NON-DETERMINISM IN LOGIC-BASED LANGUAGES

Serge ABITEBOUL *

INRIA, Rocquencourt, 78153 le Chesnay Cedex, France

Victor VIANU **

University of California at San Diego, San Diego, CA 92037, USA

Abstract

The use of non-determinism in logic-based languages is motivated using pragmatic and theoretical considerations. Non-deterministic database queries and updates occur naturally, and there exist non-deterministic implementations of various languages. It is shown that non-determinism resolves some difficulties concerning the expressive power of deterministic languages: there are non-deterministic languages expressing low complexity classes of queries/updates, whereas no such deterministic languages are known. Various mechanisms yielding non-determinism are reviewed. The focus is on two closely related families of non-deterministic languages. The first consists of extensions of Datalog with negations in bodies and/or heads of rules, with non-deterministic fixpoint semantics. The second consists of non-deterministic extensions of first-order logic and fixpoint logics, using the *witness* operator. The expressive power of the languages is characterized. In particular, languages expressing exactly the (deterministic and non-deterministic) queries/updates computable in polynomial time are exhibited, whereas it is conjectured that no analogous deterministic language exists. The connection between non-deterministic languages and determinism is also explored. Several problems of practical interest are examined, such as checking (statically or dynamically) if a given program is deterministic, detecting coincidence of deterministic and non-deterministic semantics, and verifying termination for non-deterministic programs.

1. Introduction

While the traditional logic-based languages used in databases and AI are deterministic, recent investigations suggest that non-deterministic query and update languages may have considerable advantages. In this paper, we bring together in a cohesive framework some of the evidence to this effect. We focus primarily on results dispersed in [2–7], but also include pointers to other relevant work, such as [28,32,33,39].

* Work supported by the Projet de Recherche Coordonnée BD3.

** Work supported in part by the National Science Foundation under grants IRI-8816078 and INT-8817874. The work was done in part while the author was visiting INRIA.

The arguments in favor of non-deterministic languages are both practical and theoretical. The first is that non-determinism is already here in various forms. There are natural non-deterministic queries and updates, whose implementation using deterministic languages is contrived and inefficient. There are well-known applications in Artificial Intelligence which naturally lead to non-determinism, and expert systems shells (such as KEE [26] or OPS5 [10]) whose rule-based components work non-deterministically. The theoretical arguments for non-determinism involve primarily the expressive power of non-deterministic languages. Indeed, the use of non-determinism circumvents some of the problems associated with deterministic languages. For instance, it is conjectured that there is no deterministic language expressing exactly the queries computable in polynomial time. On the other hand, there are non-deterministic languages expressing exactly the (deterministic and non-deterministic) queries computable in polynomial time. The motivation for non-determinism is discussed in detail in the paper.

A variety of mechanisms yielding non-determinism are presented. We consider rule-based languages with non-deterministic semantics. The languages are extensions of Datalog (i.e., "pure" Prolog) allowing negations in the bodies of rules and/or deletions in heads of rules. The languages have fixpoint semantics: the result is computed by firing the rules until a fixpoint is reached. The non-determinism results from the arbitrary choice of which rule instantiation to fire next. We also consider non-deterministic extensions of traditional logic languages such as first-order logic and its fixpoint extensions. The non-determinism is provided by an operator called *witness*, which yields formulas with several different interpretations for each given structure. This provides a uniform way of obtaining non-deterministic counterparts for traditional deterministic logics. Close connections between the Datalog-like languages and the fixpoint extensions of first-order logic are exhibited. Other mechanisms for obtaining non-determinism are also reviewed, such as the *choice* operator [28,32] and the *assume* of [39].

A primary focus of the paper is on the expressive power of the non-deterministic languages. The results help understand the impact of non-determinism in conjunction with various language constructs on expressive power. In particular, non-deterministic languages expressing exactly the queries computable in polynomial time are exhibited, whereas analogous deterministic languages remain an elusive goal.

The results on expressive power suggest that non-deterministic languages, in addition to computing useful non-deterministic queries, also provide efficient means to compute otherwise "hard" *deterministic* queries. Then it becomes of interest to detect if a non-deterministic program computes in fact a deterministic query. Such connections between non-determinism and determinism are explored. For each language, we consider the non-deterministic programs whose computation paths have a "Church-Rosser" property, and thus compute deterministic transformations. Such deterministic transformations constitute the *functional fragment* of the language. We characterize the functional fragments of various languages. In particular, we exhibit languages whose functional fragment consists precisely of the transformations computable in polynomial time. We also characterize the deterministic transformations obtained by taking the union or intersection of the possible outcomes of non-deterministic programs in a given language. This is in the spirit of *possible*, respectively *certain*, answers to queries on databases with incomplete information [20], and also related to *possible worlds* semantics in truth-maintenance systems [25].

We examine several problems arising from the use of non-deterministic programs to compute deterministic transformations. The problems were suggested by practical work on implementing rule-based languages similar to ours [30,35]. First we look at the problem of *checking* whether a given program expresses a deterministic transformation. We show that static checking is generally unfeasible. However, dynamic checking can be done in some of the languages (i.e., a program can be made to issue a "warning" of non-functionality on a given input).

In the case of rule-based languages, the same program can be given deterministic or non-deterministic semantics. It is of interest in practice to know when the two semantics coincide. For instance, a program might be evaluated more efficiently with deterministic semantics, since the rules can be fired in parallel rather than one at a time. Thus, coincidence of the semantics can be used in optimization [35]. Note that this problem is related but different from the problem of whether a program with non-deterministic semantics belongs to the functional fragment. We show that the general problem of checking coincidence of the semantics is undecidable for some languages. We exhibit a language such that a decision procedure exists for queries in that language but not for updates. This highlights a surprising distinction between queries and updates, which also arises in several other results in the paper.

Some of the languages allow for programs with non-terminating computations. We study the notion of "loop-freedom". A program is loop-free if it has only terminating computations. We show that static checking of loop-freedom is generally undecidable. However, we show, again, that dynamic checking is feasible in some of the languages (i.e., programs can be made to issue "warnings" of non-termination without actually entering a non-terminating computation).

2. Why non-determinism?

Starving Frenchman:	"May I have a roast-beef sandwich, please?"
Irate Server:	"What *kinda* bread?"
Starving Frenchman:	"Any kind will do."
Irate Server:	"You *gotta* choose."
Starving Frenchman:	"But I do not know what bread you have!"
Irate Server:	"Sorry *buddy*, I can't order for you! Next please!"

Real-world experience

The immediate argument in favor of non-deterministic database languages is that there are natural non-deterministic queries/updates to be answered. Consider, for example:

- "Find one cafe at the intersection of Blvd. St-Germain and Blvd. St-Michel".
- "Assign as undergraduate advisor one junior faculty member who has not been an advisor in the past three years".

The first query has several (four) possible answers, any one of which will satisfy the user. The update has also several possible outcomes, but which particular one is chosen is immaterial. (In fact, the person issuing such an update may well prefer that the actual choice be left up to the system!)

The update shown above is reminiscent of applications in AI involving *programming with constraints*. Another well-known example is "Mrs. Manners", where a set of diners must be seated at the table subject to certain constraints which allow for several possible solutions [26] (see also example 4.1).

A second practical argument in favor of a serious look at non-deterministic languages is realism: there are in fact non-deterministic languages already implemented. For instance, several implementations of production systems and rule-based expert system shells are non-deterministic. While in some cases the non-determinism is "accidental", the result of poorly understood semantics or of ad-hoc implementations, in other cases the non-determinism is deliberate. One example in the latter category is RDL1 [30,35], other, well-known ones are KEE [26] and OPS5 [10]. For instance, [26] describes a non-deterministic KEE program for the "Mrs. Manners" problem, where each solution of the problem is a possible outcome of the program.

The examples above can be viewed as partially specified queries or updates, where the specification leaves room for several acceptable solutions. A valid question is whether such queries or updates require truly non-deterministic languages, or whether the non-determinism can be "simulated" using traditional deterministic languages. Roughly speaking, in the absence of some mechanism for non-deterministic choice, deterministic programs are generally forced to produce simultaneously *all* possible solutions. While this overkill may be acceptable in some cases such as the first query, it is not acceptable in others such as the update in the second example (one might wind up with several undergraduate advisors, which was not intended and may violate the integrity constraints of the database). Equally important, the cost of unnecessarily producing all solutions can be prohibitive.

More surprisingly, it turns out that the availability of non-deterministic choice can yield more efficient ways to compute *deterministic* queries. Consider, for example, the well-known "parity" query (R is a unary relation, i.e. a set):

$$even(R) = \begin{cases} \textit{true} & \text{if } |\mathrm{R}| \text{ is even,} \\ \textit{false} & \text{if } |\mathrm{R}| \text{ is odd.} \end{cases}$$

The natural way to compute the query is to remove elements from R one at a time, and keep a binary counter. However, the elements of R are conceptually undistinguishable one from another, so picking a single element requires a non-deterministic choice. This can be easily done if a mechanism for non-deterministic choice is available. However, this query becomes hard with purely deterministic means: indeed, it has been shown that the parity query cannot be computed by the relational algebra (first-order logic), and not even by its powerful iterative extensions, the *fixpoint queries* (complete in PTIME) and the *while queries* (complete in PSPACE) [11,13].

The parity query is a deterministic query computable in polynomial time, but not computable by the usual deterministic query languages. A major open question is whether there exists a deterministic language expressing exactly the queries computable in polynomial time. The answer is conjectured to be negative for polynomial time, as well as for all lower complexity classes of queries. For the parity query, we saw that non-determinism can be used to circumvent the difficulty in the computation. It turns out that this situation extends to all queries computable in polynomial time. Thus, there are non-deterministic languages which compute exactly the queries computable in polynomial time. Of course, programs in such a language compute non-deterministic queries in addition to the deterministic ones. This suggests a trade-off between determinism and expressive power: one gives up the guarantee that every program in the language is deterministic, in exchange for the ability to express "nice" classes of queries and updates.

The need for non-deterministic choice in the parity example arises because the elements of the input set are indistinguishable at the conceptual level. This is no longer the case if the data independence principle is renounced by allowing the program access to the internal storage. This provides in effect an *ordering* of the elements of the set, which allows distinguishing one element from another. With such an ordering, it is easy to compute parity deterministically: pick the elements in R one at a time in increasing order and keep a counter. Again, the situation for the parity query is symptomatic: indeed, if an ordering of the domain is available, all queries computed in polynomial time can be computed by the fixpoint queries [22,38]. More generally, it has been shown that many complexity classes of queries can be captured by deterministic languages in the presence of order [23]. Of course, the price to pay for the violation of the data independence principle is that users must know about the internal storage in order to write queries. Also, one can then write programs where access to the internal storage is used abusively and yields queries which are nonsensical at the conceptual level, such as "Find all departments whose internal binary representation is a prime number". Thus, consistency with the information provided at the conceptual level is no longer assured.

