ELSEVIER

The Journal of Logic Programming 35 (1998) 79–101

THE JOURNAL OF LOGIC PROGRAMMING

Datalog with non-deterministic choice computes NDB-PTIME [1]

Fosca Giannotti [a,*], Dino Pedreschi [b,2]

[a] *CNR-Italian Nat. Research Council, CNUCE Institute, Via S. Maria 36, 56100 Pisa, Italy*
[b] *Dipartimento di Informatica, Univ. Pisa, Corso Italy 40, 56125 Pisa, Italy*

Received 1 May 1996; received in revised form 1 May 1997; accepted 6 June 1997

Abstract

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

1. Introduction

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

Indeed, both classes served as the basis of several extensions, aimed at enhancing the expressive power of the relational data model. For instance, the set of queries expressed by the relational algebra is strictly included in that of the *fixpoint queries* (the

* Corresponding author. Tel.: +39 50 593 339; fax: +39 50 904 052; e-mail: f.giannotti:@cnuce.cnr.it.

[1] This article is an extended, revised version of [1].

[2] E-mail: pedre@di.unipi.it.

S0743-1066/98/$19.00
PII: S0743-1066(97)10004-8

transitive closure is a fixpoint query which is inexpressible in the relational algebra), whereas it is well known that every fixpoint query can be expressed in FO extended with an inflationary fixpoint operator, or equivalently in Datalog extended with inflationary negation. Unfortunately, the expressiveness achieved by this kind of deterministic extensions of logic database languages is not satisfactory – e.g., the *parity* query is not a fixpoint query [2]. As a matter of fact, no known deterministic logic language expresses *exactly* the deterministic queries computable in polynomial time.

From a pragmatical viewpoint, a clear need for non-determinism is also emerging from applications. The *all-answers* paradigm for query execution exacerbates the need for special constructs to deal with situations where the user is not interested in all the possible answers. This problem is exemplified by the following situation: a new student must be given one (and only one) advisor. If the application of various qualification crieteria fails to narrow the search to a single qualified professor, then an arbitrary choice from the eligible faculty will have to be made and recorded. Another application of non-deterministic operators is in providing a logical basis for the notion of *object identity*, a crucial issue in the integration of deductive and object oriented databases [3]. Moreover, it has been pointed out in the literature that non-deterministic operators provide an explicit means for controlling the computation. Several examples illustrating this point are given in this paper. Explicit control mechanisms are often essential in real applications, in order to achieve efficient implementations. We refer to [4] for a comprehensive discussion on programming with non-determinism in deductive databases.

Several authors pointed out that a tight connection exists between non- determinism and ordered databases, from the point of view of the expressive power [5,6]. A fundamental result obtained independently by Immerman [7] and Vardi [8] is that the fixpoint queries over ordered databases coincide with the deterministic time-polynomial queries. On the other hand, certain non-deterministic query languages allow to obtain an (arbitrary) ordering of the database: it is therefore not surprising that such languages compute all deterministic time-polynomial queries, besides certain non-deterministic queries.

These are the motivations underlying the introduction of non-deterministic mechanisms logic database languages. A first batch of proposals is due to Abiteboul and Vianu [6,9–11], based on a non-deterministic *witness* construct for the fixpoint extensions of FO, and a non-deterministic operational semantics for Datalog¬, giving rise to the class of *N_Datalog* languages. The expressive power of these classes of proposals has been thoroughly studied by the same authors, who show how certain non-deterministic languages compute exactly the non-deterministic time-polynomial queries (NDB-PTIME) and the non-deterministic space-polynomial queries (NDB-PSPACE). These languages are described in operational terms, without a declarative semantics. Moreover, the proposals based on the witness construct are hardly amenable to efficient implementations, and therefore they do not suggest any construct which may be adopted in real database languages. This is however not the case for the N_Datalog languages, which basically correspond to the so-called production systems.

An alternative stream of proposals was started by Krishnamurthy and Naqvi [12], and later refined in [13,14]. These proposals are based on a non-deterministic *choice* construct for Datalog, which, in all cases, was designed on the basis of a declarative semantics – *choice models* in [12], and *stable models* in [13,14]. Moreover, the choice

construct can be efficiently implemented, and it is actually adopted in the logic database language (LDL) [23,24]. On the other hand, an expressiveness characterization for these proposals is lacking, which allows to compare the choice construct with the other proposals.

Fig. 1 highlights the taxonomy of non-deterministic logic languages. The dashed boxes indicate the mentioned two classes of proposals.

This work is aimed at bridging the existing gap between the two classes of proposals, by presenting an expressiveness characterization of Datalog augmented with one of the choice mechanisms, namely the *dynamic choice* construct introduced in [14]. This study is conducted both pragmatically, on the basis of examples, and formally, on the basis of known expressiveness results. In particular, we show how the dynamic choice construct is a powerful means for controlling the fixpont computation, in order to express relevant problems such as computing the complement of a relation, or computing an arbitrary ordering of a relation.

Finally, in the main result of this paper, we show that Datalog with dynamic choice expresses exactly the non-deterministic time-polynomial queries, a complexity class known as NDB-PTIME. The result is achieved by showing how the dynamic choice allows us to express the control needed to execute N_Datalog¬ programs over ordered domains – a language which is known to capture NDB-PTIME. The relevance of this result is clear: Datalog with the dynamic choice has the same (high) expressiveness of more complex languages. In fact, the languages used by Abiteboul and Vianu to capture NDB-PTIME use a combination of full FO logic (including negation) with fixpoint and non-determinism. On the contrary, Datalog with dynamic choice employs a negation-free fragment of the logic, and a combination of fixpoint and non-determinism.

We found it interesting that a rather simple language, equipped with a declarative interpretation, is capable of expressing all NDB-PTIME queries and, therefore, all *deterministic* ones. Such combined simplicity and expressiveness justifies the adoption of the language studied in this paper as the kernel of a realistic logic database language such as LDL.

It is worth noting that the result described in this paper has been already quoted on the book by Abiteboul et al. [15], where it forms material for Exercise 17.34. Also, it is the basis of the paper by Cheng and Zeng [16], where another language which captures NDB-PTIME is presented, based on a well-founded semantics for the choice construct.

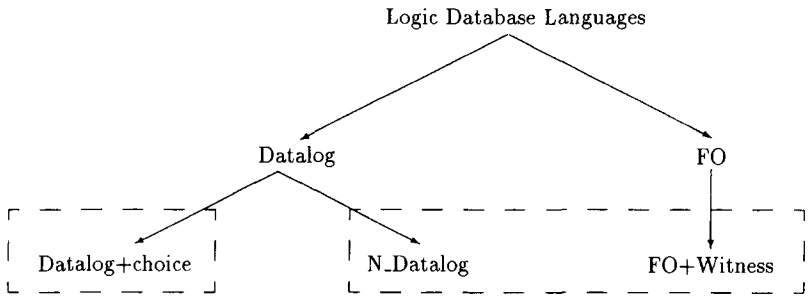

Fig. 1. Non-deterministic logic database languages.

