DATALOG Queries ⋆ with Stratified Negation and Choice: from $\mathcal{P}$ to $\mathcal{D}^P$

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

[1] DEIS, Univ. della Calabria, 87030 Rende, Italy
{ greco, sacca }@si.deis.unical.it
[2] Computer Science Dept., Univ. of California, Los Angeles, CA 90024
zaniolo@cs.ucla.edu

Abstract. This paper introduces a unified solution to the problem of extending stratified DATALOG to express DB-complexity classes ranging from $\mathcal{P}$ to $\mathcal{D}^P$. The solution is based on (i) stratified negation as the core of a simple, declarative semantics for negation, (ii) the use of a "choice" construct to capture non-determinism of stable models (iii) the ability to bind a query execution to the complexity class that includes the problem at hand, and (iv) a general algorithm that ensures efficient execution for the different complexity classes. We thus obtain a class of DATALOG programs that preserves computational tractability, while achieving completeness for a wide range of complexity classes.

1 Introduction

The issue of designing declarative, logic-oriented database languages with sufficient expressive power is the key motivation of much of current research on databases and knowledge bases. The introduction of DATALOG represents a major breakthrough in this line of work, due to DATALOG's ability of expressiving recursive queries. DATALOG is a rule-based language that has simple and elegant semantics based on the notion of minimal model—or equivalently, on the notion of least fixpoint. This second semantics leads to an operational semantics that is amenable to very efficient implementation as demonstrated by recent work on deductive database systems.

Unfortunately, the basic DATALOG language (no negation or function symbols) is still severely limited in its expressive power and cannot express many of the queries of practical interest. The exact expressive power of DATALOG is not yet understood but it has been shown that DATALOG can only express a proper subset of the monotonic polynomial-time queries [3].

In order to support non-monotonic queries, negation is allowed in the bodies of the rules. Of a particular interest is *stratified negation*, which avoids the semantic and implementation problems connected with the use of non-monotonic

⋆ The work of the first two authors has been supported by the CNR project "Sistemi Informatici e Calcolo Parallelo" and by the MURST project "Metodi Formali per Basi di Dati".

constructs in recursive definitions [4, 7, 29]. We will write $\texttt{DATALOG}^{\neg s}$ to denote $\texttt{DATALOG}$ with stratified negation. Simple, intuitive semantics, leading to efficient implementation exists for $\texttt{DATALOG}^{\neg s}$. Unfortunately, $\texttt{DATALOG}^{\neg s}$ cannot express all polynomial-time queries, and can only express a proper a proper subset of fixpoint queries [15].

The simplest step toward greater expressive power is to remove the condition that negation must be stratified. Unfortunately, the relaxation of the stratification assumption for non monotonic programs opens a pandora box of semantic and computational problems. Take for instance the concepts of *stable models* [9] and *well-founded models* [30]. The well-founded model semantics defines a unique "intended" model for each program, which can be computed in polynomial time. Several programs, however, do not have a well-founded total model: the meaning of these programs, therefore, remain undefined. In terms of expressibility, well-founded semantics can express all fixpoint queries, but neither can express all polynomial time queries, nor it can express queries in DB-$\mathcal{NP}$.

A dramatic leap in expressive power is provided by the concept of stable models, which has emerged as the compendium of many concepts and theories developed over the years by AI researches working on non-monotonic reasoning, default theories and autoepistemic logic. This gain in expressive power, however, is not without complications. One is the non-deterministic nature of such semanatics that follows from the fact that a program can have several stable models. Nevertheless, deterministic query semantics can be ensured by requesting answers that hold true for *some* stable model (*possibility semantics*), or for *all* stable models, (*certainty semantics)*; these are known in AI as membership and entailment semantics, respectively. The two semantics capture the classes $\mathcal{NP}$ and co$\mathcal{NP}$, respectively. However, there remain the following two problems:

1. The usage of unrestricted negation in programs is often not simple nor intuitive, and, e.g., might lead to the writing of programs for which no total stable model occurs (a similar problem also exists for for well-founded models)
2. exponential-time algorithms are needed to compute stable models (as expected in situations where programs express NP-hard problems). Unfortunately, we do not know how to ensure that these exponential algorithms compute efficiently on programs that solve polynomial-time problems.

With current research focusing on overcoming these limitations, several proposals have been put forward that give up declarative semantics and fall-back on procedural semantics, e.g., based on the inflactionary fixpoint computation procedure [2, 1, 16]. This paper is largely inspired by the conviction that model-theoretic semantics offers important benefits, and, therefore, we should strive to preserve it. Thus, we propose a novel approach whereby stratified $\texttt{DATALOG}$ programs are extended to include (1) an explicit non-deterministic construct called *choice* whose semantics is defined in terms of stable models but can be implemented very efficiently, and (2) the use of quantifiers in the query goals to achieve the desired levels of expressive power and computational efficiency in a more controlled fashion, that allows a user to match the expressive power of the semantic used with the intrinsic complexity of the problem at hand.

This paper proposes an approach that allows users to express easily and compute efficiently problems ranging from the (whole) class $\mathcal{P}$ to the classes $\mathcal{NP}$ and co$\mathcal{NP}$. In this way, the ability of expressing $\mathcal{NP}$-hard problems is not at the cost of gross inefficiencies in solving polynomial problems. Since many practical problems fall in the class $\mathcal{D}^p$[22], we also introduce the capability of capturing this class. Furthermore, the paper presents of a unifying algorithm that automatically adapts to the complexity of the problem at hand. The algorithm shows how to make stable model semantics amenable to effective implementations. In summary, the paper presents a simple framework for exploiting recent advances in in model-theoretic semantics for non-monotonic `DATALOG`, without surrendering naturalness and efficiency of stratified negation.