Intuitively, there is a strong connection between the use of order (i.e., information on the internal storage) and non-determinism. If a query is implemented using information on internal storage, abstracted at the conceptual level, then the answer may depend on such information and thus appear non-deterministic at the conceptual level. The trade-off between determinism and expressive power is an alternative to the trade-off between data-independence and expressive power. Indeed, the following conjectured meta-theorem has been consistently confirmed (C is a time or space Turing complexity class at least linear):

> If the class of deterministic queries in C can be expressed by a deterministic language in the presence of order, then the class $N-C$ of deterministic and non-deterministic queries computable in C can be expressed by a non-deterministic language.

The idea behind the meta-theorem is that an arbitrary order which can be used to compute the queries in C can be generated by non-deterministic means in linear time/space.

In connection with the update example above, we suggested that deterministic implementations may lead to violations of the integrity constraints of the database. We further elaborate on the need for non-determinism in updates. Suppose that the valid updates of a database are specified by a set of *admissible operations*, given as a finite set of procedures of the form $p(x, y, \ldots, z)$. Valid instances of the database are then generated by repeated calls to the given procedures. On the other hand, suppose that the valid instances of the database which must be generated are those which satisfy certain static integrity constraints. Given a specification using integrity constraints and an update language, it is of interest whether one can write a set of procedures in the given language which can be used to generate the database instances satisfying the integrity constraints. This issue was examined at length in [8,9]. In particular, it was shown in [9] that some types of integrity constraints require a non-deterministic update language. For instance, specifications using common constraints such as functional dependencies, inclusion dependencies, and join dependencies may require non-deterministic update capabilities. This is illustrated by the following.

Example 2.1

Consider a relation R over attributes ABC and the following set of constraints:

- the embedded join dependency

  $\forall xyzx'y'z'(R(xyz) \wedge R(x'y'z') \Rightarrow R(xy'z))$,
- the functional dependency C $\rightarrow$ AB,
- the inclusion dependency R[A] $\subseteq$ R[C],
- the inclusion dependency R[B] $\subseteq$ R[A],
- the inclusion dependency R[A] $\subseteq$ R[B].

It is easy to check that, for each relation satisfying the constraints, the number of constants in the relation is a perfect square (n^2, $n \geqslant 0$). Intuitively, non-determinism is needed because the unbounded "gaps" between instances cannot be crossed in a deterministic fashion, using procedures which supply only a bounded number of parameters at each call. □

We have reviewed some of the practical and theoretical arguments for non-determinism. Another potentially important but less explored consideration is optimization. Intuitively, a non-deterministic program yields a certain degree of freedom in the computation of a query or update. Such freedom can be exploited in optimization. An example is provided by the use of the *choice* operator in Datalog [28,32]. The availability of alternative execution paths in non-deterministic programs is likely to be of use in concurrency control as well. Indeed, a scheduler with more options available is likely to yield increased concurrency.

3. Background

3.1. PRELIMINARIES

In this section we review some terminology relating to relational databases. In particular, we recall some of the traditional deterministic languages, including iterative extensions of first-order logic and relational algebra (the *fixpoint* and *while* queries), and several Datalog-like languages.

We assume the reader is familiar with the basic concepts and terminology of relational database theory (see [37]). We also refer to [24] for a survey of the field. We review briefly some of the basic terminology and notation.

We assume the existence of three infinite and pairwise disjoint sets of symbols: the set **att** of *attributes*, the set **dom** of *constants*, and the set **var** of *variables*. A *relational schema* is a finite set of attributes. A *tuple* over a relational schema R is a mapping from R into **dom** ∪ **var**. A *constant tuple* over a relational schema R is a mapping from R into **dom**. An *instance* over a relation schema R is a *finite* set of constant tuples over R. The projection of an instance r over R on a subset S of its attributes is denoted $\pi_S(r)$. A *database schema* is a finite set of relational schemas. An *instance* I over a database schema $\mathbf{R}$ is a mapping from $\mathbf{R}$ such that for each R in $\mathbf{R}$, $I(R)$ is an instance over R. The set of all instances over a schema $\mathbf{R}$ is denoted by $inst(\mathbf{R})$.

Note that, in logic terms, a database schema supplies a finite set of predicates, and a database instance provides an interpretation of the predicates into *finite* structures. Indeed, only finite structures are considered in this paper.

We are interested primarily in database queries and updates, which involve transformations of database instances into other database instances. We distinguish here between *deterministic* and *non-deterministic database transformations.* A *non-deterministic database transformation* is a subset of $inst(\mathbf{R}) \times inst(\mathbf{S})$ for some $\mathbf{R}$, $\mathbf{S}$, and a *deterministic* database transformation is a mapping from $inst(\mathbf{R})$ to $inst(\mathbf{S})$. If $\mathbf{R}$ and $\mathbf{S}$ are disjoint, we will say that the transformation is a *query*, otherwise it is an *update*. In the case of queries, the input predicates are sometimes referred to as EDB (extensional database) predicates, and the output predicates as IDB (intensional database) predicates.

Database transformations are usually required to obey three conditions: *well-typedness, effective computability* and *genericity* [4,12]. Well-typedness is captured by requiring that instances over a *fixed* schema be related to instances over another *fixed* schema. Effective computability is self explanatory. Genericity originates from the data independence principle, discussed also in the previous section: a query or update can only use information provided at the conceptual level. In particular, distinct data values can be treated differently only if they can be distinguished using the information available at the conceptual level, or if they are named explicitly in the query/update. This is formalized by the notion of genericity:

> Let **R** and **S** be database schemas, and C a finite set of constants. A subset τ of $inst(\mathbf{R}) \times inst(\mathbf{S})$ is *C-generic* iff for each bijection ρ over **dom** which is the identity on C, $(I,J) \in \tau$ iff $(\rho(I), \rho(J)) \in \tau$. A transformation from $inst(\mathbf{R})$ to $inst(\mathbf{S})$ is *C-generic* iff its graph is C-generic.

In the definition of genericity, the set C specifies "exceptional" constants, which can be treated differently from other constants. This is needed because a query or update can explicitly name a finite set of constants.

We will refer to complexity classes of database transformations. We use as complexity measures the time and space used by a Turing Machine to produce a standard encoding of the output instance starting from an encoding of the input instance. Note that this is slightly different from the traditional definition in terms of the associated recognition problem [38], where the complexity of a transformation is defined as the complexity of deciding if a given tuple belongs to the result. Clearly, this is no longer appropriate for non-deterministic transformations, where several answers are possible. The complexity measures are functions of the size of the input instance. For each Turing Machine complexity class C, there is a corresponding complexity class of (non-deterministic) transformations denoted (N)DB-C. In particular, the class of non-deterministic database transformations which can be computed by a non-deterministic Turing Machine in polynomial time is denoted NDB-PTIME. By Savitch's theorem, PSPACE = NPSPACE. Note that a DB-PSPACE transformation is deterministic by definition, whereas NDB-PSPACE contains non-deterministic transformations, so DB-PSPACE $\neq$ NDB-PSPACE. For deterministic transformations, we sometimes describe complexity in terms of the associated recognition problem. Thus, DB-NP denotes deterministic transformations for which the recognition problem is in NP. (DB-NP is not to be confused with NDB-PTIME!) We will specify which measure is used only when the distinction is relevant.

Given a program P (in a transformation language L), the mapping (or relation) between database instances that the program describes is called the *effect* of the program, and denoted $\mathit{eff}_L(P)$. The concepts of *input, output* and *temporary relations* are also important. When a program is applied to a given database, its effect is often interpreted by identifying some relations as input relations, and other relations as output relations. In addition to semantically significant input and output relations, the programs may use "temporary" relations. Thus, it appears useful to also define the effect of a program with respect to specified input and output database schemas. Given a program P and two schemas $\mathbf{R}$ and $\mathbf{S}$, P transforms instances over $\mathbf{R}$ into instances over $\mathbf{S}$ as follows: relations which are not in the input schema are assumed to be empty before the program is executed; after the program is run, the relations in the output schema must contain the desired result. (The content of the other relations is immaterial.) The effect thereby obtained is denoted by $\mathit{eff}_L(\mathbf{R}, \mathbf{S}, P)$. A program P with input schema $\mathbf{R}$ and output schema $\mathbf{S}$ defines a query if $\mathbf{R} \cap \mathbf{S} = \emptyset$, and an update otherwise.

3.2. REVIEW OF DETERMINISTIC LANGUAGES

We assume the reader is familiar with relational algebra, relational calculus (i.e. the *first-order queries*, denoted FO), and Datalog, or "pure" Prolog (see, for instance, [37]). In this section, we review several deterministic Datalog-like languages and the deterministic fixpoint extensions of first-order logic.

The Datalog extensions we consider allow the use of negation in the bodies of rules and deletion in heads of rules. With the deterministic semantics, the program is evaluated by "firing", in parallel, all applicable instantiations of the rules. This is repeated until a fixpoint is reached (if ever). For instance, the program

$$\neg G(x, y) \leftarrow G(x, y), G(y, x)$$

removes (in a single stage) all edges of the graph G participating in cycles of length two.

The syntax of the Datalog extensions is defined next.

DEFINITION 3.1

A *Datalog*$^{\neg *}$ program is a finite set of rules of the form

$$(\neg)Q(x_1, \dots, x_m) \leftarrow B_1, \dots, B_n$$

($m \geqslant 0$, $n \geqslant 0$) where each x_j is a constant or a variable and each B_i a literal of the form $(\neg)Q(x_1, \dots, x_m)$ ($m \geqslant 0$). Furthermore, it is required that each variable occurring in the head of a rule also occur positively bound in the body. □

The set of relations occurring in a program P is denoted $sch(P)$.

We will be also interested in restricted versions of the languages. The "$*$" indicates that negations are allowed in heads of rules and "$\neg$" that they are allowed in the bodies. If the literals in heads are all positive, the program is also a *Datalog*$^{\neg}$ program. If the literals in bodies are all positive, the program is also a *Datalog** program. Finally, if all literals are positive, the program is also a *Datalog* program.

Note that, on some input, the simultaneous instantiation of the rules of a *Datalog*$^{\neg *}$ program may produce a set of inconsistent facts, containing both A and $\neg A$ for some A. In this case, the computation blocks and the query is undefined. Also note that *Datalog*$^{\neg *}$ programs may not terminate. However, *Datalog*$^{\neg}$ programs always terminate in polynomial time.

We next review the fixpoint (iterative) extensions of first-order logic. The extensions consist of augmenting first-order logic with fixpoint operators, which provide recursion. Essentially, a fixpoint operator allows defining a relation inductively, by repeated applications of a given first-order formula. For instance, the transitive closure T of a relation R (not definable in first-order logic alone) can be defined by recursive applications of the formula

$$\phi(x, y) = T(x, y) \vee R(x, y) \vee \exists z(T(x, z) \wedge T(z, y))$$

(the result of each iteration is reassigned to T). While in this example a fixpoint is always reached, this is not always the case. Generally, the fixpoint operator is partially defined, so interpretations of sentences in the logic are partially defined.

