ \ J \ \cup \ T_{P_H}^{\uparrow\omega}(J) \mid \ J \in \Gamma_P(I) \ \} \tag{15}$$

where:

$$\begin{array}{rcl} \Gamma_P(I) & = & \{ \ I \cup \Delta I \ \mid \ \emptyset \subset \Delta I \subseteq T_{P_C}(I) \setminus I \text{ and } I \cup \Delta I \models FD_P \ \} \\ & & \cup \ \{ \ I \ \mid \ T_{P_C}(I) \setminus I = \emptyset \ \}. \end{array} \tag{16}$$

Where $I \cup \Delta I \models FD_P$ denotes that $I \cup \Delta I$ satisfies the dependencies in FD_P. □

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 subsets of the derived choices, in order to derive all consequences of the operated choices. Non-determinism is due to fact that there are many such subsets, and, any of these can be chosen.

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, if there are possible new choices, then at least one has to be taken.

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 3
Given a choice program P, an *inflationary choice fixpoint computation* for 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 $n \geq 0$. □

Inasmuch as every sequence $\langle I_n \rangle_{n \geq 0}$ is monotonic, it has a unique limit for $n \rightarrow \infty$; this limit will be called an *inflationary choice fixpoint* for the choice program P. Thus, we have the following result [21]:

Theorem 4
Let P be a Datalog program with choice, and M a Herbrand interpretation for $SV(P)$. Then M is a dynamic choice model for P iff M is an inflationary choice fixpoint for P. □

Thus, the inflationary choice fixpoint is *sound* (every result is a choice model) and *complete* (for each choice model there is some inflationary choice fixpoint computation producing it). For logic programs with infinite Herbrand universe, an additional assumption of fairness is needed to ensure completeness.

Definition 2 leaves quite a bit of latitude in the computation of Δ, (Equation16). This freedom can be used to select Δ's that have additional properties. In many practical applications in particular, it makes sense to trade non-deterministic completeness (which is only of abstract interest to a programmer) for very concrete benefits, such as expressive power and performance. This leads to the definition of eager, lazy and greedy choice discussed in the next subsections.

Eager Dynamic Choice

Eager Dynamic Choice is simply defined by requiring ΔI in equation to be maximal. While completeness is thus lost, a dramatic increase in the expressive power occurs, as demonstrated by the next example based on [21], which shows how to emulate negation using eager choice. The following choice program defines a relation *not_P* as the complement of a predicate p with respect to a universal relation U. We assume here that both P and U are extensional relations.

Example 19
Negation by eager choice. The choice program $NOT[P, U]$ consists of the following rules:

$$\begin{array}{ll} r_1: & \texttt{not\_P(X)} \leftarrow \texttt{comp\_P(X,1)}. \\ r_2: & \texttt{comp\_P(X,I)} \leftarrow \texttt{tag\_P(X,I)},\ \texttt{choice((X),(I))}. \\ r_3: & \texttt{tag\_P(nil,0)}. \\ r_4: & \texttt{tag\_P(X,0)} \leftarrow \texttt{P(X)}. \\ r_5: & \texttt{tag\_P(X,1)} \leftarrow \texttt{U(X)},\ \texttt{comp\_p(\_,0)}. \end{array}$$

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

Observe that, by rule r_2, the relation *comp_P* is a subset of the relation *tag_P*, which must obey the FD $(X \rightarrow I)$. Now, because of rule r_4, the eager choice in r_2

binds all the elements in the extension of P to 0. Then all the elements which do not belong to P will be chosen in the next step of the eager choice fixpoint, and hence will be bound to 1. Rule r_3 handles the case where relation P is empty. Thus, by the operational fixpoint 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; thus r_1 delivers the negation of P.

Essentially, this example shows that the eager dynamic choice offers a flexible mechanism for handling the control needed to emulate the set-difference of the two relations. It is shown in [13] that the program of Example 19 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 and we have the following result [13].