2 Preliminary Definitions

We assume the reader is familiar with the concepts of relational databases and of the `DATALOG` language [14, 28] as well as of logic programming [17].

The semantics of logic programs is given in terms of total stable model semantics [9] which we briefly recall next.

Given a logic program P and an interpretation M (i.e., a subset of the Herbrand base), M is a (*total*) *stable model* of P if it is the minimal model of the program $gl(P)$ defined as follows: $gl(P)$ is obtained from the ground instance of P by (i) deleting all rules of which have some negative literal $\neg b$ in their body with $b \in M$, and (ii) removing all negative literals in the bodies of the remaining rules.

A logic program may have 0 or one or several stable models and deciding whether it admits at least one stable model is $\mathcal{NP}$-complete [18].

Positive (i.e., negation free) programs have a unique stable model, corresponding to the minimum model.

Given a program P and two predicate symbols p and q, we write $p \to q$ if there exists a rule where q occurs in the head and p in the body or there exists a predicate s such that $p \to s$ and $s \to q$. If $p \to q$ then we say that q *depends on* p; also we say that q *depends on* any rule where p occurs in the head.

A program is *stratified* if there exists no rule r where a predicate p occurs in a negative literal in the body, q occurs in the head and $q \to p$, i.e., there is no recursion through negation. Also stratified programs have a unique stable model, corresponding to the stratified fixpoint.

A $\mathtt{DATALOG}^{\neg}$ program P is a function-free logic program whose predicates are partitioned into *extensional* and *intensional* predicates (EDB and IDB predicates, respectively). A $\mathtt{DATALOG}^{\neg}$ program P has associated a relational database scheme $DB_P = \{r \mid r \text{ is an EDB predicate symbol of } P\}$. EDB predicates are seen as database relation schemes, whose attribute domain is a countable set U, and they never occur in the rule heads as they are assumed to be defined by some database. A database D on a scheme DB_P is a set of finite relations, one for each EDB predicates in P. The relation associated with an EDB predicate r is denoted $D(r)$. Thus, any (finite) database D over DB_P is considered as a set of facts $\{r(t)|t$ is a tuple in some relation r of $D\}$. The set of all databases on DB_P is denoted $\mathbf{D}_P$.

We assume that the semantics of a program P applied to database D is given by the stable models of $P \cup D$.

If a $\mathtt{DATALOG}^\neg$ program P is positive then it is simply called a $\mathtt{DATALOG}$ program and, then, the unique stable model of P applied to a database D can be computed in time polynomial in the size of D. We point out the we are referring to the so-called *data complexity* [31] for which the size of P is assumed to be a constant.

If a $\mathtt{DATALOG}^\neg$ program is stratified then it is called a $\mathtt{DATALOG}^{\neg s}$ *program*. The unique stable model of a $\mathtt{DATALOG}^{\neg s}$ program P applied to a database D can be computed in time polynomial in the size of D.

A (*ground*) $\mathtt{DATALOG}^\neg$ *query* Q is a pair (G, P) where G is a conjunction of ground literals, called *query goal*, and P is a $\mathtt{DATALOG}^\neg$ program. $\mathbf{Q}^\neg$ denotes the set of all possible $\mathtt{DATALOG}^\neg$ queries, i.e., all possible query goals on all possible $\mathtt{DATALOG}^\neg$ programs.

3 Semantics and Complexity of $\mathtt{DATALOG}^\neg$

We assume that the reader is familiar with the basic notions of complexity classes [13, 21] and of query language complexity evaluation (see, for instance, [1, 2, 6, 12, 14, 15, 16, 27, 31]).

The *answer* to a query Q on a database D over DB_P is defined as follows:

1. under the *non-deterministic semantics* ($Q_!$):,
   (a) *true* if there is a stable model M of $P \cup D$ for which G is true in M
   (b) *false* if either there exists no stable model or there is a stable model M of $P \cup D$ for which G is false in M

   i.e., for some queries both a *true* and *false* answer is correct.
2. under the *possibility semantics* ($Q_\exists$):
   (a) *true* if there is a stable model M of $P \cup D$ for which G is true in M, and
   (b) *false* otherwise;
3. under the *certainty semantics* ($Q_\forall$):
   (a) *true* if for each stable model M of $P \cup D$, G is true in M, and
   (b) *false* otherwise;

A query $Q = (G, P)$ on a database D under semantics T is denoted by Q_T or, equivalently, by $(T(G), P)$, where the subscripts !, $\forall$, $\exists$, respectively, stand for non-deterministic semantics, possibility semantics and certainty semantics. The answer to a query Q on a database D under semantics T is denoted by $Q_T(D)$ or equivalently by $(T(G), P)(D)$.

Any query $Q = (G, P)$ in $\mathbf{Q}^\neg$ can be considered as a function from the collection $\mathbf{D}$ of all databases over DB_P to $\{true, false\}$. It turns out that we can describe the complexity of a query under a given semantics as the complexity of the recognizing whether a given database input has a *true* answer or not under that semantics.