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

1.1. Preliminaries

We assume that the reader is familiar with the relational data model and associated algebra, the relational calculus (i.e. the *first-order queries*, denoted FO), and Datalog [15,17–20]. In the extended language Datalog¬ the use of negation in the bodies of clauses (or *rules*) is also allowed; in another extension of Datalog, N_Datalog(¬), we shall also admit the presence of multiple atoms in the heads of clauses. Datalog(¬) (and N_Datalog(¬)) rules obey the *safety* constraint, i.e. each variable occurring in the head of a clause also occurs in a positive literal in the body. The operational semantics of Datalog, in the usual deterministic case, consists of evaluating "in parallel" all applicable instantiations of the rules. This is formalized using the consequences operator T_P associated to a Datalog program P, which is a monotonic map over (Herbrand) interpretations defined as follows:

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

The least model M_P of program P can then be computed as the limit (union) of the finite powers of T_P starting from an interpretation I, denoted $T_P \uparrow \omega(I)$ [21]:

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

The notation $T_P \uparrow \omega$ is used as an abbreviation of $T_P \uparrow \omega(\emptyset)$.

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

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

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

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

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

query Φ on I where S is assigned to J_k.

Notice that IFP converges in polynomial time on all input databases. A *partial* fixpoint operator (PFP) can also be defined, which gives raise to possibly infinite computations: PFP is not considered in this paper. The first-order logic augmented with IFP is called *inflationary fixpoint logic* and is denoted by FO + IFP. The queries computed by FO + IFP are the so-called *fixpoint queries*, for which various equivalent definitions exist in the literature [2,22].

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

Finally, the complexity measures are functions of the size of the input database. For Turing Machine complexity class C there is a corresponding complexity class of (non-deterministic) queries (N)DB-C. In particular, the class of (non-deterministic) database queries that can be computed by a (non-deterministic) Turing Machine in polynomial time is denoted by (N)DB-PTIME. No known deterministic language expresses all and only the queries in DB-PTIME.

2. Non-deterministic logic database languages

In this section, several mechanisms for dealing with non-determinism in logic database languages are briefly surveyed. In particular, we present a non-deterministic construct for the fixpoint extensions of FO, a non-deterministic operational semantics for Datalog¬ (*à la production systems*), and a non-deterministic mechanism for pure Datalog. The first two classes of proposals are due to Abiteboul and Vianu [6,11], whereas the third class of proposals is due to Krishnamurthy and Naqvi [12] and Giannotti et al. [13,14].

2.1. The witness operator

A non-deterministic extension of FO is achieved by introducing the so- called *witness* operator [6,11]. Informally, given a formula (query) $\Phi(X)$, the witness operator W_X applied to $\Phi(X)$ chooses an arbitrary X that makes Φ true. The extension of the inflationary fixpoint logic FO + IFP with the witness operator is denoted by FO + IFP + W.

Let us define more precisely the semantics of W. Notice that, in presence of non-determinism, we have a *set* of possible interpretations for a given formula in FO + IFP + W, or equivalently, a set of possible sets of answers to a given query. Consider a formula $W_X(\Phi(X, Y))$, where Y is the vector of variables other than X that occur free in Φ. Then I is an interpretation of $W_X(\Phi(X, Y))$ iff, for some interpretation J of $\Phi(X, Y)$ such that $I \subseteq J$:

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

Intuitively, one " witness" X_Y is arbitrarily chosen for each Y satisfying $\exists X \cdot \Phi(X, Y)$. 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 2.1. 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 which assigns exactly one advisor to each student.

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

Form the viewpoint of the expressive power, the relevance of FO + IFP + W is due to the following result of Abiteboul and Vianu [11].

Theorem 2.1. *A query is in* NDB-PTIME *iff it is expressed in* FO + IFP + W.

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

2.2. N_Datalog

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

Abeiteboul and Vianu [11] proposed to adopt the mentioned non-deterministic operational semantics for N_Datalog¬, an extension of pure Datalog which allows the use of negation in clause bodies, and multiple atoms in clause heads. Thus, an N_Datalog program is a finite set of rules of the form

$$A_1,\ldots,A_k \leftarrow L_1,\ldots,L_m \quad (k \geqslant 1, m \geqslant 0),$$

where each A_j is an atom and each L_i is a literal, i.e. an atom or its negation.

To define the non-deterministic operational semantics, the notion of *immediate successor* of an interpretation (i.e. a set of facts) I w.r.t. a rule r is introduced. Let

$$r' = A_1,\ldots,A_k \leftarrow L_1,\ldots,L_m$$

be a ground instance of an N_Datalog¬ rule r such that all literals $L_1,\ldots,L_m$ in the body of r' are true in I. Then the interpretation $J = I \cup \{A_1,\ldots,A_k\}$ is called an *immediate successor of I using r*. We then define a computation of an N_Datalog¬ program P starting from an initial interpretation I_0 as a sequence $I_0,\ldots,I_n,\ldots$ of interpretations such that, for $k \geqslant 0, I_{k+1}$ is an immediate successor of I_k using some rule from P. A *model* of P is the last interpretation of a maximal computation.

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

Example 2.2. The following Datalog¬ program takes as input a binary relation G representing an undirected graph g, and computes (into the relation DG) an arbitrary orientation of g:

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

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

Theorem 2.2. *A query is in* NDB-PTIME *iff it is expressed in N_Datalog¬∀ or, equivalently, in N_Datalog¬ over ordered databases.*

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

2.3. The family of choice operators

Another approach to non-determinism in logic database languages was started by Krishnamurthy and Naqvi [12], and later refined by Saccà and Zaniolo [13] and Giannotti [14]. The proposals described in this section are based on a non-deterministic *choice* construct for Datalog, which, in all cases, was designed on the basis of a declarative semantics – *choice models* in [12], and *stable models* in [13,14]. Moreover, the choice construct can be efficiently implemented, and it is actually adopted in the logic database language $\mathscr{LDL}$ [23,24]. On the other hand, an expressiveness characterization for these proposals is lacking, which allows to compare the choice construct with the previously discussed proposals. The rest of this section surveys the original proposal and two refinements, which improve from several viewpoints.