Theorem 5
Datalog with eager dynamic choice can express all deterministic queries in $PTIME$. □

Lazy Dynamic Choice

In contraopposition to eager choice, we have the notion of *lazy dynamic choice* which is obtained by minimizing Δ, i.e., by letting it be a singleton set $\Delta = \{\delta\}$. Then, the inflationary choice fixpoint operates by repeating the following cycle: fire all Horn clauses till no further progress can be made (saturation) and then choose a single new fact using the `chosen` rule. As shown in [17], the inflationary choice fixpoint computation under the lazy policy is as powerful as the original one—i.e., it provides a *sound* and *complete* computation for choice models. Furthermore, Datalog with lazy dynamic choice can only express deterministic queries that are monotonic [17]. Therefore, Datalog with lazy dynamic choice is less expressive than Datalog with eager dynamic choice [13], since the latter can express all $PTIME$ queries. In order to express all $PTIME$ queries with lazy dynamic choice, therefore, stratified negation must also be supported [14]:

Theorem 6
Datalog$^{\neg s}$ with lazy dynamic choice can express all deterministic queries in $PTIME$. □

Greedy Choice

When dealing with ordered domains, it is often useful to select a value that is the least (or the most) among those that can be selected without compromising the choice FDs. Therefore, with X denoting one or more variables of r and A

denoting one single variable, a goal `choice-least((X),(A))` in a rule r, can be used to denote that the FD `choice((X),(A))` is to be satisfied using least values of `A` [22].

We can now define a choice-least rule as one that contains one choice-least goals and possibly some other choice goals. A program that contains one choice-least rule, and possibly some rules with only choice goals, is called a *choice-least* program. The correct computation for `choice-least` program can be thus defined by modifying the non-deterministic immediate consequence operator of Definition 2.

Definition 7
Let P be a choice-least program with the goal `choice-least((X),(A))`. Then, the *greedy immediate consequence operator* for P, is denoted Γ_P and defined as follows:

$$\begin{array}{lcl} \Theta_I & = & \{x \in T_{P_C}(I) \setminus I \ \mid \ I \cup \{x\} \models FD_P\} \\ \Gamma_P(I) & = & \{I \cup \{\delta\} \ \mid \ \delta \in min_A(\Theta_I)\} \ \cup \ \{I \ \mid \ \Theta_I = \emptyset\}. \end{array} \tag{17}$$

where $min_A(\Theta_I)$ denotes the subset of tuples of Θ_I which are minimal in the value of A.

□

Thus, all the tuples that do not violate the given FDs (including the FDs implied by choice-least) are considered, and one is chosen that is minimal. Thus greedy choice is a special case of lazy choice, and the inflationary choice fixpoint computation restricted to make greedy selections is still sound and produces choice models, although completeness is normally lost. This computation will be called the *greedy choice fixpoint*.

The `choice-least` operator (or its converse, `choice-most`) supports the convenient and efficient formulation of many greedy algorithms, such as Dijkstra's shortest path and Prim's minimum-spanning tree algorithms.

In Example 6, the elements of a domain `d(X)` are chained in an arbitrary order. Say now that a lexicographical order is pre-defined, and we would like to sort these elements accordingly. Then, we can write the rules as follows:

Example 20
Sorting the elements of a relation.

```
ordered-d(root,root).
ordered-d(X,Y)        ←  ordered_d(_,X), d(Y),
                         choice-least((X),(Y)), choice((Y),(X)).
```

□

The `choice-least((X),(Y))` goal in our program ensures that the following two constraints are satisfied:

- the FDs defined by `choice((X),(Y))` must hold, and
- `Y` must be the least among the alternatives that satisfy the given set of FDs.

The benefits of using the *greedy choice fixpoint* as concrete semantics follow from the fact that a programmer knows from the `choice-least((X),(Y))` goal that particular choice models are produced by selecting least values for `Y` at each step, rather than arbitrary ones. Then, the programmer can formulate the least-cost spanning tree algorithm as follows:

Example 21
A program to construct a least-cost spanning tree in a directed graph with root node a; the graph is represented by `cg(X,Y,C)`, where C denotes the cost of the arc from X to Y.

$$
\begin{array}{lll}
\mathtt{prim(root,a,0).} & & \\
\mathtt{prim(X,Y,C)} & \leftarrow & \mathtt{prim(\_,X,C),\ \ cg(X,Y,C),\ \ Y \neq a,} \\
 & & \mathtt{choice((Y),(X)),\ \ choice\text{-}least((X,Y),(C)).}
\end{array}
$$