The expressive power of the above semantics is given by the set of all the functions that can be formulated by all possible queries and will be denoted by $EXPR_T(\mathbf{Q}^{\neg})$. To measure expressive powers, for each Turing Machine complexity class C, there is a corresponding class $DB\text{-}C$ denoting the class of all the above functions for which the recognition problem is in C. In particular, we have the class $DB\text{-}\mathcal{P}$, $DB\text{-}\mathcal{NP}$, $DB\text{-co}\mathcal{NP}$and $DB\text{-}\mathcal{D}^p$.

Under the non-deterministic semantics, the answer to a query can be both $true$ and $false$, therefore, queries cannot be considered as functions. However, the only bound queries of practical interest have only one answer; i.e., these queries are actually deterministic, and non-determinism is only used to increase the expressive power and/or to simplify the formulation. (For instance, to test whether a relation has an even number of tuples, we can write a DATALOG$^{\neg}$ program which non-deterministically partitions the tuples into a number of pairs; there will then be as many stable models as the number of 2-combinations of all tuples but the result will be the same no matter which stable model is selected.) Therefore, from now on, we shall only consider deterministic queries. Thus, under the non-deterministic semantics $\mathbf{Q}^{\neg}$ will actually denote only the set of all possible DATALOG$^{\neg}$ queries which are deterministic.

Observe that the problem of whether a query on a given database is deterministic under the non-deterministic semantics is in $\mathcal{D}^p$; worse, the problem of whether a query is deterministic for all databases is not decidable at all. In our view, this is not a real problem as we assume that the programmer has the responsibility to write deterministic queries in the same way as it is his/her responsibility to write a correct (and terminating) program. If the query turns out not to be deterministic, he/she will eventually get an answer that is not be appropriate as for the case of wrong programs.

Because of the above restriction, queries can be considered as functions also under the non-deterministic semantics and also their expressive power can be then expressed in terms of the above DB complexity classes.

The following results are known in the literature.

Fact 1 *(see [18, 26])*

1. $EXPR_{\exists}(\mathbf{Q}^{\neg}) = DB\text{-}\mathcal{NP}$;
2. $EXPR_{\forall}(\mathbf{Q}^{\neg}) = DB\text{-}co\mathcal{NP}$ □

Let us now denote the set of all DATALOG queries by $\mathbf{Q}^{+}$ and the set of all DATALOG$^{\neg s}$ queries by $\mathbf{Q}^{\neg s}$. Any query in $\mathbf{Q}^{+}$ and in $\mathbf{Q}^{\neg s}$ is deterministic.

Fact 2 *(see [3, 20, 15])*

$$EXPR_F(\mathbf{Q}^{+}) \subset EXPR_F(\mathbf{Q}^{\neg s}) \subset DB\text{-}\mathcal{P}$$

where F is any of the three types of semantics. □

Next we show the expressive power of non-deterministic semantics for deterministic queries.

Proposition 1. $EXPR_!(\mathbf{Q}^\neg) = DB\text{-}\mathcal{NP}$.
Proof. (Membership to $DB\text{-}\mathcal{NP}$.) Take any query $Q = (G, P)$ in $\mathbf{Q}^\neg$ and a database D over DB_P; we want to recognize whether $Q_!(D)$ is *true*. To this end we just guess an interpretation M of $P \cup D$ and test in (deterministic) polynomial time whether (i) M is a total stable model and (ii) G is in M. To verify that M is a stable model can be done in polynomial time as we only need to compute the least fixpoint of $gl(P)$ and test whether it coincides with M.
(Completeness.) We use Fagin's result [8] (see also [13, 21]) that every $\mathcal{NP}$ recognizable database collection is defined by an existential second order formula $\exists R \Phi$, where R is a list of new predicate symbols and Φ is a first-order formula involving predicate symbols in a database scheme Z and in R. As shown in [16], this formula is equivalent to one of the form (*second order Skolem normal form*)

$$(\exists S)(\forall X)(\exists Y)(\theta_1(X,Y) \vee \ldots \vee \theta_k(X,Y))$$

where S is a superlist of R, $\theta_1, ..., \theta_k$ are conjunctions of literals involving variables in X and Y, and predicate symbols in S and DB_P. Consider the following program P:

$$\begin{array}{lll}
s_j(W_j) & \leftarrow \neg \hat{s}_j(W_j). & (\forall s_j \in S) \\
\hat{s}_j(W_j) & \leftarrow \neg s_j(W_j). & (\forall s_j \in S) \\
q(X) & \leftarrow \theta_i(X,Y). & (1 \le i \le k) \\
g & \leftarrow \neg q(X). & \\
p & \leftarrow g,\ \neg p &
\end{array}$$

The first two groups of rules define an instance of the predicates in S. Therefore, the first three groups of rules implement the above second order formula. The fourth rule checks if there is some X for which the formula is not satisfied. The fifth rule force the global program to have a total stable model only if g is false. Note that, if g is true the program does not have total stable model. Thus, the program P has a stable model and then the query goal $\neg p$ is true under the non-deterministic semantics iff a database belongs to the collection. □

Observe that the restriction that $\mathbf{Q}^\neg$ only contains deterministic queries under the non deterministic semantics is not necessary for Proposition 1 to hold. Notice also that the expressive power of the non-deterministic semantics coincides with that of possibility semantics.

In order to provide more flexibility in expressing queries (and without any additional cost as we shall discuss later), we allow to combine possibility and certainty semantics. Thus, we write *compound* query goals having the following form $\exists(G_1) \wedge \forall(G_2)$, where G_1 and G_2, are related in useful ways to be discussed later. In particular, if $G_1 = G_2$, then the program must have at least one stable model and G_1 must hold true in each such a model.