2.3.1. Static choice

The choice construct was first proposed by Krishnamurthy and Naqvi [12]. 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 2.3. 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 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 $M_1, \ldots, M_4$ can be constructed by first removing the choice goal from the rule and computing the resulting *a_st* facts. Then the basic operation of enforcing the FD constraints is performed, by selecting a maximal subset of the previous *a_st* facts that satisfies the FD $Crs \rightarrow St$ (there are four such subsets).

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

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

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

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

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

Thus, computing with the static choice entails three stages of a bottom-up procedure. In the first stage, the saturation of $\mathrm{PV}(P)$ 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.

An interesting application of static choice is in the logical reconstruction of the notion of object identify of object-oriented database languages [3]. Suppose we want to associate to each instance of a relation P a unique object identifier from a domain *ID* of identifiers. According to the described operational semantics of choice, the following choice rule performs this task:

$$OID\_P(X, Y) \leftarrow P(X), ID(Y), choice((X), (Y)), choice((Y), (X)).$$

Notice that it is crucial that the two FD $X \rightarrow Y$ and $Y \rightarrow X$ are maximized simultaneously. For this reason, such a specification cannot be achieved using the witness operator, which maximizes only one FD at a time.

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 cannot be safely used within recursive rules. As observed in [14], the choice models semantics fails when mixed with recursion, in the sense that the delivered results do not comply with any declarative reading. Moreover, the procedure for computing choice models is extremely inefficient as operating the choices only after a general saturation phase is wasteful – a more efficient procedure should instead operate choices as soon as possible, in order to reduce the amount of work for future saturations. Finally, due to the impossibility of being adopted within recursion, the static choice has a limited expressive power. To remedy these drawbacks, some refinements of the static choice have been proposed, which are discussed next.

2.3.2. *Model-theoretical choice*

An alternative definition of a declarative semantics for the choice construct was proposed by Saccà and Zaniolo [13]. According to this proposal, programs with choice are transformed into programs with negation which exhibit a multiplicity of stable models. [3] Each stable model corresponds to an alternative set of answers for the original program. Following [13], therefore, given a choice program P, we introduce the *stable version* of P, denoted by $SV(P)$, defined as the program with negation obtained from P by the following two transformation steps:

1. Consider a choice rule of P, say

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

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

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

2. Add the new rule

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

3. Add the new rule

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

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

The transformation directly generalizes to FD involving vectors of variables, and to multiple choice goals. When the given program P is such that none of its choice rules is recursive, then P and its stable version are semantically equivalent in the

[3] Stable models semantics is a concept originating from autoepistemic logic, which was applied to the study of negation in Horn clause languages by Gelfond and Lifschitz [25].

sense that the set of choice models of P coincides with the set of stable models of SV(P) on common predicate symbols [13].

Example 2.4. The following is the stable version of Example 2.3.

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

This programs admits four distinct stable models, corresponding to the four choice models of Example 2.3.

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

This new characterization of choice overcomes the cited deficiencies of static choice of Krishnamurthy and Naqvi [12]. Indeed, the new formulation correctly supports the use of choice within recursive rules, avoiding the semantical anomalies of the static choice [14]. Moreover, it can be efficiently implemented by a straightforward fixpoint procedure which allows to interleave non-deterministic choices and ordinary rule applications in the bottom-up computation (the so-called *stable backtracking fixpoint* [13]). Nevertheless, the expressiveness of this form of choice can be considerably enhanced by adopting a particular instance of the cited fixpoint procedure.

2.3.3. Dynamic choice

We now introduce a particular operational semantics for the choice construct, following the presentation of Giannotti et al. [14]. This operational semantics is an instance of the general bottom-up procedure of Saccà and Zaniolo [13] for computing stable models, and is obtained by adopting a particular policy of interleaving non-deterministic choices and the ordinary fixpoint computation. The resulting procedure is referred to as DCF for *dynamic choice fixpoint*, and the associated form of choice construct is referred to as *dynamic choice*.

The DCF procedure, and thus the dynamic choice construct, reflects the institution that choices should be operated as soon as possible during the fixpoint computation. This design principle has two relevant consequences. First, a higher degree of efficiency is achieved, as early choices have the effect of reducing the number of inferred facts at the intermediate stages of the fixpoint computation, and possibly of anticipating its termination. Second, a higher degree of expressiveness is achieved: the next sections of this paper are devoted to this point. For instance, we will show how the dynamic choice construct is expressive enough to capture various forms of negation for Datalog. It is worth noting that we do not prove in this paper that dynamic choice is *strictly* more expressive than model-theoretical choice. However, we anticipate that this result can be easily established by showing that model-theoretical choice computes only *monotonic* queries, and therefore cannot express negation.

Informally, the DCF procedure behaves as follows. Given a choice program P and its stable version SV(P), call **C** the set of *chosen* rules in SV(P), **D** the set of *diffChoice* rules in SV(P), and **O** the set of the remaining (original) rules in SV(P). Then, the DCF procedure operates as follows.

1. Find the fixpoint of the **O** part.
2. While there exists an enabled ground instance r of a *chosen* rule in **C**, repeat: (a) execute r; (b) execute all rules in **D** enabled by r.
3. Repeat steps 1 and 2 until no rule is enabled.

Notice that we used the term "execute" to mean the ordinary bottom-up computation mechanism of asserting the head of a rule whenever its body is true. The idea underlying the DCF procedure can be explained as follows. There are two modes of operation: a saturation mode and a choice mode. In the saturation mode, the consequences of the original rules are computed by an ordinary fixpoint mechanism. When nothing more can be deduced, the procedure switches to the choice mode. In the choice mode, a *chosen* rule together with the associate *diffChoice* rules are executed, until no more choices can be made. Then the procedure switches to the saturation mode again, and the process continues until a fixpoint is reached. Notice that the execution the *diffchoice* rules shrinks the set of enabled *choice* rules.

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

The DCF procedure is correct w.r.t. the stable model semantics, in the sense that the result of DCF is a stable model of the program SV(P). However, DCF cannot compute an arbitrary stable model, but only some *preferred* ones, due to the particular policy for operating choices. Therefore, dynamic choice is sound, although not complete, w.r.t. stable model semantics [14]. From now on, we shall use the term *choice model* (of a choice program) to denote one of the possible outcomes of the DCF procedure, since we restrict our attention to the dynamic choice construct.

2.3.4. A fixpoint characterization of dynamic choice

An alternative, more declarative definition of the DCF procedure can be given in terms of a non-deterministic immediate consequence operator Ψ_P associated to a choice program P. This will provide the basis of a fixpoint semantics for the dynamic choice, which will be convenient when discussing the expressive power of the dynamic choice in the next sections.