□

At any point in the algorithm, `prim(_,X,C)` is a set of nodes in the spanning tree constructed so far. The goals, `prim(_,X,C),cg(X,Y,C),Y` $\neq$ `a,choice((Y),(X))` produce a set Δ' of candidate new nodes reachable from these, each with their respective cost. Then, the goal `choice-least((X,Y),(C))` selects an arc with least cost `C` out this Δ' set. For this example, therefore, the inflationary choice fixpoint computation behaves exactly as Prim's celebrated algorithm [1]. The final result is a choice model in every case, and a minimum-cost spanning tree when all arcs are known to have a non-negative cost. When this is not the case, a choice model is still produced, but this need not be a minimum cost spanning tree[3].

The use of choice-least (or its dual choice-most) represents only one example where the use of constraints to guide the computation of choice models can be very beneficial. Several other applications are being investigated.

[3]The reader can easily contrast this program with that obtained by replacing `choice-least` with a goal `least`, which in turn can be re-expressed using negation [22]. When all arcs have a positive cost, the program with `least` has stable models which coincide with those obtained by the greedy choice fixpoint above. Otherwise, the program with `least` might not have stable models, or if it has some, they need not coincide with the results of the greedy choice fixpoint.

8 – Beyond Polynomial Time

The dynamic choice construct is also important in expressing algorithms beyond polynomial data complexity. The two main approaches explored so far are discussed next.

Possibility and Certainty Semantics

An interesting approach to extend the expressive power of a language with choice, or other non-deterministic constructs, is explored in [23]. Consider a program P with stratified negation and lazy dynamic choice and let Q be a boolean query on P. Then it is possible to specify any of the following three semantics:

1. **Non-deterministic semantics**: Compute an arbitrary choice model M for P and compute whether Q is true or false in this model. This query is denoted: $!(Q)$
2. **Possibility semantics**: Compute if there exists some choice model M for P for which Q is true. This query is denoted: $\exists(Q)$
3. **Certainty semantics**: Compute if Q is true in every choice model M of P. This query is denoted: $\forall(Q)$.

Then, the first semantics is basically that we have considered so far and captures $PTIME$. Non-polynomial expressive power, is, however, available under the other two semantics. For instance the following program and query decides whether there is an *Hamiltonian path in a graph.* This $\mathcal{NP}$-complete problem [19] is expressible through another variation of our basic chain program.

Example 22
Hamiltonian path in a graph. We are given a undirected graph whose nodes and arcs are stored as facts of the form $\mathtt{n(x)}$ and $\mathtt{g(x,y)}$, respectively.

```
simple-path(root,root).
simple-path(X,Y)      ←  simple_path(_,X), g(X,Y),
                         choice((X),(Y)), choice((Y),(X))
non-hp-path           ←  n(X), ¬simple-path(_,X).
hp-path               ←  ¬non-hp-path.
```

The graph has an Hamiltonian path iff there is a simple path which visits all nodes exactly once, that is iff the query $?\exists(\mathtt{hp\text{-}path})$ has answer *true*. □

The possibility semantics here require to find a choice model for which the goal `?hp-path` is true, if such a path exists. Thus, different choices will be considered till one is found for which `hp-path`, i.e., every node is in the circuit. In practice, this can then be easily extended to a constructive, open-query semantics, where the actual Hamiltonian circuit is returned to the user.

The problem of characterizing the expressive power of Datalog$^{\neg_s}$ programs with (lazy dynamic) choice constructs, under the different semantics has been studied in [23], yielding the following theorems:

Theorem 8
Under the possibility *semantics, Datalog*$^{\neg_s,c}$ *expresses exactly the class of boolean queries in* $\mathcal{NP}$. □

Theorem 9
Under the certainty *semantics, Datalog*$^{\neg_s,c}$ *expresses exactly the class of boolean queries in co*$\mathcal{NP}$. □

Thus, under these more advanced versions of semantics, the expressive power of Datalog$^{\neg_s,c}$ is comparable to that of stable models. However, by using choice and different query modalities the user can match the implementation algorithm to the complexity of the problem at hand. For instance, in order to compute the *PTIME* examples discussed in the previous sections, the user will request a non-deterministic semantics (by attaching the "!" annotation to the query); this results in the derivation of one choice model— a polynomial-time computation. This choice-based approach, is clearly preferable to the alternative of using only negation under stable model semantics; in fact, this would translate into an exponential implementation algorithm even for polynomial problems.