Proposition 2. $EXPR_{\exists\forall}(\mathbf{Q}^\neg) = DB\text{-}\mathcal{D}^p$.
Proof. A compound query corresponds to the conjunction of a query under the possibility semantics and a query under the certainty semantics. The result follows from the fact that possibility semantics captures $\mathcal{NP}$and certainty semantics captures co$\mathcal{NP}$, □

4 Choice and Stratification in DATALOG Queries

The *choice* construct (supported in $\mathcal{LDL}$++ [19] and, in some form, in Coral [23]), is used to enforce functional constraints on rules of a logic program. Thus, a goal of the form, `choice((X),(Y))`, in a rule r denotes that the set of all consequences derived from r must respect the FD $X \rightarrow Y$. In general, X can be a vector of variables—possibly an empty one denoted by "`( )`" —and Y is a vector of one or more variables.

As shown in [25] the formal semantics of the construct can be given in terms of stable model semantics as follows. A rule of the form

$$p(X,Y,W) \leftarrow q(X,Y,Z,W),\ \mathit{choice}((X),(Y)),\ \mathit{choice}((Y),(Z))$$

is rewritten as:

$$\begin{array}{ll} p(X,Y,W) & \leftarrow q(X,Y,Z,W),\ \mathit{chosen}(X,Y,Z). \\ \mathit{chosen}(X,Y,Z) & \leftarrow q(X,Y,Z),\ \neg \mathit{diffchoice}(X,Y,Z). \\ \mathit{diffchoice}(X,Y,Z) & \leftarrow \mathit{chosen}(X,Y',Z'),\ Y \neq Y'. \\ \mathit{diffchoice}(X,Y,Z) & \leftarrow \mathit{chosen}(X',Y,Z'),\ Z \neq Z'. \end{array}$$

Let P be a $\mathtt{DATALOG}^{\neg}$ program with choice constructs. If P is stratified modulo choice (i.e., by removing the choice predicates) the stable model is not in general unique but existence of at least one is guaranteed. (This result remains valid also when choice predicates occur in recursive rules [10].) Moreover, if P is a stratified modulo choice program then one of the stable models can be computed in polynomial time [25, 10, 11].

Let us now study the expressive power of queries on $\mathtt{DATALOG}^{\neg s,c}$ programs, i.e., $\mathtt{DATALOG}^{\neg s}$ programs with choice constructs. Let $\mathbf{Q}^{\neg s,c}$ denote the set of all queries on $\mathtt{DATALOG}^{\neg s,c}$ programs. Again we restrict our attention to deterministic queries; therefore, under the non-deterministic semantics, $\mathbf{Q}^{\neg s,c}$ is actually only a subset of all possible queries.

Theorem 1.

1. $EXPR_{!}(\mathbf{Q}^{\neg s,c}) = DB\text{-}\mathcal{P}$.
2. $EXPR_{\exists}(\mathbf{Q}^{\neg s,c}) = DB\text{-}\mathcal{NP}$.
3. $EXPR_{\forall}(\mathbf{Q}^{\neg s,c}) = DB\text{-}co\mathcal{NP}$.
4. $EXPR_{\exists\forall}(\mathbf{Q}^{\neg s,c}) = DB\text{-}\mathcal{D}^{p}$.

Proof. (1) Papadimitriou has shown that stratified negation (actually a reduced version of it) plus an ordering on the domain captures $\mathcal{P}$. Computing a stable model of any query in $\mathbf{Q}^{\neg s,c}$ over any database D can be done in polynomial time since it is sufficient to extend the stratified fixpoint with a simple polynomial-time implementation of the choice construct. Since $\mathbf{Q}^{\neg s,c}$ only contains deterministic queries, the answer does not depend on which stable model is returned. Hence the expressive power cannot go beyond $\mathcal{P}$. Thus, It is sufficient to introduce an ordering on the domain and to be able to find the minimal and the maximal element in the domain. This can be done using choice and stratified negation, as reported in the following program:

$$
\begin{array}{lll}
min(X) & \leftarrow domain(X),\ choice((),(X)). \\
succ(X,Y) & \leftarrow succ(\_,X),\ min(Z),\ domain(Y),\ Y \neq Z, \\
 & \quad choice((X),(Y)),\ choice((Y),(X)). \\
max(X) & \leftarrow domain(X),\ \neg succ(X,\_)
\end{array}
$$

(2) We follow the lines of the proof of Proposition 1 but this time we use a $\mathtt{DATALOG}^{\neg s,c}$ program P defined in the following way:

$$
\begin{array}{lll}
label(1). & & \\
label(2). & & \\
\hat{s}_j(W_j,K) \leftarrow label(K),\ choice(W_j,K). & & (\forall s_j \in S) \\
s_j(W_j) & \leftarrow \hat{s}_j(W_j,1). & (\forall s_j \in S) \\
q(X) & \leftarrow \theta_i(X,Y). & (1 \leq i \leq k) \\
g & \leftarrow \neg q(X). &
\end{array}
$$

It is easy to see that, given the query goal $\neg g$, the query on a database is true under the possibility semantics iff the database belongs to the collection.
(3) Consider the program P in the above proof of Part (2). This time it is required to check whether a database does not to belong to the collection. This happens iff given the query goal g, the query on a database is true under the certainty semantics.
(4) By combining the proofs of Part (2) and (3). $\Box$

Observe that, this time the proof of Part (1) of Theorem 1 strongly depends on the restriction that $\mathbf{Q}^{\neg s,c}$ only contains deterministic queries.