We next discuss *partial* fixpoint logic, which is first-order logic extended with a partial fixpoint operator.

DEFINITION 3.2

Partial fixpoint formulas are obtained by repeated applications of first-order operators ($\neg$, $\wedge$, $\vee$, $\exists$, $\forall$) and the partial fixpoint operator starting from atoms. The partial fixpoint operator is defined as follows. Let $\phi(S)$ be an $FO + PFP$ formula with n free variables, where S is an n-ary predicate occurring in ϕ. Then $PFP(\phi(S), S)(\vec{t})$ is a formula, where $\vec{t}$ is a sequence of n variables or constants. The interpretation of $PFP(\phi(S), S)$ is the following: for a given instance of the database, $PFP(\phi(S), S)$ denotes the n-ary predicate which is the limit, if it exists, of the sequence defined by: $J_0 = \mathrm{I}(S)$ and for each $i > 0$, $J_i = \phi(J_{i-1})$ (the result of evaluating ϕ on the database instance where S is assigned J_{i-1}). If ϕ is undefined on J_{i-1}, then J_i and the interpretation of $PFP(\phi(S), S)$ are undefined. The predicate $PFP(\phi(S), S)$ can then be used inside formulas just like the atomic predicates. □

The $FO + PFP$ formula defining the transitive closure of R using the formula ϕ shown earlier, is $PFP(\phi(\mathrm{T}), \mathrm{T})(\mathrm{x}, \mathrm{y})$.

The queries expressible by $FO + PFP$ formulas are called the *while* queries. They were originally introduced via a procedural language in [11,13].

As mentioned above, the *PFP* operator is generally partially defined. One can ensure termination of the iteration by adopting a semantics which provides monotonicity. This variation of the *PFP* operator is called the *inductive* fixpoint operator, denoted *IFP*. The difference with *PFP* is that at each iteration, the newly computed relation is *added* to the relation computed at the previous iteration. This ensures convergence in polynomial time on all inputs. The first-order logic augmented with the *IFP* operator is called *inflationary fixpoint logic* and is denoted $FO + IFP$. The queries computed by $FO + IFP$ are referred to as the *fixpoint queries*. Various equivalent definitions of the fixpoint queries exist in the literature [11,13,19].

It turns out that there are close connections between the fixpoint extensions of first-order logic and the Datalog extensions we presented. In [5], we showed that *Datalog*$^{\neg}$ expresses exactly the class of fixpoint queries. It also provides an existential normal form for $FO + IFP$ (see also the remark in section 6 of [18]). The equivalence between *Datalog*$^{\neg}$ and $FO + IFP$ in conjunction with recent results by Kolaitis [27] and Dahlhaus [15] showing that Datalog with stratified negation is strictly included in $FO + IFP$, implies that *Datalog*$^{\neg}$ is *strictly* more expressive than Datalog with stratified negation. It is also shown in [6] that *Datalog*$^{\neg *}$ is equivalent to $FO + PFP$.

4. Non-deterministic languages

In this section, we consider several mechanisms for introducing non-determinism in query and update languages. We focus primarily on Datalog extensions with non-deterministic semantics [5,6], and non-deterministic counterparts of the fixpoint logics [6,7].

4.1. DATALOG-LIKE LANGUAGES

We first consider non-deterministic versions of the deterministic Datalog-like languages described earlier.

Recall that the deterministic semantics for the Datalog-like languages was the result of evaluating programs by repeatedly firing all rules *in parallel*. The non-deterministic semantics is obtained by firing one instantiation of a rule at a time, based on a non-deterministic choice. For instance consider again the program:

$$\neg G(x, y) \leftarrow G(x, y), G(y, x).$$

Recall that, with the deterministic semantics, the program removes all cycles of length two. With the non-deterministic semantics, the program computes one of several possible "orientations" for the graph G (i.e., for every pair of edges (x, y) and (y, x) in G, one of the edges is removed).

We first define the syntax of the non-deterministic version of *Datalog*$^{\neg *}$, denoted *N-Datalog*$^{\neg *}$. The difference is that, in the non-deterministic version, heads of rules may contain several literals, and equality can be used in bodies. It can be seen that these features would be redundant with the deterministic semantics.

DEFINITION 4.1

An *N-Datalog*$^{\neg *}$ program is a finite set of rules of the form

$A_1,\dots, A_k \leftarrow B_1,\dots, B_n$

($k \geqslant 1$, $n \geqslant 0$), where each A_j is a literal of the form $(\neg)Q(x_1,\dots, x_m)$ ($m \geqslant 0$), and each B_i is a literal of the same form, or $(\neg)x_1 = x_2$ (the x_i's are variables or constants). It is required that each variable occurring in the head of a rule also occur positively bound in the body. □

As in the deterministic case, we will also be interested in restricted versions of the language. The "*" indicates that negations are allowed in heads of rules and "$\neg$" that they are allowed in the bodies. Thus, if the literals in heads are all positive, the program is also an *N-Datalog*$^{\neg}$ program. If the literals in bodies are all positive, the program is also an *N-Datalog** program. Finally, if all literals are positive, the program is also an *N-Datalog* program.

To formally define the non-deterministic semantics, we introduce the notion of (non-deterministic) immediate successor of a set of facts using a rule. Let r be an *N-Datalog*$^{\neg *}$ rule. Let I be a set of facts and r' be a ground instance of r such that (i) each literal of the body of r' is true in I, (ii) the head of r' is consistent and (iii) each variable is valuated to some constant occurring in I. Then the instance J obtained from I by deleting the facts A such that $\neg A$ is in the head of r', and inserting the facts A in the head of r', is called an *immediate successor* of I using r.

By condition (ii) above, a ground instance of a rule is not considered if its head contains a ground literal and its negation.

DEFINITION 4.2

Let P be an *N-Datalog*$^{\neg *}$ program. The *effect* of P is a relation over sets of facts defined as follows: for each I, (I, J) is in *eff*(P) iff there exists a sequence $I_0 = I,\dots, I_n = J$ such that (i) for each i, I_{i+1} is an immediate successor of I_i using some r in P, and (ii) there is no immediate successor $J' \neq J$ of J using some rule in P. □

Remark

To each rule

$A_1,\dots, A_k \leftarrow B_1,\dots, B_n,$

in P, we associate the first-order sentence

$\forall \vec{x}(B_1 \wedge \ldots \wedge B_n \Rightarrow A_1 \wedge \ldots \wedge A_k)$,

where $\vec{x}$ is the vector of the variables occurring in the rule. Let P be a program such that for each ground instance r of a rule in P, the head of r' does not contain a fact A and its negation. Let $\Sigma(P)$ be the set of sentences associated with the rules in P. For each I, J, if $\langle I, J\rangle \in \mathit{eff}(P)$ then J is a model of $\Sigma(P)$. Furthermore, if negation is not allowed in heads of rules, then J is a model of $\Sigma(P)$ containing I. □

Following is an example of a program for the "Mrs. Manners" problem.

Example 4.1

We show how the "Mrs Manners" problem can be solved using non-deterministic languages like those studied in the paper. We assume that two relations are given. The first one, *guest*, gives for each guest, his/her name, sex and hobbies. The second one, *next*, specifies the adjacent seats. For instance, a possible input is:

guest(*jeremie*, *male*, *chess*)	*next*(1, 2)
guest(*jeremie*, *male*, *tennis*)	*next*(2, 3)
guest(*manon*, *female*, *chess*)	*next*(3, 4)
guest(*gaspard*, *male*, *chess*)	*next*(4, 1)
guest(*gaspard*, *male*, *nap*)	
guest(*margot*, *female*, *nap*)	
guest(*margot*, *female*, *tennis*).	

Guests must be seated so that neighbours are of different sexes and share a hobby. (To simplify, assume that the number of seats equals the number of guests.) Intuitively, guests are randomly seated using the first two rules. The compatibility relation (indicating who can sit next to whom) is computed using the third rule. The last rule is used to check that the seating was correct. The predicate *current* holds the identifier of the last seat where somebody was seated. The predicate *seated* contains the names of the guests that have been seated so far. Finally, *seating* tells who sits where. The rules are as follows (with the symbol $\perp$ indicating an error):

$$\begin{array}{rcl} \mathit{seated}(\mathit{name}),\ \mathit{current}(\mathit{seat1}), & \leftarrow & \mathit{guest}(\mathit{name},\ \_,\ \_), \\ \mathit{started},\ \mathit{seating}(\mathit{name},\mathit{seat1}) & & \neg\ \mathit{started},\ \mathit{next}(\mathit{seat1},\ \_) \end{array}$$

$$\begin{array}{rcl} & & \mathit{current}(\mathit{seat1}), \\ \neg\ \mathit{current}(\mathit{seat1}),\ \mathit{current}(\mathit{seat2}), & \leftarrow & \mathit{next}(\mathit{seat1},\mathit{seat2}), \\ \mathit{seated}(\mathit{name}),\ \mathit{seating}(\mathit{name},\ \mathit{seat2}) & & \mathit{guest}(\mathit{name},\ \_,\ \_), \\ & & \neg\ \mathit{seated}(\mathit{name}) \end{array}$$

$$compatible(name1,\ name2) \leftarrow \begin{array}{l} guest(name,\ sex1,\ hobby), \\ guest(name,\ sex2,\ hobby), \\ sex1 \neq sex2 \end{array}$$

$$\perp \leftarrow \begin{array}{l} seating(name,\ seat1), \\ seating(name,\ seat2), \\ next(seat1,\ seat2), \\ \neg\ compatible(name1,\ name2) \end{array}$$

With the above program, the correct solutions are defined to be those where $\perp$ is not inferred. (This particular language, denoted *N-Datalog*$^{\neg}\perp$, is discussed in [5,6], and in section 5.2.) However, $\perp$ can be inferred if a wrong choice has been made. This may require backtracking. Some of our languages can indeed simulate the control necessary to simulate backtracking. Note also that smarter algorithms can be easily implemented, for example one that checks compatibility whenever assigning a seat. □

4.2. NON-DETERMINISTIC FIXPOINT EXTENSIONS OF FO

In this section, we consider fixpoint extensions of first-order logic which correspond to non-deterministic languages. Such extensions allow formulas that define *several* predicates for each given structure. This is achieved by a non-deterministic operator on formulas, called the *witness* operator. Informally, given a formula $\phi(x)$, the witness operator Wx applied to $\phi(x)$ chooses an arbitrary witness x which makes ϕ true. The witness operator is related to Hilbert's ϵ-symbol [29]. Its semantics is quite different.