To this purpose, we adopt the transformation of a choice program P, defined by Krishnamurthy and Naqvi [12], which replaces the choice goals from the rules with new predicate symbols. A choice rule R from P:

$$R: H \leftarrow \mathbf{B}, \mathbf{C},$$

where **C** are the choice goals and **B** the other goals, is transformed into the two rules:

$$H \leftarrow \mathbf{B}, chosen_R(Y), \qquad chosen_R(Y) \leftarrow \mathbf{B},$$

where Y are the variables occurring in **C**. We denote by P_{choice} the set of rules of the transformed program whose heads are $chosen_R$ relations, and P_{original} the remaining rules of the transformed program. From now on, we deliberately use the term choice program to denote both a choice program and its transformed version, the different role being clear from the context. Also, when we refer to an interpretation of a choice

program, we actually mean an interpretation of its transformed version, containing the $chosen_R$ atoms. Given a choice program P and an interpretation I, we write $I \models \mathrm{FD}_P$ if, for any choice rule R of P, the set of $chosen_R$ atoms of I satisfies the FD constraint specified by rule R.

Definition 2.1. 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:

$$\Psi_P(I) = \Big\{ M_I \cup I_1 \mid M_I = T_{P_{\text{original}}} \uparrow \omega(I), \text{ and. } I_1 \text{ is a maximal subset of } T_{P_{\text{choice}}}(M_I) \text{ such that } I_1 \cup I \models \mathrm{FD}_P \Big\}.$$

Intuitively, the Ψ_P operator formalizes the mapping associated to an iteration of the outer loop of the DCF procedure. Observe that, for any interpretation I:

$$I \models \mathrm{FD}_P \quad \text{iff} \quad \Psi_P(I) \neq \emptyset, \tag{2.1}$$

$$I \models \mathrm{FD}_P \quad \text{implies} \quad J \models \mathrm{FD}_P \text{ for } J \in \Psi_P(I). \tag{2.2}$$

It is legitimate to ask ourselves whether Ψ_P is a monotonic operator, in the sense that if $I \subseteq J$ then for all $I' \in \Psi_P(I)$ there exists $J' \in \Psi_P(J)$ such that $I' \subseteq J'$. The answer to this question is no: it suffices to consider I, J such that $I \subseteq J$, $I \models \mathrm{FD}_P$ and $J \not\models \mathrm{FD}_P$, and observe that, by Eq. (2.1), $\Psi_P(J)$ is empty and $\Psi_P(I)$ is not. However, we show next that the above form of monotonicity holds when considering only interpretations which satisfy FD_P. This result will provide the basis for a fixpoint procedure adopting the Ψ_P operator.

Next, we define the powers of the Ψ_P operator starting from the empty interpretation, as follows:

$$\Psi_P \uparrow 0 = \{\emptyset\},$$

$$\Psi_P \uparrow (n+1) = \bigcup_{I \in \Psi_P \uparrow n} \Psi_P(I),$$

$$\Psi_P \uparrow \omega = \bigsqcup_{n \geqslant 0} \Psi_P \uparrow n.$$

The following proposition points out the relevant properties of the powers of the Ψ_P operator.

Proposition 2.1. *Let P be a choice program. Then, for all $n \geqslant 0$:*
- *for any $I \in \Psi_P \uparrow n$, $I \models \mathrm{FD}_P$;*
- *for any $I \in \Psi_P \uparrow n$, there exists $J \in \Psi_P \uparrow (n+1)$ such that $I \subseteq J$.*

Proof. The first property follows from Eq. (2.2) and the fact that $\emptyset \models \mathrm{FD}_P$. To prove the second property, we show the following one, which, together with Eq. (2.1), implies the thesis. For all interpretations I, J:

$$I \in \Psi_P \uparrow n, J \in \Psi_P(I) \text{ implies } I \subseteq J.$$

The proof is by induction on n. In the case $n = 0$ the proof is trivial. In the induction case, the induction hypothesis is the following, for all interpretations I, J:

$$I \in \Psi_P \uparrow (n-1), J \in \Psi_P(I) \text{ implies } I \subseteq J.$$

Our aim is to prove, for any interpretation K:

$$K \in \Psi_P(J) \text{ implies } J \subseteq K.$$

By Definition 2.1, $J = M_I \cup I_1$, where $M_I = T_{P_{\text{original}}} \uparrow \omega(I)$, and $I_1 \subseteq T_{P_{\text{choice}}}(M_I)$.

Again, by Definition 2.1, $K = M_J \cup J_1$, where $M_J = T_{P_{\text{original}}} \uparrow \omega(J)$, and $J_1 \subseteq T_{P_{\text{choice}}}(M_J)$.

By the induction hypothesis, we have $I \subseteq J$, which implies $T_{P_{\text{original}}} \uparrow \omega(I) \subseteq T_{P_{\text{original}}} \uparrow \omega(J)$ by the monotonicity of $T_{P_{\text{original}}}$. Hence, we obtain:

$$M_I \subseteq M_J. \tag{2.3}$$

Moreover, from Eq. (2.3) and the monotonicity of $T_{P_{\text{choice}}}$ we obtain:

$$T_{P_{\text{choice}}}(M_I) \subseteq T_{P_{\text{choice}}}(M_J). \tag{2.4}$$

By Definition 2.1, I_1 and J_1 are maximal subsets of $T_{P_{\text{choice}}}(M_I)$ and $T_{P_{\text{choice}}}(M_J)$, respectively, which satisfy the FD's. Hence, we obtain:

$$I_1 \subseteq J_1 \tag{2.5}$$

from Eq. (2.4). Finally, we obtain the thesis from Eqs. (2.3) and (2.5).

The above proposition ensures us that $\Psi_P \uparrow \omega$ is the limit of the powers of Ψ_P, and justifies the fact that the dynamic choice fixpoint is inflationary.

Finally, the following result relates the limit of the Ψ_P operator with the output of the DCF procedure. The proof is lengthy and uninteresting, and therefore omitted.

Proposition 2.2. *Let P be a choice program. Then M is a choice model of P iff $M \in \Psi_P|\omega$.*

The main interest for the dynamic choice construct lies in the fact that it is highly expressive – it allows to compute efficiently some relevant queries, such as negation, and ordering. These and other issues related to the expressive power of the dynamic choice are addressed in the rest of this paper.

3. Computing negation with the choice operator

We recall here an example taken from [14]: the realization of a form of negation, which can be used to model stratified and inflationary negation for Datalog. The following choice program defines relation NOT_P as the complement of a relation P with respect to a universal relation U. We assume here that both P and U are extensional relations, although this constraint will be soon relaxed.