Even higher expressive powers are achievable by combining existential and universal queries [23].

Emulating Production Systems

A non-deterministic version of Datalog called N_Datalog¬* has been proposed in [4]. N_Datalog¬* is a powerful non-deterministic language that expresses exactly the queries in *PSPACE* [3]. The rules of N_Datalog$^{\neg *}$ have the form:

$$A_0(X_0), \ldots, A_m(X_m), \neg B(Y_1), \ldots, \neg B_n(Y_n) \leftarrow \quad P_1(Z_1), \ldots, P_k(Z_k), \neg Q_1(W_1), \ldots, \neg Q_h(W_h).$$

According to the non-deterministic semantics, at each fixpoint iteration a new instance is computed by choosing only one instantiation among the possible ones of the single rule chosen among the firable ones. The negated goals in the head of the rule are interpreted as updates, so the rule can also be viewed as a rule of a production system:

$$ACTION \leftarrow CONDITION$$

or

$$\textbf{if } CONDITION \textbf{ do } ACTION$$

Thus, the actual semantics of N_Datalog$^{\neg *}$ is based on the imperative semantics of non-deterministic *active rule* systems, rather than the declarative logic-based semantics of deductive databases. In this section however, we show how N_Datalog$^{\neg *}$ can be easily emulated by the Datalog$_{1S}$ programs with choice, denoted Datalog$^{c}_{1S}$. The merits of Datalog$_{1S}$ for modeling temporal and dynamic systems have, for instance, been described in [12, 45, 7]. In Datalog$_{1S}$, predicates may have an additional argument called the *stage argument.* Values in the stage argument are taken from the domain 0, 0+1, 0+1+1, ..., that is the domain of integers generated by using the infix successor function +1. For instance, the integer 3 is represented as 0+1+1+1. Alternatively, using the standard functional notation, the successor of `J` can denoted by `s(J)`, and this notation is at the root of the name Datalog$_{1S}$. Atoms have the form $\mathtt{p}(\mathtt{J},\vec{\mathtt{t}})$ where J is a stage term and $\vec{t}$ is a list of terms. Also, in conjunction with recursive Datalog$_{1S}$ rules, we allow programs to follow a particular form of compilable local stratification, called XY-stratification [45].

We now present a technique that given a N_Datalog$^{\neg *}$ program produces an XY-stratified Datalog$^{c}_{1S}$ program which is equivalent to N_Datalog$^{\neg *}$, i.e., it emulates the non-deterministic semantic of such language. The translation is based on a transformation technique introduced in [13], which *eager dynamic choice* to replace negation. We use Datalog$_{1S}$ to model state changes, via a limited usage of the integer successor function [12]. For the sake of simplicity we present the transformation technique by means of an example. The final program is an optimized version of what the transformation algorithm will return.

Example 23
Array ordering by adjacent element swapping. Let us consider the nondeterministic ordering of an array performed by swapping at each step any two adjacent elements for which the ordering does not hold.

$$\textbf{if } a[I] > a[I+1] \textbf{ do } swap(a[I], a[I+1])$$

where the procedure *swap* performs the multiple assignment:

$$a[I], a[I+1] := a[I+1], a[I]$$

□

To write a N_Datalog$^{\neg *}$ version of this problem, we have to find a representation that avoids the use of array indices and index arithmetic. If we assume that all the entries in our array are unique, we can use the chain representation used in the previous examples; moreover we will assume that the distinguished symbols `bot` and `top`, respectively, represent the first element and the last element in the chain. Then we obtain the following program:

Example 24
Array ordering by adjacent element swapping in N_Datalog$^{\neg *}$

$$\begin{array}{lcl} \mathtt{a(X1,Y), a(Y,X), a(X,Y1), \neg a(X1,X), \neg a(X,Y), \neg a(Y,Y1)} & \leftarrow & \mathtt{a(X1,X), a(X,Y),} \\ & & \mathtt{a(Y,Y1), X > Y.} \end{array}$$