Proposition 2. *Let Q be a query in $\mathbf{Q}^{\neg s,c}$ and D be a database. Then*

1. *$Q_!(D)$ can be solved in time $p(||D||)$;*
2. *$Q_\exists(D)$, $Q_\forall(D)$ or $Q_{\exists\forall}(D)$ can be solved in time $\mathcal{O}(K \times p(||D||))$*

where p is a polynomial in the size of D and K is the number of stable models of the program $P \cup D$.

Proof. Each of the stable models of a DATALOG program with choice can be computed in polynomial time [25, 10, 11]. Finding any stable model is sufficient to solve the query under non-deterministic semantics. For the other semantics, in the worst case we have to find all stable models. This can be done with a simple exhaustive search scheme for generating all possible selections in the choice constructs. Every selection will produce a different stable model in polynomial time. $\Box$

Proposition 2 clarifies the results of using choice models instead of unrestricted stable models as the basis of computation. The number of choice models of a program can be exponential, and so is the number of stable models of a program. However, computing each choice model is polynomial-time, while computing each stable model is exponential-time. Thus, by using choice models, answering queries under the possibility or certainty semantics remains exponential, but polynomial problems can now be computed in polynomial time, by requesting a non-deterministic semantics.

5 Examples

We next present some classical graph problems which can be easily expressed in $\mathtt{DATALOG}^{\neg s,c}$. The first two examples will present two classical problems which are polynomial and $\mathcal{NP}$-complete, respectively.

Example 1. Spanning tree in a graph. Given an undirect graph whose arcs are stored as pairs of edges $g(X,Y)$. A spanning tree for this graph, starting from the source node a, can be defined by the following program ST:

$$\mathtt{spanning\text{-}tree(X,Y)} \leftarrow \mathtt{connected(X)},\ \mathtt{g(X,Y)},\ \mathtt{Y} \neq \mathtt{a},\ \mathtt{X} \neq \mathtt{Y},\ \mathtt{choice(Y,X)}.$$

$$\mathtt{connected(a)}$$
$$\mathtt{connected(Y)} \leftarrow \mathtt{connected(X)},\ \ \mathtt{spanning\text{-}tree(X,Y)}.$$

$$\mathtt{non\text{-}spanning\text{-}tree} \leftarrow \mathtt{g(X,Y)},\ \ \neg\mathtt{spanning\text{-}tree(\_,Y)}.$$

The choice goal here ensures that no node in the connected subgraph constructed by the transitive closure has two incoming arcs, thus delivering a tree. The graph has a spanning tree connecting all nodes to the source node a if the query

$$(!(\neg \textit{not-spanning-tree}), ST)$$

has answer *true*. The query is obviously deterministic. To understand the relevance of non-deterministic semantics, observe that the same query also returns the correct answer under the possibility semantics. Computation under possibility semantics is as efficient as that under non-deterministic semantics for the cases where the graph has a spanning tree. But when this is not the case, then the possibility semantics insists in trying all other selections for choice (and there can be an exponential number of these), while the non-deterministic semantics stops after the first faliure. □

Example 2. Hamiltonian path in a graph. We are given a undirected graph whose nodes and arcs are stored as facts of the form $n(x)$ and $g(x,y)$, respectively. The graph does not contain loops (edges of the form $g(x,x)$). Consider the following $\mathtt{DATALOG}^{\neg s,c}$ program HP which selects a simple paths in the graph:

$$\begin{aligned}\mathtt{simple\text{-}path(X,Y)} \leftarrow\ & \mathtt{connected(X)},\ \ \mathtt{g(X,Y)},\ \ \mathtt{first(Z)},\ \ \mathtt{Y} \neq \mathtt{Z},\ \ \mathtt{Y} \neq \mathtt{X},\\ & \mathtt{choice((X),(Y))},\ \ \mathtt{choice((Y),(X))}\end{aligned}$$

$$\begin{aligned}&\mathtt{first(X)} \leftarrow \quad \mathtt{n(X)},\ \ \mathtt{choice((),(X))}.\\ &\mathtt{connected(Y)} \leftarrow \mathtt{first(Y)}.\\ &\mathtt{connected(Y)} \leftarrow \mathtt{connected(X)},\ \ \mathtt{simple\text{-}path(X,Y)}.\end{aligned}$$

$$\mathtt{non\text{-}HP\text{-}path} \leftarrow \mathtt{n(X)},\ \ \neg\mathtt{simple\text{-}path(\_,X)}.$$

The graph has an Hamiltonian path if there is a simple path which visits all nodes exactly once, that is if the query $(\exists(\neg \textit{not-HP-path}), HP)$ has answer *true*. □

The next example shows how the program of Example 2 can be replicated to express problems in the classes co$\mathcal{NP}$ and $\mathcal{D}^p$.