The extension based on the witness operator is orthogonal to the fixpoint extensions of first-order logic corresponding to the deterministic languages. Thus, we will consider inflationary and non-inflationary versions of fixpoint logic with the W operator, denoted $FO + IFP + W$ and $FO + PFP + W$, respectively. We note that each "deterministic" logic has a natural W-extension. Thus, one can consider W-extensions of first-order logic, Horn clause logic (Datalog), etc. This yields a family of "non-deterministic" logics parallel to the traditional logics used for query languages. Following is a simple example of the use of the witness operator in conjunction with first-order logic.

Example 4.2

Consider two relations

bonus(passenger-name) and

records(passenger-name, flight♯, day, month, year)

of an airline database. Relation *bonus* holds the names of all passengers who have been given a bonus for which it is necessary to have flown in March 1988.

The following ($FO + W$) formula defines a relation *verification* which exhibits *one* qualifying flight (flight♯ and day) for each passenger given the bonus:

$$\mathit{verification}(\mathrm{n, f, d}) = \mathit{bonus}(\mathrm{n}) \wedge W\mathrm{fd}(\mathit{records}(\mathrm{n, f, d, \text{“March”}, \text{“1988”}})). \quad \square$$

More precisely, the W operator is used in conjunction with first-order formulas as follows. If $\phi(\vec{x}, \vec{y})$ is a formula, where $\vec{x}$ and $\vec{y}$ are vectors of distinct free variable in ϕ, then $W\vec{x}(\phi(\vec{x}, \vec{y}))$ is a formula. All free variables of ϕ, including $\vec{x}$, remain free in $W\vec{x}(\phi(\vec{x}, \vec{y}))$.

We next define informally the semantics of the W operator. In this context, a formula defines a *set* of predicates, i.e., the set of possible interpretations of the formula. Let $W\vec{x}(\phi(\vec{x}, \vec{y}))$ be a formula, where $\vec{y}$ is the vector of variables other than $\vec{x}$ which are free in ϕ. The set of predicates defined by $W\vec{x}(\phi(\vec{x}, \vec{y}))$ is the set of I such that for some J defined by ϕ,

- $I \subset J$,
- for each $\vec{y}$ for which $[\vec{x}, \vec{y}]$ is in J for some $\vec{x}$, there exists a *unique* $\vec{x}_y$ such that $[\vec{x}_y, \vec{y}]$ is in I.

Intuitively, one “witness” $\vec{x}_y$ is chosen for each $\vec{y}$ satisfying $\exists\vec{x}\phi(\vec{x}, \vec{y})$.

It is also possible to describe the semantics of the W operator using functional dependencies: for each instance J defined by $\phi(\vec{x}, \vec{y})$, $W\vec{x}(\phi(\vec{x}, \vec{y}))$ defines all maximal sub-instances I of J such that the attributes corresponding to the variables in $\vec{y}$ form a key in I.

Note that, in general, $Wx(Wy\phi(x, y))$ is *not* equivalent [1] to $Wxy\phi(x, y)$; also, $Wx(Wy\phi(x, y))$ is not equivalent to $Wy(Wx\phi(x, y))$. To see the latter, let $\phi = \mathrm{R}(x, y)$, where R is interpreted as $\{[0,1], [2,1], [2,3]\}$. Note first that $\{[0,1], [2,1]\}$ and $\{[0,1], [2,3]\}$ are the only possible interpretations of WyR(x, y), and $\{[0,1]\}$, $\{[2,1]\}$ and $\{[0,1], [2,3]\}$ the only possible interpretations of Wx(WyR(x, y)). It is easily seen that $\{[2,3]\}$ belongs to the set of predicates defined by Wy(WxR(x, y)), so

$$Wx(WyR(x, y)) \text{ and } Wy(WxR(x, y))$$

are not equivalent.

Although the focus here is on $FO + IFP + W$ and $FO + PFP + W$, it is useful to first examine in more detail the W-extension of first-order logic or relational algebra. Example 4.2 shows an $FO + W$ query. Similarly, relational algebra can be extended with an operator $W_X(\mathrm{R})$, where R is a relation and X a subset of its attributes (the semantics is the obvious one). It can be shown that the W-extensions of the calculus and algebra are equivalent, so we shall look just at the calculus extension $FO + W$. However, the language $FO + W$ suffers from some subtle drawbacks. This arises from the fact that two identical subformulas in a $FO + W$ formula can define distinct relations due to non-determinism. Moreover, there is no mechanism to re-use an intermediate result defined by a non-de-

[1] Two formulas are *equivalent* iff they define the same set of predicates for each given structure.

terministic subformula. For instance, the formula

$$\exists x(Wy(R(x, y))) \vee \exists y(Wy(R(x, y)))$$

does *not* define

$$\pi_y(p) \cup \pi_x(p)$$

for each *fixed p* defined by $Wy(R(x, y))$, but instead it defines

$$\pi_y(q) \cup \pi_x(r)$$

for all *independent* choices of q and r among the relations defined by

$$Wy(R(x, y)).$$

To remedy this rather artificial difficulty, we must allow a mechanism to name subformulas. We opt for a variation of *FO*, denoted FO^+, where a program is a sequence of statements of the form $r := \phi$, where r is a relation variable and ϕ an *FO* formula. Each formula can only use relation symbols from the database or defined in previous statements. For example, the program

$$p := Wy(R(x, y));$$

$$answer := \exists x(p(x, y)) \vee \exists y(p(x, y)).$$

now defines

$$\pi_y(p) \cup \pi_x(p)$$

for each fixed p defined by $Wy(R(x, y))$. Clearly, FO^+ is equivalent to *FO*. However, $FO^+ + W$ is more powerful than $FO + W$ (with respect to the non-deterministic transformations computed) due to the ability to re-use results defined by non-deterministic subformulas.

We proceed with the non-deterministic extensions of the fixpoint logics, $FO + IFP + W$ and $FO + PFP + W$. Formulas in these logics are obtained by repeated applications of first-order operators, the *IFP* (*PFP*) operator, and the *W* operator. Note that the naming problem discussed above for $FO + W$ does not arise in $FO + IFP + W$ and $FO + PFP + W$ because intermediate results can be named and re-used from one iteration to the next. Following is a simple example in $FO + PFP + W$:

Example 4.3

Let G be a symmetric, binary relation. Consider the formula (in $FO + PFP + W$):

$$PFP(\phi(G), G)(x, y),$$

where

$$\phi(x, y) = [G(x, y) \wedge \neg Wxy(G(x, y) \wedge G(y, x))].$$

It defines an orientation G' of G, where one edge $[x, y]$ is retained for each $[x, y]$ and $[y, x]$ in G. The program removes from G one "redundant" edge at each iteration. Note that an orientation G' of G cannot generally be defined by deterministic means, since a non-deterministic choice of the edges to be removed is generally required. □

The semantics of the *IFP* and *PFP* operators are similar to the ones for the deterministic case, with the complication that each stage of the iteration has several possible outcomes.

In section 5 we examine the expressive power of the non-deterministic Datalog-like languages and fixpoint extensions of *FO*. In particular, we will show that the Datalog languages are equivalent to their fixpoint counterparts. The simulations of $FO + IFP + W$ and $FO + PFP + W$ by the Datalog extensions, and the converse simulations, provide some interesting results on $FO + IFP + W$ and $FO + PFP + W$ themselves. The results concern normal forms (implying the collapse of the respective hierarchies based on the depth of nesting of the fixpoint operators). In particular, it is shown that $FO + IFP + W$ has a "*W*" normal form and a "*W-exists*" normal form on ordered databases.

PROPOSITION 4.4

(i) For each $FO + IFP + W$ formula ϕ, there exists a first-order formula ψ whose free variables are $\vec{x}$, such that ϕ is equivalent to

$$IFP(W\vec{x}\psi(\vec{x}), T)(\vec{t})$$

for some $\vec{t}$ and predicate T of ψ.

(ii) For each $FO + IFP + W$ formula ϕ, there exists an *existential* first-order formula ψ with free variables $\vec{x}$, such that ϕ is equivalent *on ordered databases* to

$$IFP(W\vec{x}\psi(\vec{x}), T)(\vec{t})$$

for some $\vec{t}$ and predicate T of ψ.

(iii) For each $FO + PFP + W$ formula ϕ, there exists an $FO + W$ formula ψ such that ϕ is equivalent to

$$PFP(\psi, T)(\vec{t})$$

for some $\vec{t}$ and predicate T of ψ. □

Remark

It turns out that $FO + PFP + W$ has an *existential* normal form.

4.3. OTHER APPROACHES

In [39], Warren proposes a modal operator "assume" as an alternative to Prolog's "assert". The idea of viewing an update as a modality is elaborated in

[17,31]. Non-determinism is naturally incorporated in such approaches. To illustrate this, we consider the language DLL of [31].

Example 4.5

Consider a database where the relation ES stores the salary of each employee, and the relation ED the department of each employee. A tuple $AS(dep, avg)$ indicates that the average salary of department dep is avg. Consider first the following Prolog program:

:- $assert(ES(\text{John}, 200\text{K}))$, $assert(ED(\text{John}, \text{toy}))$, $AS(\text{toy}, avg)$,
$avg < 50\text{K}$.

Intuitively, this is intended to hire John in the toy department with a salary of 200K if the average salary of the department stays below 50K. However, because of Prolog semantics, if the average salary after hiring John is more than 50K, John has nevertheless been hired. (The reader might try to write the "correct" Prolog program.)

Now consider the following DLL update procedure:

$$\langle hire\ (emp, sal, dep)\rangle$$
$$\leftarrow \langle +ES(emp, sal)\rangle(\langle +ED(emp, dep)\rangle(AS(dep, avg)\ \&\ avg < 50\text{K}))$$

A call $hire$(John, 200K, toy) hires John in the toy department only if after hiring him, the average salary of the department stays below 50K. The "+" symbols indicate insertions. The parentheses after the insertions in ES and ED indicate that after the two tuples have been inserted the condition must hold. Otherwise, the update is not realized (i.e., the system backtracks). Tuple insertions and deletions in DLL can be viewed as modal operators. A key difference with Prolog *assert* is that the database updates are "undone" while backtracking. □

Note that DLL is essentially non-deterministic. This is illustrated by the following example:

Example 4.6

Consider the DLL update:

$$\langle enroll(Ename)\rangle \leftarrow \langle +sec1(Ename)\rangle(size(sec1, N)\ \&\ N < 30)$$
$$\langle enroll(Ename)\rangle \leftarrow \langle +sec2(Ename)\rangle(size(sec2, N)\ \&\ N < 30)$$

The semantics is that, for a given employee name, a rule is non-deterministically chosen. One tries to enroll the employee in a section. If this does not succeed, some backtracking is done. If neither section is full, the employee is enrolled in section 1 or 2. If they are both full, the update fails. □

In *LDL* [28,32], the primitive *choice* is introduced to provide a form of non-determinism. The choice primitive is closely related to our *W*-operator. We illustrate it with an example.