□

The Datalog$^{c}_{1S}$ emulation of this N_Datalog$^{\neg *}$ program is as follows:

Example 25
Array ordering by adjacent element swapping in Datalog$^{c}_{1S}$

$$\begin{array}{lcl} \mathtt{ar(0,X,Y)} & \leftarrow & \mathtt{a(X,Y).} \\ \mathtt{swp(s(J),X,Y)} & \leftarrow & \mathtt{ar(J,X,Y),\ X > Y,\ choice((J),(X,Y)).} \\ \mathtt{add(s(J),Y,X)} & \leftarrow & \mathtt{swp(s(J),X,Y).} \\ \mathtt{add(s(J),X1,Y)} & \leftarrow & \mathtt{swp(s(J),X,Y),\ ar(J,X1,X).} \\ \mathtt{add(s(J),X,Y1)} & \leftarrow & \mathtt{swp(s(J),X,Y),\ ar(J,Y,Y1).} \\ \mathtt{del(s(J),X,Y)} & \leftarrow & \mathtt{swp(s(J),Y,X).} \\ \mathtt{del(s(J),X1,X)} & \leftarrow & \mathtt{swp(s(J),X,Y),\ ar(J,X1,X).} \\ \mathtt{del(s(J),Y,Y1)} & \leftarrow & \mathtt{swp(s(J),X,Y),\ ar(J,Y,Y1).} \\ \mathtt{ar(s(J),X,Y)} & \leftarrow & \mathtt{ar(J,Y,X),\ \neg del(s(J),X,Y).} \\ \mathtt{ar(s(J),X,Y)} & \leftarrow & \mathtt{add(s(J),Y,X).} \end{array}$$

□

The first rule initializes the staged *array* predicate with the initial state of the array. The second rule, non-deterministically selects a pair of `(X,Y)` to be switched. Observe the use of *choice* to ensure that, at each step, only one pair out of all the eligible pairs is switched. The `add` computes the new values that will be added, while `del` computes the pairs to be deleted. The last two rules in fact implement frame axioms, where the old pairs, excluded the deleted ones, are copied, while the new ones are added.

This example, illustrates a general programming paradigm, where (i) Datalog$_{1S}$ is used to record various steps in the computation, (ii) choice is used for non-deterministically selecting actions at each step, and (iii) the frame axioms is used for establishing the content of a new state; this technique provides a very powerful modeling paradigm that has been successfully applied to many problems, including, e.g., planning problems [7]. Other non-deterministic extensions of Datalog, such as [40], can also be captured with this technique.

The final program obtained by the transformation above is XY-stratified, modulo choice [45, 7]. XY-stratification is a particular form of local stratification whereby, using the syntactic structure of the program, local strata can be determined at compile time [45]. Due to this syntactic structure, the heads of recursive Datalog_{1S} rules can be viewed as defining new values for the predicates. Goals of recursive rules sharing the same stage values as those in their heads refer to new values of the predicates, while goals with a stage argument one less than those of their heads refer to the old values of the predicates. The unique stable model for an XY-stratified program can be computed in polynomial time. Also each stable model of a program that is XY-stratified modulo choice can be computed in polynomial time. Moreover, due to the particular syntactic structure of programs, old states need not be memorized, and copy rules, such as those produced by the frame axioms used in this section can be supported at no cost. A more detailed discussion of these properties can be found in [7].

9 – Conclusions

While the notion of non-determinism has long been recognized as central to logic programming, non-deterministic constructs of deductive database languages have, in the past, received only passing attention. The objective of this paper, therefore, has been that of providing a brief introduction to such constructs with the aim of illustrating their considerable importance from a practical and theoretical viewpoint. On the theoretical side, the paper has discussed why constructs such as dynamic choice are needed in $\text{Datalog}^{\neg s}$ to achieve $PTIME$ completeness. On the practical side, the paper has shown why this construct is needed for writing basic programs, such as relation nesting, set aggregate computation, and depth-first graph traversal.