Example 3. Unique Hamiltonian path. Consider the following $\texttt{DATALOG}^{\neg s,c}$ program UHP which selects three simple paths in the graph that are labeled 1, 2 and 3 respectively. The predicate *diff*-2-3 verifies that the path with label 3 is different from the path with label 2.

```
label(1).
label(2).
label(3).
simple-path(L,X,Y) ← connected(L,X), g(X,Y), first(L,Z), Y ≠ X, Y ≠ Z,
                     choice((X),(Y)), choice((Y),(X))
first(L,Y) ←       label(L), n(Y), choice((L),(Y))
connected(L,Y) ← first(L,Y).
connected(L,Y) ← connected(L,X), simple-path(L,X,Y).
non-HP-sp(L) ←   n(X), ¬simple-path(L,Y).
diff-2-3 ← simple − path(2,X,Y1), simple − path(3,X,Y2), Y1 ≠ Y2.
two-HP ←   ¬non-HP-sp(2), ¬non-HP-sp(3), diff-2-3.
```

The query $(\exists(\neg non\text{-}HP\text{-}sp(1)), UHP)$ solves the $\mathcal{NP}$-complete problem of whether the graph has a Hamiltonian path. The query $(\forall(\neg non\text{-}HP\text{-}sp(1)), UHP)$ solves the co$\mathcal{NP}$-complete problem of whether all maximal simple paths are Hamiltonian. Finally the query $(\exists(\neg non\text{-}HP\text{-}sp(1)) \wedge \forall(\neg two\text{-}HP)), UHP)$ solve the $\mathcal{D}^p$ problem of whether the graph has a unique Hamiltonian path — note that this problem is not $\mathcal{D}^p$-complete as it belongs to the class $\mathcal{US}$, i.e., problems with a unique solution [5]. □

For the program of Example 3, deterministic queries expressible under a non-deterministic semantics are of little practical interest. However, we observe that for this example and others discussed in [24], it is useful to combine fully unbound queries under non-deterministic semantics with bound queries under possibility semantics—under the reasonable assumption is that the stable model chosen for the non-deterministic part is the same used for the existential part of the query. For instance, the query $sp(1, X, Y) \wedge \exists(\neg$ *non-HP-sp(1))* outputs a Hamiltonian path if one exists or returns false otherwise.

6 Implementation

In this section we shall provide an implementation scheme for the computation of choice stratified queries under the various types of semantics.

We first consider the case of non-deterministic semantics. Let a query $Q = (!G, P)$ and a database D be given. In order to solve the query, we have to find a stable model of $P \cup D$. To this end we extend the choice fixpoint proposed for positive choice programs [25, 10, 11] to cope with stratified negation. Therefore we find a stratification for the program P and divide it into a number of subprograms $P_1, \ldots, P_n$, each corresponding to a stratum. Then for each subprogram P_i we fire the function *choice_fixpoint*, passing the stable model of P_{i-1} as input set of facts; for the subprogram P_1 the input set of facts is the database D. Thus we use the following algorithm:

Algorithm 1 *[Non-Deterministic Query Evaluator]*
Input: *Q = (!G,P) : Query, D: Facts*
Output: *true/false*
var: *LP: List of Programs; n: int;* M_P*, M: Facts;*
begin
    compute_strata(P,n,LP);
    let $LP = [P_1, \ldots, P_n]$*;*
    $M := D$*;*
    for $i := 1$ **to** n **do begin**
        $M_P := M$*;*
        $choice\_fixpoint(P_i, M_P, M)$
    end*;*
    return $G \in M$
end.

Observe that the database as well as interpretations are represented as sets of facts (type $Facts$). The function *compute_strata* implements one of well-known methods for finding a stratification; a detailed description of the function *choice_fixpoint* can be found in [11]. Obviously the overall algorithm runs in time polynomial in the size of D.

We next give a unique implementation scheme for the other semantics. For the the sake of presentation, under the compound semantics we shall only consider a subclass of queries that is amenable for a simple implementation. As shown next, the restriction to this subclass does not reduce the generality of our approach.

Let $Q = (\exists(G_1) \wedge \forall(G_2), P)$ be a $\mathtt{DATALOG}^{\neg s,c}$ query. Then Q is *choice disconnected* if $H_1 \cap H_2 = \emptyset$, where H_1 (resp. H_2) is the set of choice rules defining a predicate on which some goal in G_1 (resp., G_2) depends.

Let $\mathbf{Q}^{\neg s,cd}$ denote the set of all choice disconnected queries. The next result shows that the class $\mathbf{Q}^{\neg s,cd}$ has the same expressive power as the class $\mathbf{Q}^{\neg s,c}$.

Proposition 1. $EXPR_{\exists\forall}(\mathbf{Q}^{\neg s,cd}) = DB - \mathcal{D}^p$.

Proof. For the combination of the proofs of Part (2) and (3) of Theorem 1, it is enough to consider two separated problems. $\Box$

Every query in $\mathbf{Q}^{\neg s,c}$ can be made choice disconnected by simply replicating some of the rules. For instance, the query in the Example 4, finding the unique Hamiltonian path, is not choice disconnected but can be put in this format by simply replicating the labeled rules for each value of the label and moving the label from the argument of each predicate into the name of the predicate symbol—e.g., $sp(L)$ becomes sp_1, sp_2 and sp_3 in the corresponding replicated rules.

We are now ready to describe an implementation algorithm for a choice disconnected query. By simply considering the existential goal or the universal goal as true we get a query under the possibility semantics and the certainty semantics respectively. Therefore the algorithm works for these two semantics as well.

Let $Q = (\exists(G_1) \wedge \forall(G_2), P)$ be a query in $\mathbf{Q}^{\neg s,cd}$ and let g_1, g_2 be the predicate symbols of G_1 and G_2, respectively. We divide the program P into three subprograms:

- P_1 (resp., P_2) consists of all rules r having p as head predicate symbol such that g_1 (resp., g_2) depends on p and p depends on some choice rule;
- $P_0 = \{r | r \in P - (P_1 \cup P_2)$ and g_1 or g_2 depends on $r\}$.