Example 4.7

Suppose that we have a relation *emp* that relates *employees* and *departments* and that we want to select one employee per department. To do that, we use the *LDL* rule:

$$selectEMP(Name) \leftarrow emp(Name,\ Dept),\ choice((Dept),\ (Name)).$$

The *choice* predicate forces the non-deterministic choice of any maximal subset of *emp* satisfying the functional dependency *Dept* → *Name*. It is therefore similar to: $W_{Name}(emp)$. □

It is observed that the *choice* primitive can be viewed as an alternative for the *cut* in Prolog. More precisely, a "pure" non-deterministic version of Prolog without meta-predicates and without left-to-right search order is considered. It is shown that in this context, *cut* can be simulated using *choice*.

A further use of non-determinism in logic programs is proposed in [33]. They provide a non-deterministic semantics for Datalog extended with negation, where each *stable model* is a possible outcome of the program. Note that it is not always possible to compute one stable model of a program deterministically. This is illustrated by the following example from [33].

Example 4.8

Consider the following program, where *takes*(*student*, *course*) is the input predicate and *one-student*(*student*, *course*) is the output predicate:

$$one\text{-}student(St,\ Crs) \quad \leftarrow \quad \begin{array}{c} takes(St,\ Crs), \\ \neg\ different\text{-}student(St,\ Crs) \end{array}$$

$$different\text{-}student(St,\ Crs) \quad \leftarrow \quad \begin{array}{c} St' \neq St,\ takes(St,\ Crs), \\ one\text{-}student(St',\ Crs) \end{array}$$

In the stable models of the program, relation *one-student* contains *one arbitrary* student associated with each course. With the non-deterministic semantics proposed in [33], each of the stable models is a possible outcome of the program. Note that this is equivalent to $W_{St}(takes(St,\ Crs))$. Also note that there can be no deterministic program producing *one* of the stable models, because then genericity would be violated. In other words, producing *any* of the stable models *requires* a non-deterministic choice. □

It is shown in [33] that the non-deterministic semantics based on stable models subsumes the *choice* construct described above, and thus the *cut* in Prolog.

Lastly, in [36] non-determinism is introduced in logic programs using a construct which assigns to tuples in a relation arbitrary integer tuple identifiers. The power of the construct is similar to that provided by an arbitrary ordering of the domain. If arithmetic operations on integers (such as *successor*) are also allowed, the language becomes computationally complete. The use of operations like *successor* is similar to the use of "invented" values in [5]. Indeed, both mechanisms provide an unbounded number of values not in the database which can be used throughout the computation, which results in computational completeness.

5. Expressive power

In this section, we present results of [5–7] on the expressive power of the non-deterministic languages considered in the previous section: *N-Datalog*$^{\neg *}$ and $FO + PFP + W$, then *N-Datalog*$^{\neg}$ and $FO + IFP + W$ and finally *N-Datalog**. As discussed in section 2, the most significant result involves expressibility of NDB-PTIME. Also of interest is the close connection established between the Datalog-like languages and the fixpoint logics.

The Datalog extensions and the fixpoint logics contain, roughly, two categories of languages. The first includes *N-Datalog*$^{\neg}$ and $FO + IFP + W$, and consists of languages where termination is always guaranteed in polynomial time. This is due to the increasing use of space, a consequence of the absence of deletion. Such languages are called *inflationary*. The other languages – *N-Datalog*$^{\neg *}$, $FO + PFP + W$, and *N-Datalog** – are *non-inflationary*, in that their use of space is non-increasing due to the ability to delete, or "re-use" space. Since a fixed schema is used throughout the computation and relations can only contain constants present in the input or program, the amount of space which can be built by programs in our languages is bounded by a polynomial in the size of the input. Thus, for the non-inflationary languages, the maximum expressive power which can be expected is NDB-PSPACE. On the other hand, for inflationary languages the use of space is increasing throughout the computation, so the maximum power that can be expected is NDB-PTIME. It turns out that NDB-PTIME and NDB-PSPACE can indeed be expressed using the above languages. We examine first the non-inflationary languages, then turn to the inflationary languages.

5.1. THE POWER OF NON-INFLATIONARY LANGUAGES

We consider here the expressive power of the non-inflationary languages – *N-Datalog*$^{\neg *}$, $FO + PFP + W$, and *N-Datalog**. As discussed above, NDB-

PSPACE is a bound on the expressive power of non-inflationary languages. We show that, in fact, *N-Datalog*$^{\neg *}$ and $FO + PFP + W$ compute *precisely* the NDB-PSPACE transformations. In particular, the equivalence between *N-Datalog*$^{\neg *}$ and the seemingly more powerful $FO + PFP + W$ is surprising. This extends analogous results in the deterministic case on the equivalence of *Datalog*$^{\neg}$ and fixpoint logic.

THEOREM 5.1

Let τ be a transformation from a database schema **R** to a single relation S not in **R**. The following are equivalent:
- τ is defined by an $FO + PFP + W$ formula,
- τ is the effect of an *N-Datalog*$^{\neg *}$ program,
- τ is in NDB-PSPACE. □

The proof that $FO + PFP + W$ expresses NDB-PSPACE is based on a simulation of the procedural language STL, which was defined in [4] and shown to express NDB-PSPACE. For the equivalence of $FO + PFP + W$ and *N-Datalog*$^{\neg *}$, the non-trivial part of the proof involves the simulation of $FO + PFP + W$ by *N-Datalog*$^{\neg *}$, since *N-Datalog*$^{\neg *}$ seemingly has much weaker control capability than $FO + PFP + W$. Surprisingly, it turns out that the deletions provide in fact the ability to simulate explicit control. To compute the fixpoint operator on a formula, some rules are used to "simulate" the formula that is iterated, and others for the control of the iteration. Since rules are allowed to fire at any time, there is no guarantee that the control and simulation rules fire in the intended sequence. Therefore, a journal of the "updates" is kept. If an error in the sequencing of the rules is detected, "roll-back" is possible using the journal and deletions. We will see that some difficulties arise in the analogous simulation for the inflationary case due to the lack of deletions.

Let us note again that the non-deterministic mechanisms present in the above languages allow expressing not only additional non-deterministic transformations, but also additional *deterministic* ones. For example, recall that the languages $FO + PFP$ and *Datalog*$^{\neg *}$ cannot express the parity query. However, this query is expressible in the non-deterministic counterparts of these languages. For instance, parity is expressed by the *N-Datalog*$^{\neg *}$ program:

$$odd, \neg R(x) \leftarrow R(x), \neg odd$$

$$\neg odd, \neg R(x) \leftarrow R(x), odd.$$

The issue of the deterministic transformations expressible using non-deterministic languages is examined in detail in the next section.

As discussed in the previous section, the equivalence between the languages $FO + PFP + W$ and *N-Datalog*$^{\neg *}$ yields a normal form for the language $FO + PFP + W$.

The ability to perform what amounts to deletion in the non-inflationary languages leads to the existence of non-terminating programs. We briefly address next this issue of practical interest, discussed in detail in [2,3].

In non-deterministic programs, termination comes in two flavors. It may be that, on given input, *all* computations of the program terminate. In this case the program is said to be *loop-free* (on the given input). A weaker version is that, for each input, there is some terminating computation producing an output, but there may be other computations which are non-terminating. In this case the program is called *total*. Note that loop-freedom implies totalness, but the converse is generally false. Unfortunately, static checking of totalness or loop-freedom (on all inputs) is generally unfeasible, as shown by the following.

THEOREM 5.2

It is undecidable, given a *N-Datalog*$^{\neg *}$ or $FO + W + PFP$ program, whether the program is total or loop-free. □

The language *N-Datalog** (deletions in heads but no negation in bodies) provides an interesting special case where, surprisingly, the distinction between queries and updates is crucial.

PROPOSITION 5.3

- It is decidable whether a given *N-Datalog** *query* is loop-free or total.
- It is undecidable whether a given *N-Datalog** *update* is loop-free or total. □

While static checking of termination is unfeasible in general, the situation is better for dynamic checking. First, given a *N-Datalog*$^{\neg *}$ or $FO + W + PFP$ program and an input, it is decidable whether the program is loop-free or total on that particular input. Intuitively, this is so because one need only look at computations of length exponential in the size of the input to detect a cycle. Moreover, non-termination can be detected *within the languages themselves*. More precisely, we introduce a notion of "loop-free simulation" of programs. Let Γ be a program. Let Γ' be a program using the predicates in Γ and a distinguished 0-ary predicate (not in Γ), called *defined*. Then Γ' is a loop-free simulation of Γ iff on each input I, Γ' always stops and:

- if Γ has a non-terminating computation on input I, then there is a computation of Γ' which stops with *defined* set to false,
- if Γ has a computation on input I yielding output J, then there is a computation of Γ' on input I which stops with *defined* set to true and such that the projection of the output on the predicates of Γ is J.

Furthermore, this characterizes all computations of Γ' with input over the predicates of Γ.

A program Γ has a loop-free simulation in a given language iff there is a program Γ' in that language which is a loop-free simulation for Γ.

We now have:

THEOREM 5.4

- Each *N-Datalog*$^{\neg *}$ program has a loop-free simulation in *N-Datalog*$^{\neg *}$.
- Each $FO + W + PFP$ program has a loop-free simulation in $FO + W + PFP$. □

Moreover, for each program in $FO + PFP + W$ (*N-Datalog*$^{\neg *}$), a corresponding loop-free simulation program can be constructed effectively and efficiently.

5.2. THE POWER OF INFLATIONARY LANGUAGES

We consider here the expressive power of the inflationary languages – *N-Datalog*$^{\neg}$ and $FO + IFP + W$. As discussed earlier, inflationary programs are guaranteed to terminate in polynomial time. The main result of the section is to exhibit (inflationary) languages which capture exactly the transformations in NDB-PTIME. In particular, the following theorem shows that $FO + IFP + W$ is such a language. However, the symmetry between the Datalog-like languages and the fixpoint languages breaks down in the inflationary, non-deterministic case. Indeed, we will show that *N-Datalog*$^{\neg}$ is strictly weaker than $FO + IFP + W$ but will augment the language to compensate for the loss of expressive power.

THEOREM 5.5

Let τ be a transformation from a database schema $\mathbf{R}$ to a single relation S not in $\mathbf{R}$. The following are equivalent:

- τ is in NDB-PTIME,
- τ is defined by an $FO + IFP + W$ formula. □

We now turn to *N-Datalog*$^{\neg}$. It is easy to see that each *N-Datalog*$^{\neg}$ transformation is in NDB-PTIME. It turns out that there are simple NDB-PTIME queries that cannot be expressed in *N-Datalog*$^{\neg}$. We show this next, and then show how *N-Datalog*$^{\neg}$ can be augmented to increase the expressive power to NDB-PTIME.

The strict inclusion in NDB-PTIME, is shown using the following example.