The paper has also stressed the advantages that constructs such as dynamic choice provide over non-deterministic constructs of logic programming languages, such as the cut or committed choice. The first advantage is the memorization ability of choice, which leads to greater expressive power. The second advantage of choice follows from its declarative nature, since choice can be re-expressed through programs with negation under stable model semantics. This equivalence is significant w.r.t. computational aspects of non-monotonic reasoning. In fact, high-complexity problems in non-monotonic reasoning can be re-expressed in simpler and computationally more effective formulations by means of dynamic choice combined with simpler forms of negation, such as stratified negation.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *Data Structures and Algorithms*, Addison Wesley,1983.

[2] S. Abiteboul, E. Simon, V. Vianu, Non-Deterministic Language to Express Deterministic Transformation, in *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*,1990, pp.215–229, .

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

[4] S. Abiteboul and V. Vianu, Datalog Extensions for Databases Queries and Updates, Journal of Computer and System Science, 43,1991,pp.62–124.

[5] F. Afrati, S. S. Cosmadakis, M. Yannakakis, On Datalog vs. Polynomial Time, Journal of Computer and System Science, Vol. 51, No.1,1991, pp.177–196.

[6] C. Apt, H. Blair, A Walker, Towards a Theory of Declarative Knowledge, in *Proc. Workshop on Found. of Deductive Database and Logic programming*, (I. Minker Ed.),1988,pp.89–148.

[7] A. Brogi, V.S. Subrahmanian and C. Zaniolo, The Logic of Linear and Non-Linear Plans: A Deductive Database Approach, this issue of AMAI.

[8] A. Chandra and D. Harel, Computable Queries for Relational Databases, Journal of Computer and System Science, 21,1980,pp.156–178.

[9] A. Chandra and D. Harel, Structures and Complexity of Relational Queries, Journal of Computer and System Science, 25,1982, pp.99–128.

[10] A. Chandra and D. Harel, Horn clause and generalizations, Journal of Logic Programming, Vol. 2, No.1,1985, pp.1–15.

[11] W. Chen and J. Zeng, Nondeterminism through Well-Founded Choice, Journal of Logic Programming, Vol. 26, No.3,1996, pp.285–309.

[12] J. Chomicki, Temporal deductive databases, in *Temporal Databases: Theory, Design and Implementation*, A. Tansel et al. eds., Benjamin/Cummings, 1993.

[13] L. Corciulo, F. Giannotti, D. Pedreschi, Datalog with Non-deterministic Choice Computes *NDB-PTIME*, in *Proc. Deductive and Object-oriented Databases, Third International Conference, DOOD'93*, S. Ceri, K. Tanaka, S, Tsur eds., LNCS n. 760,1993,pp.49-66.

[14] L. Corciulo, F. Giannotti, D. Pedreschi, C. Zaniolo, Expressive Power of Non-Deterministic operators for logic based languages, in Proc.2nd ICLP Workshop on Deductive Databases and Logic Programming, Santa Margherita Ligure, June 1994,U. Geske, D. Seipel eds., GMD-Studien n. 231, 1994, pp 27-40.

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

[16] F. Giannotti, D. Pedreschi, C. Zaniolo, Declarative Semantics for Pruning Operators Logic Programming, Methods of Logic in Computer Science, 1, 1994, pp.61-76.

[17] F. Giannotti, D. Pedreschi, C. Zaniolo, Semantics and Expressive Power of Non-Deterministic Constructs in Deductive, Technical Report C96-04, Istituto CNUCE, Pisa,1996, submitted for publication.

[18] S. Ganguly, S. Greco, and C. Zaniolo, Extrema Predicates in Deductive Databases, Journal of Computer and System Science, Vol. 51, No. 1, 1991, pp.244–259.

[19] M. R. Garey and D. S. Johnson, *Computers and Intractability. A Guide to the Theory of NP-Completeness*, Freeman, 1979.

[20] M. Gelfond and V. Lifschitz, The stable model semantics of logic programming, in *Proceedings of the Fifth Intern. Conference on Logic Programming*,1988, pp.1070–1080.

[21] F. Giannotti, D. Pedreschi, D. Saccà, and C. Zaniolo, Nondeterminism in deductive databases, in *Proc. 2nd Int. Conf. on Deductive and Object-Oriented Databases*, 1991,pp.129–146.