By definition of choice disconnected query, $P_1 \cap P_2 = \emptyset$; also, by construction, $P_1 \cap P_0 = P_2 \cap P_0 = \emptyset$. Moreover, P_0 is a stratified program without choice, P_1 and P_2 are $\texttt{DATALOG}^{\neg s,c}$ programs. Consider a stratification for each of the three subprograms. For the sake of presentation, we assume that possible choice rules are all in the same stratum. Then, by construction, the choice rules are in the first stratum both in P_1 and in P_2.

We can now write the algorithm for solving the query Q:

Algorithm 2 *[Choice-Disconnected Query Evaluator]*
Input: $((\exists(G_1) \wedge \forall(G_2)), P) : query;\ D : Facts;$
Output: *true/false;*
var: P_0, P_1, P_2 : *Program;* M_0, M_2, M : *Facts;*
begin
    let $P = P_0 \cup P_1 \cup P_2$;
    $stratified\_fixpoint(P_0, D, M_0)$;
    if $solve(G_2, false, P_2, M_0, M_2)$ **then**
        return $solve(G_1, true, P_1, M_2, M)$
    else
        return $false$
end.

Let D be the initial database, the algorithm proceeds as follows:

1. first, using the classical stratified fixpoint, it computes a set of facts that are relevant for resolving the two goals and are common to every stable model as they do not depend on choice rules; let M_0 be this set of facts;
2. next, using the function *solve*, it computes the query (G_2, P_2) over the database M_0 under certainty semantics; say M_2 the set of facts computed;
3. finally, using the function *solve*, it computes the query (G_1, P_1) over the database M_2 under possibility semantics; say M the set of facts computed.

The function *solve* has five arguments. The first argument denotes the query goal, the second one denotes the semantics of the query, i.e., it is true if we are looking for any model and false otherwise, the third one denotes the input program, the last two arguments denote the input and output set of facts, respectively. The procedure *solve* is defined as follows:

```
function solve(G: Goal; isAny: Bool; P: Program; M0, M: Facts):Bool;
var Previous_choices: list of Choices; Last_choices: Choices; choice:Bool;
     LP: List of Programs;  n: int;
begin
    if P = ∅ then begin
        M := M0;
        return (G = true)
    end
    compute_strata(P, n, LP);
    let LP = [P1, ..., Pn];
    Previous_choices := [];
    repeat
        choice := choice_fixpoint(P1, M0, M, Previous_choices, Last_choices)
        if choice then begin
            append(Previous_choices,Last_choices);
            for i := 2 to n do
                M := T^∞_{Pi ∪ M}(∅);
            end
    until (not choice) or (isAny and G ∈ (M ∪ {true})) or (not isAny and G ∉ M);
    return (G ∈ (M ∪ {true})
end;
```

Note that the type *Facts* is considered as a sub-type of *Program* so that $P_i \cup M$ represents a program; indeed M can be thought of as defining the EDB predicates of P_i. The type *Choices* describes a complete set of choices and also includes suitable information in order to allow the function *choice_fixpoint* any time is fired to select a set of choices different from the previous ones stored in *Previous_choice*.

The function *choice_fixpoint* used here is a simple extension of the original function used for the non-deterministic semantics. The new function returns the set of choices that have been performed and, when called again, it receives all previously performed sets of choices in order to select a new set of choices at any time. The function returns the value *false* when all possible choices have been explored. Therefore, the procedure must be only extended to handle the list of previous sets of choices in order to select the next set of choices or to eventually report that no more choices are available. Of course this part can be easily implemented in time polynomial in the size of M; hence, since also the fixpoint is computed in time polynomial in the size of M and the size of M is polynomially bounded by the size of the input database D, the function *choice_fixpoint* runs in polynomial time.

Not surprisingly, the overall algorithm runs in exponential time because of the iterations of "repeat" in the function *solve*. This cycle is executed a number of times equal to the number of different sets of choices carried out by the procedure *choice_fixpoint* which is, clearly, exponential and is equal to the number of stable models. It is interesting to observe the behavior of this function in the two cases it is used. When it works on P_1, the iteration is kept on until a set of choices is found which satisfies the goal — this corresponds to solve a query under the possibility semantics. On the other hand, when it works on P_2, the iteration is stopped as soon as a set of choices is found which does not satisfy the goal — this corresponds to solving a query under the certainty semantics.

7 Conclusion

The table in Figure 1 summarizes the results that we have obtained about the expressive power of various classes of DATALOG queries. The fact that the expressive power of stable model semantics can be achieved using simple declarative constructs such as stratified negation and choice is of obvious conceptual interest. The practical significance of these results follows from the fact that programs with these constructs always have a well-defined semantics (i.e., never lack a total stable model), have simplicity and intuitive appeal and have an efficient implementation algorithm. In fact, the class of queries for which we have presented an effective implementation scheme has an expressive power (namely, $\mathcal{D}^p$) higher than the class of classical queries under the possibility or certainty semantics.