Definition 3.1. The choice program $NOT[P, U]$ consists of the following rules:

R_1: $NOT\_P(X) \leftarrow COMP\_P(X, 1)$.
R_2: $COMP\_P(X, I) \leftarrow TAG\_P(X, I),\ choice((X), (I))$.
R_3: $TAG\_P(nil, 0)$.
R_4: $TAG\_P(X, 0) \leftarrow P(X)$.
R_5: $TAG\_P(X, 1) \leftarrow U(X), COMP\_P(-, 0)$.

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

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

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

The following result states the correctness of the program of Definition 3.1.

Proposition 3.1. *Let P and U be n-ary EDB relations. Then program $NOT[P, U]$ has a unique choice model M_{NOT}, and $M_{NOT} \models NOT\_P(x)$ iff $\models U(x) \wedge \neg P(x)$.*

Proof. It is sufficient to prove, for any choice model M_{NOT}:

$$M_{NOT} \models NOT\_P(x) \text{ iff } \models U(x) \wedge \neg P(x)$$

since this implies the uniqueness of M_{NOT}.

If part. We calculate:

$\models U(x) \wedge \neg P(x)$

$\Rightarrow$ { By rules: R_2, R_3, R_4, R_5 of Definition 3.1}

$M_{NOT} \not\models TAG\_P(x, 0) \wedge M_{NOT} \models TAG\_P(x, 1)$

$\Rightarrow$ { By rule: R_2 of Definition 3.1}

$M_{NOT} \models COMP\_P(x, 1)$.

$\Rightarrow$ { By rule: R_1 of Definition 3.1}

$M_{NOT} \models NOT\_P(x, 1)$.

Only-if part. If $M_{NOT} \models NOT\_P(x)$, we have that $M_{NOT} \models chosen_{R_2}(x, 1)$, since the rules R_1 and R_2 are needed to derive $NOT\_P(x)$. Assume, by contradiction, that $\models P(x)$. This implies that $M_{NOT} \models chosen_{R_2}(x, 0)$, since rule R_4 derives $TAG\_P(x, 0)$. Thus, $M_{NOT} \models chosen_{R_2}(x, 0) \wedge chosen_{R_2}(x, 1)$, which violates the FD's. □

Essentially, this example shows how the dynamic choice offers a flexible mechanism for handling the control needed to emulate the difference between two relations. It is shown in [26] that the above program can be refined in order to realize more powerful forms of negation, such as stratified and inflationary negation. This goal is achieved by suitably emulating the extra control needed to handle program strata and fixpoint approximations, respectively.

4. Ordering with the choice operator

It has been pointed out in the literature that a tight connection exists between non-determinism and ordered databases [5,6]. On one hand, consider the case that a query Q relies on the ordering in which elements are stored in the database: when abstracted at the conceptual level, where physical details are unrelevant, Q exhibits a non-deterministic behavior. On the other hand, it is often possible to emulate ordering using non-deterministic mechanisms.

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

Definition 4.1. The choice program $ORD[U]$ consists of the following rules:

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

R_2: $SUCC(X, Y) \leftarrow SUCC(-, X), U(Y), SUCC(min, Z),\ X \neq Y,$
$Y \neq Z,\ choice((X), (Y)),\ choice((Y), (X))$.

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

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

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

The correctness of the program of Definition 4.1 is stated by the following Proposition.

Proposition 4.1. *Let U be an EDB relation. Then, in any choice model M_{ORD} of program $ORD[U]$, the (transitive closure of the) relation SUCC is an irreflexive total ordering over U.*

Proof. We prove equivalently that given the restriction M of any choice model to the *SUCC* facts, we have:

$$M = \{SUCC(min, y_1), SUCC(y_1, y_2), \ldots, SUCC(y_{k-1}, y_k)\},$$

where $\{y_1, \ldots, y_2\}$ are the tuples in the extension of the relation U.

The thesis is directly implied by the fact that, for $i \geqslant 2$, $M_i \in \Psi_{ORD} \uparrow i$ iff

$$\begin{aligned} M_i = {} & \{U(y_1), \ldots, U(y_k)\} \cup \{chosen_{R_1}(y_1)\} \\ & \cup \{SUCC(min, y_1), SUCC(y_1, y_2), \ldots, SUCC(y_{i-2}, y_{i-1})\} \\ & \cup \{chosen_{R_2}(y_1, y_2), \ldots, chosen_{R_2}(y_{i-2}, y_{i-1}, chosen_{R_2}(y_{i-1}, y_i)\}. \end{aligned}$$

In fact, we have $M = M_k$. The proof of the above assertion proceeds by induction on i.

Base case. For $i = 2$, it is readily checked that:

$$M_2 = \{U(y_1), \ldots, U(y_k)\} \cup \{chosen_{R_1}(y_1)\} \cup \{SUCC(min, y_1)\} \cup \{chosen_{R_2}(y_1, y_2)\}.$$

Induction case. Observe that, for $i > 2$, we can define:

$$M_{i+1} = M_i \cup \{SUCC(y_{i-1}, y_i), chosen_{R_2}(y_i, y_{i+1})\}.$$

To prove this property, it suffices to show that only one fact $chosen_{R_2}(y_i, y_{i+1})$ can be added to M_i, and that y_{i+1} is different from all previous choices $y_1, \ldots y_i$. This is a consequence of the following observations:

- the constraint $X \neq Y$ in R_2 implies that $y_{i+1} \neq y_i$;
- the constraint $X \neq Z$ in R_2 implies that $y_{i+1} \neq y_1$;
- the FD $Y \rightarrow X$ in R_2 and the induction hypothesis imply that y_i does not occur as a first argument in a *SUCC* fact of M_i;
- the FD $X \rightarrow Y$ in R_2 and the induction hypothesis imply that y_{i+1} does not occur as a second argument in a *SUCC* fact of M_i. □

This application brings further evidence to the effectiveness of the dynamic choice as a control mechanism. It also suggests that the dynamic choice is highly expressive, as languages over ordered domains are known to be strictly more expressive than languages over unordered domains [5]. Indeed, the fact that dynamic choice can express ordering is essential in the proof of the main result of this paper.

5. Emulating N_Datalog with the choice operator

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

Definition 5.1. Let P be a N_Datalog¬ program. Let δ be the finite set of distinct constants occurring in P, and L the cardinality of δ. Let l be the number of distinct relations of P, and l_1 be the maximal arity of the relations in P. [4] Given an array V of distinct variables $\{V_1, \ldots, V_{l*l_1}\}$, we assume that all variables occurring in the head of a rule in P are renamed using variables from this set.