Example 5.6

Let $\mathbf{R} = \{P(A), Q(AB)\}$, $\mathbf{S} = \{S(A)\}$. Then it can be shown that there is no *N-Datalog*$^{\neg}$ program which computes $P - \pi_A(Q)$ in S. It is straightforward to obtain a formula in $FO + IFP + W$ which has this effect. □

The precise characterization of the power of *N-Datalog*$^{\neg}$ is open. We note, however, that *N-Datalog*$^{\neg}$ expresses exactly NDB-PTIME in the presence of order.

As seen in the example above, there are very simple transformations that *N-Datalog*$^{\neg}$ cannot compute. We now look at the origin of this weakness and show how it can be corrected. Note that *N-Datalog*$^{\neg}$ does not provide sufficient control capability to simulate the composition of two programs. Indeed, $P - \pi_A(Q)$ can be obtained as the composition of the mappings defined by the following two rules:

$T(x) \leftarrow Q(x, y)$, and

$S(x) \leftarrow P(x), \neg T(x)$.

The weak control capability of *N-Datalog*$^{\neg}$ makes it impossible for programs in this language to simulate the explicit control inherent in $FO + IFP + W$ and necessary to compute NDB-PTIME transformations. Note that, in the case of *N-Datalog*$^{\neg *}$, the control needed is provided by deletions. For example, the query in example 5.6 is computed by the following *N-Datalog*$^{\neg *}$ program (note that this is in fact an *N-Datalog** program):

$$\begin{array}{lcl} S(x) & \leftarrow & P(x) \\ \neg S(x), \neg P(x) & \leftarrow & Q(x, y) \end{array}$$

The constructs we add to *N-Datalog*$^{\neg}$ essentially provide sufficient control to simulate composition (in an inflationary manner). We consider two alternative constructs. The first construct allows for an "inconsistency" symbol $\perp$ to appear in heads of rules. The resulting language is denoted *N-Datalog*$^{\neg}\perp$. The idea is that if such a symbol is derived in a computation, that particular computation is abandoned. The second construct is universal quantification in bodies of rules and yields the language *N-Datalog*$^{\neg}\forall$. We first present *N-Datalog*$^{\neg}\perp$ and *N-Datalog*$^{\neg}\forall$.

N-Datalog$^{\neg}$ with inconsistency symbol: *N-Datalog*$^{\neg}\perp$
The language *N-Datalog*$^{\neg}$ is extended with the symbol $\perp$ that can occur only as a literal in the head of rules. A pair (I, J) is in the effect of a *N-Datalog*$^{\neg}\perp$ program iff J is obtained by a computation where $\perp$ is not derived.
N-Datalog$^{\neg}$ with universal quantification: *N-Datalog*$^{\neg}\forall$
The language *N-Datalog*$^{\neg}$ is extended to allow rules of the form:

$$A_1, \ldots, A_q \leftarrow \forall \vec{x} B_1, \ldots, B_n,$$

where $\vec{x}$ is a sequence of variables occurring *only* in the body of the rule. Let $\vec{y}$ be the vector of the variables occurring in $B_1, \ldots, B_n$ and not in $\vec{x}$, and v be a valuation of $\vec{y}$. The rule is fired with valuation v if for each extension $\bar{v}$ of v to the variables in $\vec{x}$ (which valuates variables in $\vec{x}$ in the active domain), $\bar{v}B_1 \wedge \ldots \wedge \bar{v}B_n$ holds.

To illustrate these two languages, we show how to compute the query of example 5.6 with *N-Datalog*$^{\neg}\forall$ or *N-Datalog*$^{\neg}\perp$ programs.

Example 5.7

The mapping $P - \pi_A(Q)$ is computed by the following *N-Datalog*$^{\neg}\forall$ program: $T(x) \leftarrow \forall y R(x), \neg Q(x, y)$.

An *N-Datalog*$^{\neg}\perp$ program computing the same query is:

$$\begin{array}{lll} PROJ(x) & \leftarrow & \neg \textit{done-with-proj}, Q(x, y) \\ \textit{done-with-proj} & \leftarrow & \\ \perp & \leftarrow & \textit{done-with-proj}, Q(x, y), \neg PROJ(x) \\ T(x) & \leftarrow & \textit{done-with-proj}, R(x), \neg PROJ(x). \end{array}$$

Intuitively, in *N-Datalog*$^{\neg}\forall$, one can check that a stage is completed (using $\forall$) before proceeding to the next one; this allows simulating composition. In *N-Datalog*$^{\neg}\perp$, a detected error leads to the derivation of $\perp$. The following shows that in fact these constructs provide sufficient power to bridge the gap between *N-Datalog*$^{\neg}$ and NDB-PTIME.

THEOREM 5.8

Let $\langle \mathbf{R}, \mathbf{S} \rangle$ be an i-o schema with **R** and **S** disjoint, and τ a query. The following are equivalent:
- τ is in NDB-PTIME,
- τ is defined by a *N-Datalog*$^{\neg}\perp$ program, and
- τ is defined by a *N-Datalog*$^{\neg}\forall$ program. □

This provides a three-way characterization of NDB-PTIME transformations: $FO + IFP + W$ (inflationary logic), *N-Datalog*$^{\neg}\forall$, and *N-Datalog*$^{\neg}\perp$.

As mentioned in section 4, the simulation of $FO + PFP + W$ by the Datalog extensions, and the converse simulations, provide a normal form for $FO + PFP + W$.

We finally consider briefly the language *N-Datalog** (no negations in bodies but deletions allowed in heads of rules). We first note the interesting fact that, despite the deletions, *N-Datalog** queries are essentially inflationary.

LEMMA 5.9

Let P be an *N-Datalog** query and I an instance over the EDB predicates of P. Let $(I_0 = I, \ldots, I_n = J)$ be a computation of P on input I reaching a fixpoint J. Then (i) for each i, $I_i \subseteq I_{i+1}$ and (ii) each computation of P on input I terminates in J. □

It should be noted that the lemma does not imply that no tuple is deleted in the computation of an *N-Datalog** query. It only indicates that computations where deletions occur never terminate. Furthermore, one can show that if an *N-Datalog** query is "loop-free" (on each input all computations terminate), then there exists an equivalent *N-Datalog* (so also an equivalent *Datalog*) query. The lemma and remarks do not extend to updates. For example, the lemma fails for the program $\neg Q(x) \leftarrow Q(x)$.

There is no precise characterization of the expressive power of *N-Datalog**. We show that *N-Datalog*$^\neg$ and *N-Datalog** are incomparable.

Example 5.10

Consider the program P:

$$\begin{aligned} R(x) &\leftarrow P(x), \neg Q(x) \\ W(x, y) &\leftarrow P(x), R(y) \end{aligned}$$

Suppose that the input is $\{P, Q\}$. One can show that no *N-Datalog** program can yield the same effect. (To see this, consider the inputs $\{P(1)\}$ and $\{P(1), Q(1)\}$). □

It is trivial to see that given an *N-Datalog** program P and a predicate Q occurring in P, then there does not always exist an *N-Datalog*$^\neg$ program which is equivalent to P for Q. (To see that, choose Q to be an input predicate and have P be the program that empties Q. With the inflationary semantics, it is not possible to invalidate an input fact.) However, *N-Datalog** is not included in *N-Datalog*$^\neg$ for more interesting reasons. Indeed, we next show that there exists an *N-Datalog** program P and a non-input predicate Q of P, such that there does not exist an *N-Datalog*$^\neg$ program which is equivalent to P for Q.

Example 5.11

Consider example 5.6. Recall that it is not possible to compute $P - \pi_A(Q)$ in S using *N-Datalog*$^\neg$. This can be done in *N-Datalog** using the simple program:

$$\begin{aligned} S(x) &\leftarrow P(x) \\ \neg S(x), \neg P(x) &\leftarrow Q(x, y) \end{aligned}$$ □

Remark

Throughout the section, the issue of expressive power was intertwined with the ability of various languages to simulate explicit control. The languages which express complexity classes of transformations are typically capable of simulating explicit control. For instance, recall that *N-Datalog*$^{\neg *}$ and *N-Datalog*$^{\neg\forall}$ are equivalent to languages with powerful explicit control (composition and iteration) such as $FO + PFP + W$ and $FO + IFP + W$. This indicates that *N-Datalog*$^{\neg *}$ and *N-Datalog*$^{\neg\forall}$ are capable of simulating explicit control such as composition and iteration of programs in the same language. Thus, if a user chooses to specify the semantics of a program using compositions Γ; Γ' and iterations $(\Gamma)^*$ of programs in the language, this can be allowed and the resulting program can still be compiled *within* the language. As suggested in [21], this allows for flexible specification of semantics. For instance, if a compiler for *N-Datalog*$^{\neg *}$ is available but the user prefers a stratified semantics involving strata $\Gamma_1, \ldots, \Gamma_n$, the user can express the query as $(\Gamma_1)^*; \ldots; (\Gamma_n)^*$ (where each Γ_i is a *N-Datalog*$^{\neg *}$ program), which can then be compiled into one *N-Datalog*$^{\neg *}$ program.

6. Connections with determinism

6.1. FUNCTIONAL FRAGMENTS

In this section we consider the ability of various non-deterministic languages to express deterministic transformations. We characterize the functional fragments of our languages. We also consider the deterministic transformations definable by considering the "possible" and "certain" answers of a non-deterministic transformation, in the spirit of queries on databases with incomplete information. The results for inflationary languages (no deletions) concern queries, while for non-inflationary languages they concern arbitrary transformations. The results are from [3].

The notion of *functional fragment* expressed by a language is defined next.

DEFINITION 6.1

The *functional fragment* of a (non-deterministic) language is the set of deterministic transformations which are effects of programs in the language. The functional fragment of a language L is denoted *funct*(L). □

In the previous section we characterized the non-deterministic transformations expressible in the various languages. These results can be used to characterize the functional fragments expressible in these languages. Thus, we have:

THEOREM 6.1

(1) funct(*N-Datalog*$^{\neg\forall}$) = *funct*(*N-Datalog*$^{\neg\perp}$) = *funct*(*FO* + *IFP* + *W*) = *DB-PTIME.*

(2) funct(*N-Datalog*$^{\neg *}$) = *funct*(*FO* + *PFP* + *W*) = *DB-PSPACE.* □

It can be shown that *funct*(*N-Datalog*$^{\neg}$) ⊂ DB-PTIME and *funct*(*N-Datalog**) ⊂ DB-PSPACE.

The above does *not* provide *languages* that express precisely *DB-PTIME, DB-PSPACE* since non-deterministic transformations can also be expressed and, as will be seen in the next subsection, it is undecidable if a program is deterministic. Instead, the result shows the power of non-deterministic constructs. Thus, augmenting a deterministic language L with the witness operator may allow expressing *more* deterministic transformations than in L. For instance, *FO* +

IFP + W can express the DB-PTIME queries, while *FO + IFP* alone cannot express the simple "parity" query.