[22] S. Greco, C. Zaniolo, and S. Ganguly, Greedy by Choice, *Proceedings of the Eleventh ACM Symposium on Principles of Database Systems*,1992, pp.105–163.

[23] S. Greco, D. Saccà, and C. Zaniolo, DATALOG Queries with Stratified Negation and Choice: from P to D^P in *Proc. of the Fifth Int. Conf. on Database Theory*, 1995, pp. 82-96.
[24] N. Immerman, Languages that Capture Complexity Classes, SIAM Journal of Computing, 16(4),1987, pp.760–778.
[25] D. S. Johnson. A Catalog of Complexity Classes, Handbook of Theoretical Computer Science, Vol. 1, J. Leewen ed. North-Holland,1990, pp.67–161.
[26] P. C. Kanellakis Elements of Relational Databases Theory, Handbook of Theoretical Computer Science, Leewen ed. North-Holland,1990, pp.1075–1155.
[27] P. Kolaitis. The Expressive Power of Stratified Logic Programs, Information and Computation, 90, 1990, pp.50–66.
[28] P. Kolaitis and C. Papadimitriou, Why not negation by fixpoint, Journal of Computer and System Science, 43,1991, pp.125–144.
[29] P.G. Kolaitis, M.Y. Vardi, On the expressive power of Datalog: Tools and a case study, in *ACM Proc. Symp. on Principles of Database Systems*,1990, pp.61-71.
[30] R. Krishnamurthy and S. Naqvi, Non-deterministic choice in Datalog, in *Proceedings of the 3rd International Conference on Data and Knowledge Bases*, 1988, pp. 416–424.
[31] J. W. Lloyd, *Foundations of Logic Programming*, Springer Verlag, Berlin, 1987.
[32] S. Naqvi and S. Tsur, *A logic language for data and knowledge bases*, Computer Science Press, New York,1989.
[33] C. Papadimitriou, A Note on the Expressive Power of Prolog, Bull. of the EATCS, 26, pp.21–23, 1985.
[34] C. Papadimitriou, *Computational Complexity*, Addison-Wesley, 1994.
[35] R. Ramakrishnan, D. Srivastava, and S. Sudanshan. CORAL — Control, Relations and Logic. in *Proceedings of 18th VLDB Conference*, 1992.
[36] D. Saccà and C. Zaniolo, Stable models and non-determinism in logic programs with negation. in *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, 1990, pp.205–217.
[37] Saccà D. and C. Zaniolo, Deterministic and Non-Deterministic Stable Models, submitted for publication, 1994.
[38] J. S. Schlipf, The expressive power of the logic programming semantics, in *Proceedings of the Ninth ACM Symposium on Principles of Database Systems*, 1990, pp.196–204.
[39] J. S. Schlipf, A Survey of Complexity and Undecidability Results in Logic Programming, *Proc. Workshop on Structural Complexity and Recursion-Theoretic Methods in Logic Programming*, 1993, pp.143–164.
[40] Y-H. Sheng, A Non-deterministics Deductive Database Language, in *Proceedings of the 1991 ACM-SIGMOD Int. Conference on Management of Data*, 1991, pp.188-198.
[41] J. D. Ullman, *Principles of Databases and Knowledge Base Systems*, Vol. I and II. Computer Science Press, 1988.
[42] A. Van Gelder, Negation as failure using tight derivations for general logic programs, Journal of Logic Programming, vol. 6, No. 1, 1989, pp.109–133.
[43] A. Van Gelder, K.A. Ross, and J.S. Schlipf, The well-founded semantics for general logic programs, Journal of ACM, 38(3), 1991, 620–650.
[44] M. Vardi. The Complexity of Relational Query Languages, in *Proceedings of the 14th ACM Symposium on Theory of Computing*, 1982, pp.137–146.
[45] Zaniolo, C., N. Arni, K. Ong, Negation and Aggregates in Recursive Rules: the $\mathcal{LDL}$++ Approach, in *Proc. 3rd Int. Conf. on Deductive and O-O DBs, DOOD-93*, Phoenix, AZ, Dec 6-8, 1993, pp. 104–221.