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

1. Add the following facts:
   LEVEL(*min*).
   $UNIV(a_1, \ldots, a_{l_1+1})$.
   for $a_j \in \delta, j = 1, \ldots, l_1 + 1$, together with the rules of program $ORD[UNIV]$ as in Definition 5.1. Here, min is an array (of arity $l_1 + 1$) of new constants.
2. Add the following program defining the complement of a relation P with respect to another relation U. Such program extends that of Definition 5.1 to deal with

[4] Notice that L^{l_1+1} is an upper bound for the number of instances which are derivable from P, under the natural assumption that $L \geqslant l$.

the level of the fixpoint iteration. The notation $NOT[P,U](x,n)$ is used in the following to refer to the following program.

$NOT\_P(X,N) \leftarrow COMP\_P(X,1,N).$
$COMP\_P(X,I,N) \leftarrow TAG\_P(X,I,N), choice((X,N),(I)).$
$TAG\_P(nil,0,N) \leftarrow LEVEL(N).$
$TAG\_P(X,0,N) \leftarrow P(X), LEVEL(N).$
$TAG\_P(X,I,N) \leftarrow U(X), COMP\_P(\_,0,N).$

where *nil* is a new constant.

3. For each rule R_i of P:

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

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

$$\begin{aligned} NEW(U,N,i) \leftarrow\ & P_1(Y_1),\ldots,P_k(Y_k),\\ & NOT[Q_1,UNIV\_Q_1](Z_1,N),\ldots,\\ & NOT[Q_h,UNIV\_Q_h](Z_h,N),\\ & NOT[A_0,UNIV\_A_0](X_0,N).\\ \vdots\\ NEW(U,N,i) \leftarrow\ & P_1(Y_1),\ldots,P_k(Y_k),\\ & NOT[Q_1,UNIV\_Q_1](Z_1,N),\ldots,\\ & NOT[Q_h,UNIV\_Q_h](Z_h,N),\\ & NOT[A_m,UNIV\_A_m](X_m,N). \end{aligned}$$

Here, i is a constant identifying the rule R_i, and U is an array of arity $l * l_1$, which contains the variables in the head of the rule R_i; all the extra arguments of U are filled in with a new constant ∂. Moreover, $UNIV\_Q$, for any relation Q of P, is defined by the set of facts $UNIV\_Q(a_1,\ldots,a_p)$ for $a_i \in \delta$ $(1 \leqslant i \leqslant p)$, where p is the arity of Q.

4. Add the two rules:

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

where I is a variable denoting a generic rule from P.

5. Replace rule R_i with the following m rules:

$$\begin{aligned} & A_0(X_0) \leftarrow INSTANCE(V,N,i).\\ & \vdots\\ & A_m(X_m) \leftarrow INSTANCE(V,N,i). \end{aligned}$$

Before analyzing the transformation of Definition 5.1, let us recall the behaviour of the non-deterministic semantics: at each fixpoint iteration a new instance is computed by choosing a unique rule chosen from the applicable ones. The relation $NEW(\_,\_,i)$ collects the new instances which can be derived at the ith stage of the computation using the rule R_i. In fact, the relation $NOT[A_j,UNIV\_A_j]$ used in the body of the rules for *NEW* ensures that instances for the relation A_j occurring in the head have not been computed yet (Definition 5.1(3)). The relation *INSTANCE* collects the selected instance of the selected rule (Definition 5.1(4)). From an *INSTANCE* fact it is possible to infer the instances of the relations in the head of

the selected rule (Definition 5.1(5)), and the next stage of the computation (*LEVEL*, Definition 5.1(4)), which will fire the rules for the *NOT_Q* relations. This yields the set of negative facts for the next stage of the computation (Definition 5.1(2)). At this stage, a new value for *LEVEL* is inferred, which will fire the rules of the relations *NOT_Q*. Initially, the fact *LEVEL*(*min*) (Definition 5.1(1)) triggers the first computation of the relations *NOT_Q*. Finally, observe that the definition of the relation *UNIV* (Definition 5.1(1)) ensures that its cardinality is an upper bound for the length of any N_Datalog¬ computation of P.

Example 5.1. We illustrate the transformation of Definition 5.1 on a simple N_Datalog¬ program:

R_1: $P(X), Q(Y) \leftarrow \neg R(a), S(X), T(Y).$
R_2: $S(X) \leftarrow T(X).$
R_3: $T(b).$

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

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

The rest of this section is devoted to the proof that the transformed choice program correctly simulates the original N_Datalog¬ program. The proof is based on a bisimulation relation between the two programs: a correspondence is established between a step of the computation of the N_Datalog¬ program P and a step of the dynamic choice fixpoint computation of the transformed program P'. The key concept is the notion of *counterpart*: intuitively, an interpretation of M of P' is a counterpart of an interpretation I of P if M contains the facts needed to describe the computation of P which yields I, in terms of the auxiliary relations *LEVEL*, *INSTANCE*, etc., introduced in Definition 5.1. Essentially, the proof shows that the property of being a counterpart is preserved during the computation of either programs.

Remark 5.1. Consider a choice program P' obtained with the transformation of Definition 5.1. As a consequence of Proposition 4.1, in any choice model of P' the extension of relation *SUCC* is isomorphic to an initial fragment of the natural numbers. Therefore, to simplify the notation, we are allowed to adopt numbers to denote the elements of the relations *UNIV* and *LEVEL*. Accordingly, if $SUCC(a,b)$

holds and a is denoted by n, we denote b by $n+1$. This convention greatly simplifies the treatment of level indicators.

Definition 5.2. Let P be a N_Datalog$\neg$ program, and P' the choice program obtained from P with the transformation of Definition 5.1. Let $I_0, \ldots, I_n$ be a computation of P, and M an interpretation for P'. M is a *counterpart* of I_n if:

$$M \models Q(x) \text{ iff } I_n \models Q(x) \tag{5.1}$$

for any relation Q of P.

$$M \models NOT\_Q(x,k) \text{ iff } I_k \not\models Q(x) \quad (0 \leqslant k \leqslant n) \tag{5.2}$$

for any relation Q of P.

$$M \models LEVEL(k) \quad (0 \leqslant k \leqslant n) \tag{5.3}$$

$$M \models NEW(v,k,i) \quad (0 \leqslant k \leqslant n) \tag{5.4}$$

iff there exists an instance r of rule R_i, $r: A_0, \ldots, A_m \leftarrow L_1, \ldots, L_h$ such that $I_k \models L_1, \ldots, L_h$ and $I_k \not\models A_1, \ldots, A_m$, and the variables in the head of R_i are instantiated according to v

$$M \models INSTANCE(v,k,i) \quad (0 \leqslant k \leqslant n-1) \tag{5.5}$$

iff I_{k+1} is an immediate successor of I_k using an instance of rule R_i such that the variables in the head of R_i are instantiated according to v.