Query	Complexity	Reference
$!(\mathbf{Q}^{\neg})$	DB-$\mathcal{NP}$	this paper
$\exists(\mathbf{Q}^{\neg})$	DB-$\mathcal{NP}$	Marek-Truszczynski [18]
$\forall(\mathbf{Q}^{\neg})$	DB-co$\mathcal{NP}$	Marek-Truszczynski [18]
$!(\mathbf{Q}^{\neg s,c})$	DB-$\mathcal{P}$	this paper
$\exists(\mathbf{Q}^{\neg s,c})$	DB-$\mathcal{NP}$	this paper
$\forall(\mathbf{Q}^{\neg s,c})$	DB-co$\mathcal{NP}$	this paper
$\exists\forall(\mathbf{Q}^{\neg s,c})$	DB-$\mathcal{D}^p$	this paper
$\exists\forall(\mathbf{Q}^{\neg s,cd})$	DB-$\mathcal{D}^p$	this paper

Fig. 1. *Expressive Power* $\mathtt{DATALOG}^{\neg}$ *and* $\mathtt{DATALOG}^{\neg s,c}$

References

1. S. Abiteboul, E. Simon, V. Vianu. Non-Deterministic Language to Express Deterministic Transformation. In *Proc. of the Ninth ACM PODS Conference*, 1990.
2. S. Abiteboul and V. Vianu. Datalog Extensions for Databases Queries and Updates. In *Journal of Computer and System Science*, 43, pages 62–124, 1991.
3. F. Afrati, S. S. Cosmadakis, M. Yannakakis. On Datalog vs. Polynomial Time. In *Proc. of the Tenth ACM PODS Conference*, pages 13–25, 1991.
4. C. Apt, H. Blair, A Walker. Towards a Theory of Declarative Knowledge. In *Proc. Work. on Found. of Deductive Database and Logic Prog.*, (IMinker Ed.), 1988.
5. A. Blass and Y. Gurevich. On the unique satisfiability problem. In *Information and Control 55*, pages 80–88, 1982.
6. A. Chandra and D. Harel. Structures and Complexity of Relational Queries. In *Journal of Computer and System Science*, 25, pages 99–128, 1982.
7. A. Chandra and D. Harel. Horn clause and generalizations. In *Journal of Logic Programming*, vol. 2, No. 1, pages 1–15, 1985.

8. R. Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets, in *Complexity of Computation (R. Karp, Ed.)*, SIAM-AMS Proc., 1974.
9. M. Gelfond and V. Lifschitz. The stable model semantics of logic programming. In *Proc. of the Fifth Intern. Conf. on Logic Programming*, pages 1070–1080, 1988.
10. F. Giannotti, D. Pedreschi, D. Saccà, and C. Zaniolo. Nondeterminism in deductive databases. In *Proc. 2nd DOOD Conference*, pages 129–146, 1991.
11. S. Greco, C. Zaniolo, and S. Ganguly. *Greedy by Choice.* In *Proc. of the Eleventh ACM PODS Conference*, pages 105–163, 1992.
12. N. Immerman. Languages that Capture Complexity Classes. In *SIAM Journal of Computing*, 16(4), pages 760–778, 1987.
13. D. S. Johnson. A Catalog of Complexity Classes, In *Handbook of Theoretical Computer Science*, Vol. 1, (Ed. J. Leewen) North-Holland, pages 67–161, 1990.
14. P. C. Kanellakis Elements of Relational Databases Theory. In *Handbook of Theoretical Computer Science*, (Ed. J. Leewen) North-Holland, pages 1075–1155, 1990.
15. P. Kolaitis. The Expressive Power of Stratified Logic Programs. *Information an Computation*, 90, pages 50–66, 1990.
16. P. Kolaitis and C. Papadimitriou. Why not negation by fixpoint. In *Journal of Computer and System Science*, 43, pages 125–144, 1991.
17. J. W. Lloyd. *Foundations of Logic Programming*, Springer Verlag, Berlin, 1987.
18. W. Marek, M. Truszczynski. Autoepistemic Logic. *Journal of ACM*, 38(3):588–619, 1991.
19. S. Naqvi and S. Tsur. *A logic language for data and knowledge bases.* Computer Science Press, 1989.
20. C. Papadimitriou. A Note on the Expressive Power of Prolog. In *Bull of the EATCS*, 26, pages 21–23, 1985.
21. C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.
22. C. Papadimitriou and M. Yannakakis, The Complexity of Facets (and Some Facets of Complexity), *Journal of Computer and System Sciences 28*, 2, 1982, pp. 244-259.
23. R. Ramakrisnhan, D. Srivastava, and S. Sudanshan. CORAL — Control, Relations and Logic. In *Proc. of 18th VLDB Conference*, 1992.
24. D. Saccà. The Expressive Power of Stable Models for DATALOG Queries with Negation. unpublished manuscript, 1994 (extended abstract in *Proc. of 2nd Work. on Structural Complexity and Recursion-theoretic methods in Logic Progr.*, 1993).
25. D. Saccà and C. Zaniolo. Stable models and non-determinism in logic programs with negation. In *Proc. of the Ninth ACM PODS Conference*, pages 205–217, 1990.
26. J. S. Schlipf. The expressive power of the logic programming semantics. In *Proc. of the Ninth ACM PODS Conference*, pages 196–204, 1990.
27. J. S. Schlipf. A Survey of Complexity and Undecidability Results in Logic Programming. *Work. Structural Compl. and Recursion-Theoretic Met. in L. P.*, 1993.
28. J. D. Ullman. *Principles of Databases and Knowledge Base Systems*, Vol. I and II. Computer Science Press, 1988.
29. A. Van Gelder. Negation as failure using tight derivations for general logic programs. *Journal of Logic Programming*, vol. 6, No. 1, pages 109–133, 1989.
30. A. Van Gelder, K.A. Ross, and J.S. Schlipf. The well-founded semantics for general logic programs. *Journal of ACM*, 38(3):620–650, 1991.
31. M. Vardi. The Complexity of Relational Query Languages. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pages 137–146, 1982.

This article was processed using the LaTeX macro package with LLNCS style