It is of interest to understand if the increase in power is due to the witness operator alone, or if the interaction with recursion is needed. We conjecture that recursion is needed in addition to the *W* operator to obtain an increase in power with respect to the deterministic transformations computed. More precisely:

Conjecture:

$funct(FO + W) = funct(FO^+ + W) = FO.$

An alternative way of obtaining deterministic transformations using non-deterministic programs is suggested by the work of [20] on incomplete information. Indeed, there is a natural connection between incomplete information and non-determinism. As noted in [1], incomplete information can be seen as resulting from incompletely specified (therefore non-deterministic) updates. The notions of *possible* and *certain* answers in [20] suggest the following definition:

DEFINITION 6.2

Given a non-deterministic program Γ, the image of an input I under Γ with the *possibility* semantics (denoted $poss(I, \Gamma)$) and its image using the *certainty* semantics (denoted $cert(I, \Gamma)$) are defined by:

$poss(I, \Gamma) = \bigcup\{J \mid (I, J) \in eff(\Gamma)\}$ and

$cert(I, \Gamma) = \bigcap\{J \mid (I, J) \in eff(\Gamma)\}$.

The deterministic transformation expressed by a program Γ under possibility semantics is denoted $poss(\Gamma)$, and under certainty semantics $cert(\Gamma)$. For a language L, $poss(L) = \{\text{poss}(\Gamma) \mid \Gamma \in L\}$ and $cert(L) = \{\text{cert}(\Gamma) \mid \Gamma \in L\}$.

The *poss* or *cert* semantics yield significant power:

THEOREM 6.2

(*1*) $poss(FO + IFP + W) = poss(N\text{-}Datalog^{\neg}\forall) = poss(N\text{-}Datalog^{\neg}\bot) = DB\text{-}NP$.

(*2*) $cert(FO + IFP + W) = cert(N\text{-}Datalog^{\neg}\forall) = cert(N\text{-}Datalog^{\neg}\bot) = DB\text{-}coNP$.

(*3*) $cert(FO + PFP + W) = poss(FO + PFP + W) = cert(N\text{-}Datalog^{\neg *}) = poss(N\text{-}Datalog^{\neg *}) = DB\text{-}PSPACE$. □

Note that, for the non-inflationary languages (part (3)), the *poss* and *cert* semantics do not yield additional power. In particular, these semantics can be simulated within the functional fragment of each language.

It turns out that, in the inflationary case, recursion is superfluous with the powerful possibility and certainty semantics, as shown next [2)].

[2)] ∃SO and ∀SO denote the existential and universal second-order queries, known to express DB-NP and DB-coNP, respectively [16].

THEOREM 6.3

(*1*) *poss*($FO^+ + W$) = $\exists SO$ = *DB-NP*.

(*2*) *cert*($FO^+ + W$) = $\forall SO$ = *DB-coNP*. □

We illustrate by an example a simulation of existential second-order features. Consider a unary relation R and the parity query "are there an even number of tuples in R?". The query is computed by poss(Γ), where Γ is the following $FO^+ + W$ query with input R and output *even*:

$$\begin{aligned} &partition(u, z) &&= Wz(R(u) \wedge ((z = 0 \vee z = 1)));\\ &map(u, v) &&= Wu[Wv(partition(u, 1) \wedge partition(v, 0))];\\ &even &&= \forall u \exists v(map(u, v) \vee map(v, u)). \end{aligned}$$

The first statement defines a partition of R in two sets; the second defines a (partial) one-to-one mapping between the two sets; and the last checks that the mapping is total and onto, in which case the sets have equal size and $|R|$ is even.

6.2. CHECKING FUNCTIONALITY

We argued that non-deterministic programs can sometimes be used to compute more efficiently *deterministic* queries. For instance, we saw that the language $FO + IFP + W$ expresses exactly the NDB-PTIME queries. Among the queries in NDB-PTIME, some happen to be deterministic. Indeed, the deterministic queries in NDB-PTIME are exactly the queries in DB-PTIME. Such queries are computed by non-deterministic programs which happen to produce a unique output for each input. Obviously, this property is crucial if one is to use non-deterministic programs to compute deterministic queries. The property, called *functionality*, is examined in the present section (most of the results are from [2,3]). We define functionality next.

DEFINITION 6.3

Let Γ be a program and Q an output predicate of Γ. Γ is *functional with respect to* Q iff for each input of Γ, all outputs have identical projections on Q.

It is obvious that $FO + PFP + W$, $FO + IFP + W$, *N-Datalog*$^{\neg}$, and *N-Datalog*$^{\neg *}$ are generally not functional with respect to given output predicates. On the other hand, *N-Datalog* programs are obviously functional. The case of *N-Datalog** is more interesting. Once more, the distinction between queries and updates becomes essential.

PROPOSITION 6.4

(i) All *N-Datalog** queries are functional;

(ii) there exist *N-Datalog** updates that are not functional. □

Part (i) is a consequence of lemma 5.9. To see (ii), consider the *N-Datalog** update Γ where Γ consists of the single rule:

$\neg P(x, y) \leftarrow P(x, y), P(y, x)$

and the input $I = \{P(0, 1), P(1, 0)\}$. □

It turns out that, not surprisingly, functionality is undecidable for languages where it is not guaranteed. Thus, we have the following.

THEOREM 6.5

(i) It is undecidable, given an *N-Datalog*$^{\neg}$ program Γ and an output predicate Q whether Γ is functional for Q.

(ii) It is undecidable, given an *N-Datalog** update Γ, and an output predicate Q whether Γ is functional for Q. □

Note that the above result implies undecidability of functionality for the languages $FO + IFP + W$ and $FO + PFP + W$. In fact, it turns out that functionality remains undecidable even for $FO + W$ (by reduction of the implication problem for functional and inclusion dependencies, known to be undecidable [14]).

Remark

The undecidability results carry over for the stronger notion of "global" functionality. A program is globally functional if it is functional with respect to *every* predicate occurring in the program. One can also consider a notion of "uniform functionality", where all predicates are both input and output. This is similar in spirit to the notion of uniform equivalence [34]. The results of [34] suggest that uniform functionality might be decidable, but this question remains open. □

The above undecidability results show that static checking of functionality is unfeasible. It turns out that, as in the case of loop-freedom, functionality can sometimes be checked *dynamically*. More precisely, we shall say that functionality can be detected dynamically within a language L iff for each program Γ in L and output predicate Q of Γ, there exists a program Γ' in L such that:

- the input predicates of Γ and Γ' are the same, and the output predicates of Γ' are those of Γ together with a 0-ary predicate (not in Γ), called *functional*;
- on each input I on which Γ is not functional (with respect to Q), Γ' always stops with the value of *functional* set to false;
- on each input I on which Γ is functional (with respect to Q), all computations of Γ' produce the same results as Γ (on the common output predicates) and the value of *functional* is set to true;

- if Γ has only terminating computation on input I, then Γ' has only terminating computations on input I.

We now have:

THEOREM 6.6

Functionality can be checked dynamically within the languages *N-Datalog*$^{\neg *}$ and $FO + PFP + W$.

Proof (sketch)

Clearly, a program is functional with respect to Q on a given input I iff the union and intersection of all possible outputs (on Q) coincide. Thus, it is sufficient to compute the union and intersection of possible outputs (over Q). It is easily seen that the membership problem for the union and intersection of the outputs is in NPSPACE and co-NPSPACE, respectively. By Savitch's theorem, both problems are in PSPACE. Since the above languages can express all DB-PSPACE queries, it can be seen that they are able to compute the union and intersection of outputs, and thus check functionality. □

It turns out that functionality cannot be checked dynamically within the other languages considered here. The argument for *N-Datalog*$^{\neg}\perp$, *N-Datalog*$^{\neg}\forall$, $FO + IFP + W$ programs and *N-Datalog* updates* is based on complexity. Indeed, it can be shown that checking functionality dynamically within each of these languages is NP-hard and in DP. In fact, the problem is NP-hard even for $FO^{+} + W$ (by reduction of 3-colorability of a graph)! On the other hand, the inflationary languages stay within polynomial time, and the language *N-Datalog** is also in polynomial time in view of lemma 5.9. This makes it highly unlikely that checking functionality is possible. For the language *N-Datalog*$^{\neg}$, there is a direct proof that functionality cannot be checked dynamically. The proof technique is the same as the one used for showing that the query in example 5.6 is not expressible in the language.

6.3. DETERMINISTIC VS. NON-DETERMINISTIC SEMANTICS

In the previous section we looked at functionality, which is the property of a non-deterministic program of computing unique results for given inputs. Note that, although such programs compute deterministic transformations, it is not necessarily true that the non-deterministic semantics coincides with the deterministic semantics. The latter property is nonetheless interesting for different reasons related to optimization. Indeed, as discussed in [35], implementing a non-deterministic program with deterministic semantics allows more efficient processing of the query using parallelism. This is due to the fact that several instantiations of rules can be "fired" in parallel without changing the final result. In this section we look at when the non-deterministic semantics of a program in the Datalog-like languages coincides with the deterministic semantics. We are concerned with programs that can be assigned both a deterministic and a non-deterministic semantics, i.e. the rules have single-literal heads, and the equality predicate is not used. The coincidence of semantics is defined next relative to a predicate. The results are from [2,3].

DEFINITION 6.4

Let Γ be a program in both *N-Datalog*$^{(\neg)(*)}$ and *Datalog*$^{(\neg)(*)}$ and an output predicate Q. The deterministic and non-deterministic semantics of Γ with respect to Q coincide iff for each input I of Γ the projection on Q of the output under the deterministic semantics equals the projection on Q of *every* output under the non-deterministic semantics. □

We first exhibit an example which shows that functionality and coincidence of deterministic and non-deterministic semantics are distinct properties.

Example 6.7

Consider the program:

$$\begin{array}{lll} A & \leftarrow & \neg A, \neg B \\ B & \leftarrow & \neg A, \neg B \\ C & \leftarrow & A, \neg B \\ C & \leftarrow & \neg A, B \\ A & \leftarrow & C \\ B & \leftarrow & C. \end{array}$$

Note first that the program, with non-deterministic semantics, is functional for all predicates. However the deterministic and non-deterministic semantics are different (C becomes true with non-deterministic semantics and stays false with deterministic semantics, assuming that all predicates are initially false). □

We now look at the coincidence of deterministic and non-deterministic semantics. Obviously, for programs without negation, the two semantics coincide. For *Datalog*$^\neg$ queries, the above example shows that the two semantics may be distinct. As shown next, one cannot decide, for *Datalog*$^\neg$ and *N-Datalog*$^\neg$ queries, whether the deterministic and non-deterministic semantics coincide.

THEOREM 6.8

It is undecidable whether for a given program Γ in both *N-Datalog*$^\neg$ and *Datalog*$^\neg$ and an output predicate Q, the deterministic and non-deterministic semantics of Γ coincide for Q. □

Let us now consider again the special case of *N-Datalog**. In this case again, the deterministic and non-deterministic semantics may differ, as shown by the following:

Example 6.9

Consider the query Γ consisting of the rules:

$\neg Q \leftarrow A, \quad Q \leftarrow A, \quad \neg A \leftarrow Q.$

With non-deterministic semantics, the program is functional and Q is derived (Q is initially false and A true). With deterministic semantics, Q is never derived. □

The decidability issue for *N-Datalog** highlights again the difference between queries and updates. Indeed, we have:

THEOREM 6.10

(i) It is decidable whether, for a given *N-Datalog** query and ouput predicate, the deterministic and non-deterministic semantics coincide with respect to the output predicate.