Proposition 5.1. *Let P be a N_Datalog$\neg$ program and P' the choice program obtained from P with the transformation of Definition 5.1. Let $I_0, \ldots, I_n$ be an computation of P, and M an interpretation for P' such that M is a counterpart of I_n. Then:*

1. *if J is an immediate successor of I_n, then there exists a counterpart N of J such that $N \in \Psi_{P'} \uparrow 3(M)$;*
2. *if $N \in \Psi_{P'} \uparrow 3(M)$, then there exists an immediate successor J of I_n such that N is a counterpart of J.*

Proof. The proof of statement 1 proceeds by case analysis.

Case 1.1 ($J = I_n$): In this case, no rule of P is applicable in I_n, i.e., for any instance

$$A_0, \ldots, A_m \leftarrow L_1, \ldots, L_h$$

of a rule from P, either $I_n \not\models L_1, \ldots, L_h$ or $I_n \models A_1, \ldots, A_m$. Therefore, by Definition 5.2 (4), $M \not\models NEW(z,n,i)$ for any z, i. By Definition 5.1, no new *INSTANCE*, *LEVEL* and *NOT_Q* facts can be derived from P', and hence if $N \in \Psi_{P'} \uparrow 3(M)$, then N coincides with M on all relations, possibly except *SUCC*. As a conclusion, N is a counterpart of $J = I_n$, which implies the thesis.

Case 1.2 ($J \supset I_n$): In this case, J is an immediate successor of I_n via some instance

$$A_0(x_0), \ldots, A_m(x_m) \leftarrow P_1(y_1), \ldots, P_k(y_k), \neg Q_1(z_1), \ldots, \neg Q_h(z_h)$$

of a rule R_i from P. By the definition of successor we have:

$$I_n \models P_1(y_1) \wedge \cdots \wedge P_k(y_k),$$

$$I_n \not\models Q_1(z_1) \vee \cdots \vee Q_h(z_h),$$

$$I_n \not\models A_0(x_0) \wedge \cdots \wedge A_m(x_m).$$

This, together with the fact that M is a counterpart of I_n, implies:

$$M \models P_1(y_1) \wedge \cdots \wedge P_k(y_k),$$

$$M \models NOT\_Q_1(z_1, n) \wedge \cdots \wedge NOT\_Q_h(z_h, n),$$

$$M \models NOT\_A_0(x_0, n) \vee \cdots \vee NOT\_A_m(x_m, n).$$

By Definition 5.1(3) we get:

$$M \models NEW(u, n, i).$$

By Definition 5.1 and Definition 5.1(4), there exists $N_1 \in \Psi_{P'}(M)$ such that:

$$N_1 \models INSTANCE(v, n, i) \tag{5.6}$$

corresponding to the choice represented by $chosen(n, v, i)$. The FD constraint guarantees that $N_1 \not\models INSTANCE(v', n, i')$ for any $v' \neq v$ or $i' \neq i$.

By Eq. (5.6) and Definition 5.1(4), we get:

$$N_1 \models LEVEL(n + 1). \tag{5.7}$$

By Eq. (5.6) and Definition 5.1(5) we get:

$$N_1 \models A_0(x_0) \wedge \cdots \wedge A_m(x_m)$$

which implies

$$N_1 \models Q(x) \text{ iff } J \models Q(x) \tag{5.8}$$

for any relation Q of P, since $J = I_n \cup \{A_0(x_0), \ldots, A_m(x_m)\}$.

By Proposition 3.1 and Definition 5.1(2), there exists $N \in \Psi_{P'} \uparrow 2(N_1)$ such that:

$$N \models NOT\_Q(x, n + 1) \text{ iff } N_1 \not\models Q(x)$$

which implies, by Eq. (5.8):

$$N \models NOT\_Q(x, n + 1) \text{ iff } J \not\models Q(x). \tag{5.9}$$

By Eqs. (5.8) and (5.9) and Definition 5.1(3) we obtain:

$$N \models NEW(v, n + 1, i) \tag{5.10}$$

for any instance of a rule R_i from P which is applicable in J.

As a conclusion, we obtain from Eqs. (5.6)–(5.10) that N is a counterpart of J.

The proof of statement 2 is analogous to that of statement 1, and therefore omitted. □

We are now in the position of proving that the transformation of Definition 5.1 correctly simulates the non-deterministic semantics of N_Datalog¬.

Theorem 5.1. *Let P be a N_Datalog¬ program, and P' the choice program obtained from P with the transformation of Definition 5.1. Then:*

- *for any model I of P there is a choice model M of P' such that M is a counterpart of I;*
- *for any choice model M of P' there is a model I of P such that M is a counterpart of I.*

Proof. We prove the following two assertions, which clearly imply the thesis:

1. for any computation $I_0 = \emptyset, \ldots, I_n$ of P there exists $M \in \Psi_{P'} \uparrow (3n+2)$ such that M is a counterpart of I_n;
2. for any $M \in \Psi_{P'} \uparrow (3n+2)$ there exists a computation $I_0 = \emptyset, \ldots, I_n$ of P such that M is a counterpart of I_n.

The proof of assertion 1 is by induction on the length of the computation.

Base case. With $n = 0$, $I_n = \emptyset$, and, by Definition 5.1(1) and (2) and Proposition 3.1, any interpretation M in $\Psi^{P'} \uparrow 2$ is such that $M \models NOT\_Q(x, 0)$ for any atom $Q(x)$. This also implies that $M \models NEW(v, 0, i)$ for all rules R_i of P which are facts. Therefore, M is a counterpart of I_0.

Induction case. It is a direct consequence of Proposition 5.1(1).

The proof of assertion 2 is analogous to that of assertion 1, and therefore omitted. □

6. Datalog with dynamic choice computes *NDB_PTIME*

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

Theorem 6.1. *A query is in NDB_PTIME iff it is expressed in Datalog with dynamic choice.*

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

- Datalog with dynamic choice emulates N_Datalog¬ (Theorem 5.1),
- Datalog with dynamic choice expresses ordering (Proposition 4.1), and
- N_Datalog¬ over ordered domains expresses NDB-PTIME (Theorem 2.2).