(ii) It is undecidable whether, for a given *N-Datalog** update and ouput predicate, the deterministic and non-deterministic semantics coincide with respect to the output predicate.

7. Conclusion

We motivated and studied the use of non-determinism in logic-based languages, primarily database queries and updates. We argued that non-determinism arises naturally in various circumstances such as:

- incomplete specification of queries or updates,
- viewing a given query or update at different levels of abstraction.

Allowing incomplete specification of queries or updates is sometimes simply a matter of convenience for the user (remember the Starving Frenchman!). More importantly, there are natural queries or updates which *require* non-deterministic implementations because they are not generic, so they cannot be computed by any deterministic program (recall example 4.3 on computing an orientation of a graph). In the context of logic programming, it was suggested that some model-based semantics are naturally non-deterministic, since they place on equal footing several models with certain properties, each of which violates genericity and so cannot be selected deterministically (see example 4.8 based on [33]).

When a transformation requires a non-deterministic choice, this is due to the fact that not enough information is available to make a deterministic choice. This can sometimes be compensated by viewing the transformation at a lower level of abstraction, where more information may be available. Thus, there is an intimate connection between non-determinism and abstraction: a query which is non-deterministic at one level of abstraction may become deterministic if viewed at a lower level of abstraction. This may occur in the context of computation connected to a type hierarchy. It also suggests a connection between non-determinism and the data-independence principle. This is captured in the meta-theorem relating expressiveness by deterministic languages in the presence of order to expressiveness by non-deterministic languages (section 3). In particular, queries expressible by a deterministic language at one level of abstraction may not be expressible in the same language when viewed at a higher level of abstraction, but may instead be expressible by a *non-deterministic* variation of the language. As a case in point, the parity query cannot be expressed by the fixpoint queries (e.g. $FO + IFP$), but can be expressed either by the fixpoint queries in the presence of order (which amounts to changing the level of abstraction) or by a non-deterministic extension of the fixpoint queries (e.g., $FO + IFP + W$). These trade-offs extend to all the transformations computable in polynomial time.

We identified a trade-off between determinism and expressive power. In particular, adding non-deterministic constructs to a language may allow computing more *deterministic* transformations than in the original language. Indeed, we showed that non-deterministic constructs allow expressing low complexity classes of deterministic transformations, whereas it is conjectured that no deterministic languages expressing such "nice" classes exist. We characterized the deterministic transformations expressible in various languages (their *functional fragments*). We exhibited languages whose functional fragments are DB-PTIME and DB-PSPACE. It would also be of interest to exhibit languages with functional fragments of lower complexity.

One can also view non-deterministic transformations as incompletely specified transformations leading to databases with incomplete information. We considered deterministic semantics for such transformations in the spirit of *possible* and *certain* answer semantics to queries on databases with incomplete information. However, the classes of deterministic transformations expressed with possibility and certainty semantics have relatively high complexity (DB-NP and DB-coNP). It appears that the possibility and certainty semantics dominate the iteration and fixpoint constructs with respect to expressive power (see theorem 6.3).

We considered several issues of practical interest arising from the use of non-deterministic programs to compute deterministic transformations, such as checking if a program is deterministic and verifying termination. The results showed that static checking of such properties is generally impossible, but dynamic checking is feasible in some of the languages. Some of the results yielded surprising distinctions between queries and updates.

Two families of non-deterministic languages were emphasized here: Datalog-like languages with fixpoint semantics, and non-deterministic extensions of fixpoint logic using the witness operator W. The properties of logics with the W operator are not yet completely understood. (For instance no axiomatization is known for $FO + W$, if indeed one exists.) Strong connections were established among the two families of languages. On the one hand, the equivalence of Datalog-like languages with the W-extensions of the fixpoint logics shows that the complex fixpoint logics, which provide ample explicit control, have simple normal forms (see proposition 4.4). On the other hand, it shows that some of the Datalog-like languages are capable of simulating explicit control, which allows for more flexible specification of program semantics within each language, using program composition and iteration (see remark, section 5).

Optimization issues in the presence of non-determinism remain largely unexplored. We noted that coincidence of non-deterministic and deterministic semantics for Datalog-like programs (or individual rules) provides one source of optimization. However, non-determinism itself provides a potential source of optimization by allowing a certain degree of freedom in the computation. Only preliminary investigations exist on such optimization (e.g., [28]). This freedom of choice could also be exploited in concurrency control.

Acknowledgement

The authors wish to thank Eric Simon for useful discussions on the material presented here.

References

[1] S. Abiteboul and G. Grahne, Update semantics for incomplete databases, *Int. Conf. on Very Large Data Bases* (1985) 1–12.

[2] S. Abiteboul and E. Simon, Fundamental properties of deterministic and non-deterministic extensions of Datalog, submitted to Theor. Comp. Sci.

[3] S. Abiteboul, E. Simon and V. Vianu, Non-deterministic languages to express deterministic transformations, *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems* (1990) pp. 218–229.

[4] S. Abiteboul and V. Vianu, Procedural languages for database queries and updates, J. Comp. Syst. Sci. 41 (2) (1990).

[5] S. Abiteboul and V. Vianu, Procedural and declarative database update languages, *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems* (1988) pp. 240–250.

[6] S. Abiteboul and V. Vianu, Datalog extensions for database queries and updates, INRIA Technical Report No.715 (1988), to appear in J. Comp. Syst. Sci.

[7] S. Abiteboul and V. Vianu, Fixpoint extensions of first-order logic and Datalog-like languages, *Proc. Symp. on Logic in Computer Science* (1989) pp. 71–79.

[8] S. Abiteboul and V. Vianu, A transaction-based approach to relational database specification, J. ACM 36(4) (1989) 758–789.

[9] S. Abiteboul and V. Vianu, The connection of static constraints with determinism and boundedness of dynamic specifications, *Proc. 3rd Int. Conf. on Data and Knowledge Bases* (Morgan Kaufmann, 1988) pp. 324–334.

[10] L. Brownston, R. Farrel, E. Kant and N. Martin, *Programming Expert Systems in OPS5* (Addison–Wesley, 1985).

[11] A.K. Chandra, Programming primitives for database languages, *Proc. ACM Symp. on Principles of Programming Languages,* Williamsburg (1981) pp. 50–62.

[12] A.K. Chandra and D. Harel, Computable queries for relational databases, J. Comp. Syst. Sci. 21(2) (1980) 156–178.

[13] A.K. Chandra and D. Harel, Structure and complexity of relational queries, J. Comp. Syst. Sci. 25(1) (1982) 99–128.

[14] A. Chandra and M. Vardi, The implication problem for functional and inclusion dependencies is undecidable, SIAM J. Comp. 14(3) (1985) 671–677.
[15] E. Dahlhaus, *Skolem Normal Forms Concerning the Least Fixpoint, Computation and Logic,* Lecture Notes in Computer Science (Springer Verlag, 1987).
[16] R. Fagin, Generalized first-order spectra and polynomial-time recognizable sets, *Complexity of Computation*, ed. R. Karp, SIAM-AMS Proc. 7 (1974) pp. 43–73.
[17] L. Farinas and A. Herzig, Reasoning about database updates, *Workshop on Foundations of Logic Programming and Deductive Databases,* J. Minker (ed.) (1986).
[18] Y. Gurevich, Logic and the challenge of computer science, in: *Trends in Theoretical Computer Science*, E. Borger (ed.) (Computer Science Press, New York, 1988) pp. 1–57.
[19] Y. Gurevich and S. Shelah, Fixed-point extensions of first-order logic, *26th Symp. on Foundations of Computer Science* (1985) pp. 346–353.
[20] T. Imielinski and W. Lipski, Incomplete information in relational databases, J. ACM 31(4) (1984) 761–791.
[21] T. Imielinski and S. Naqvi, Explicit control of logic programs through rule algebra, *Proc. ACM SIGACT-SIGART-SIGMOD Symp. on Principles of Database Systems* (1988) pp. 103–116.
[22] N. Immerman, Relational queries computable in polynomial time, Inf. Control 68 (1986) 86–104.
[23] N. Immerman, Languages which capture complexity classes, SIAM J. Comp. 16(4) (1987) 760–778.
[24] P.C. Kanellakis, Elements of relational database theory, to appear as a chapter in: *Handbook of Theoretical Computer Science* (North-Holland).
[25] J. de Kleer, An assumption-based truth maintenance system, Art. Int. 28 (1) (1986) 127–162.
[26] KEE Reference Manual, release 3.0, Intellicorp (1986).
[27] P. Kolaitis, The expressive power of stratified logic programs, to appear in Inf. Comp.
[28] S. Naqvi and R. Krishnamurthy, Non-deterministic choice in Datalog, *Proc. 3rd Int. Conf. on Data and Knowledge Bases* (Morgan Kaufmann, Los Altos, 1988) pp. 416–424.
[29] A.C. Leisenring, *Mathematical Logic and Hilbert's ϵ-symbol.* (Gordon and Breach, 1969).
[30] C. de Maindreville and E. Simon, Modelling non-deterministic queries and updates in deductive databases, *Proc. Int. Conf. on Very Large Databases*, Los Angeles (1988) pp. 395–407.
[31] S. Manchanda and D.S. Warren, A logic-based language for database updates, in: *Foundations of Logic Programming and Deductive Databases*, ed. J. Minker (1987).
[32] S. Naqvi and S. Tsur, *A Logical Language for Data and Knowledge Bases* (Computer Science Press, New York, 1989).
[33] D. Sacca and C. Zaniolo, Stable models and non-determinism in logic programs with negation, *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems* (1990) pp. 205–217.
[34] Y. Sagiv, Optimizing datalog programs, *Proc. 6th ACM Symp. on Principles of Database Systems* (1987) pp. 349–362.
[35] E. Simon and C. de Maindreville, Deciding whether a production rule is relationally computable, *Proc. 2nd Int. Conf. on Database Theory,* Bruges, Belgium (1988) pp. 205–222.
[36] Y. Shen, IDLOG: Extending the expressive power of deductive database languages, *Proc. ACM SIGMOD Int. Conf. on Management of Data* (1990) pp. 54–63.
[37] J.D. Ullman, *Principles of Database and Knowledge Base Systems* (Computer Science Press, New York, 1988).
[38] M. Vardi, Relational queries computable in polynomial time, *14th ACM Symp. on Theory of Computing* (1982) pp. 137–146.
[39] D.S. Warren, Database updates in pure Prolog, *Proc. Int. Conf. on Fifth Generation Computer Systems* (1984).