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

Theorem 6.1 defines precisely the expressive power of Datalog augmented with the dynamic choice construct. As a consequence, we obtain that such a language embodies a simple, declarative and efficiently implementable characterization of NDB-PTIME. The previous results, discussed in Section 2, adopted a combination of three mechanisms, namely fixpoint, non-determinism and negation, in order to compute all (and only) the time-polynomial queries. Theorem 6.1 improves on these results, in that it implies that a suitable combination of only *two* mechanisms, fixpoint and non-determinism, is enough to the purpose of computing NDB-PTIME.

From a more pragmatical viewpoint, these results indicate that dynamic choice is a flexible mechanism for explicitly handling the control in the fixpoint computation. A natural parallel here is with the *cut* control mechanism of Prolog, which is however much more difficult to be explained in declarative terms [27]. Also, it is natural to ask ourselves whether the dynamic choice provides us with the basis for constructing *bottom-up meta-interpreters*, capable of turning logic database programs into efficient systems by exploiting a customized computation strategy. Another open problem is whether it is realistic to implement negation and ordering by choice in a real language.

Finally, we mention another direction for future work. Abiteboul and Vianu showed that certain non deterministic languages augmented with the extra possibility

of performing *updates* are capable of expressing NDB-PSPACE, i.e., the non-deterministic space-polynomial queries [6]. We conjecture that a similar result holds when augmenting Datalog with dynamic choice and an update construct, such as that of LDL.

Acknowledgements

The authors are indebted with Luca Corciulo, who contributed to an earlier version of this manuscript. Thanks are also owing to Victor Vianu, Luigi Palopoli, Carlo Zaniolo and Mimmo Saccà for their useful suggestions on the subject of this paper. This work has been partially supported by the Commission of the European Communities – EC/US Cooperative Activity ECUS033.

References

[1] L. Corciulo, F. Giannotti, D. Pedreschi, Datalog with non deterministic choice computes NDB_PTIME, in: S. Ceri, T. Katsumi, S. Tsur (Eds.), Proceedings of the Third International Conference on Deductive and Object-Oriented Databases DOOD'93, Lecture Notes in Computer Science, vol. 760, Springer, Berlin, 1993, pp. 49–65.

[2] A. Chandra, D. Harel, Structure and complexity of relational queries, J. Comput. System Sci. 25 (1) (1982) 99–128.

[3] C. Zaniolo, Object identity and inheritance in deductive databases: An evolutionary approach, in: W. Kim, J.-M. Nicolas, S. Nishio (Eds.), Proceedings of the First International Conference on Deductive and Object-oriented Databases DOOD'89, Elsevier, Amsterdam, 1989, pp. 7–21.

[4] F. Giannotti, S. Greco, D. Saccà, C. Zaniolo, Programming with non determinism in deductive databases, Ann. of Math. and Artificial Intelligence 19 (1–2) (1997) 97–126.

[5] N. Immerman, Languages with capture complexity classes, SIAM J. Comput. 16 (4) (1987) 760–778.

[6] S. Abiteboul, V. Vianu, Non determinism in logic based languages, Ann. of Math. and Artificial Intelligence 3 (1991) 151–186.

[7] N. Immerman, Relational queries computable in polynomial time, Inform. and Control 68 (1986) 86–104.

[8] M. Vardi, The complexity of relational query languages, in: Proceedings of the ACM-SIGACT Symposium on the Theory of Computing, ACM Press, 1982, pp. 137–146.

[9] S. Abiteboul, V. Vianu, Procedural languages for database queries and updates, J. Comput. System Sci. 41 (2) (1990) 181–229.

[10] S. Abiteboul, E. Simon, V. Vianu, Non deterministic language to express deterministic transformation, in: Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems PODS 90, ACM Press, 1990, pp. 218–229.

[11] S. Abiteboul, V. Vianu, Fixpoint extension of first order logic and datalog like languages, in: Proceedings of the Fourth Symposium on Logic in Computer Science LICS'89, IEEE Computer Press, 1989, pp. 71–89.

[12] R. Krishnamurthy, S. Naqvi, Non deterministic choice in datalog, in: Proceedings of the Third International Conference on Data and Knowledge Bases, Morgan Kaufmann, Los Altos, CA, 1988, pp. 416–424.

[13] D. Saccà, C. Zaniolo, Stable models and non determinism in logic programs with negation, in: Proceedings of ACM Symposium on Principles of Database Systems PODS 90, ACM Press, 1990, pp. 205–217.

[14] F. Giannotti, D. Pedreschi, D. Saccà, C. Zaniolo, Non determinism in deductive databases, in: C. Delobel, M. Kifer, Y. Masunaga (Eds.), Proceedings of the Second International Conference on Deductive and Object-oriented Databases DOOD'91, Lecture Notes in Computer Science, vol. 566, Springer, Berlin, 1991, pp. 129–146.

[15] S. Abiteboul, R. Hull, V. Vianu, Foundations of Databases, Addison–Wesley, Reading, MA, 1995.
[16] W. Cheng, J. Zeng, Non determinism through well-founded choice, J. Logic Programming 26 (3) (1996) 285–309.
[17] P.C. Kanellakis, Elements of relational database theory, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B, Elsevier, Amsterdam, 1990, pp. 1075–1155.
[18] J.D. Ullman, Principles of databases and knowledge base system, vols. I/II, Computer Science Press, Rockville, MD, 1988.
[19] E.F. Codd, Relational completeness of database sublanguages, in: R. Rustin (Ed.), Data Base Systems, Prentice-Hall, Englewood Cliffs, NJ, 1972, pp. 33–64.
[20] H. Gallaire, J. Minker, J.M. Nicolas, Logic and databases: A deductive approach, ACM Comput. Surveys 16 (2) (1984) 153–185.
[21] K.R. Apt, Introduction to logic programming, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B, Elsevier, Amsterdam, 1990, pp. 493–574.
[22] Y. Gurevich, S. Shelah, Fixed-point extensions of first-order logic, Ann. of Pure and Appl. Logic 32 (1986) 265–280.
[23] S. Naqvi, S. Taur, A logical language for data and knowledge bases, Computer Science Press, Rockville, MD, 1989.
[24] D. Chimenti et al, The $\mathscr{LDL}$ system prototype, IEEE J. Data and Knowledge Engineering 2 (1) (1990) 76–90.
[25] M. Gelfond, V. Lifschitz, The stable model semantics for logic programming, in: Proceedings of the Fifth International Conference and Symposium on Logic Programming, MIT Press, Cambridge, MA, 1988, pp. 1070–1080.
[26] L. Corciulo, Non determinism in deductive databases (in Italian), Laurea Thesis, Dipartimento di Informatica, Università di Pisa, 1993.
[27] F. Giannotti, D. Pedreschi, C. Zaniolo, Declarative semantics for pruning operators in logic programming, Methods of Logic in Computer Science 1 (1994) 61–76.