Acta Informatica 37, 699–725 (2001)

© Springer-Verlag 2001

Extending stratified datalog to capture complexity classes ranging from $\mathcal{P}$ to $\mathcal{QH}^{\star}$

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

[1] DEIS, University of Calabria & ISI-CNR, 87030 Rende, Italy
e-mail: ({greco,sacca}@si.deis.unical.it)
[2] Computer Science Dept., University of California, Los Angeles, CA 90024, USA
e-mail: (zaniolo@cs.ucla.edu)

Received: 30 November 1995 / 1 April 2001

Abstract. This paper presents a unified solution to the problem of extending stratified DATALOG to express database complexity classes ranging from $\mathcal{P}$ to $\mathcal{QH}$; $\mathcal{QH}$ is the query hierarchy containing the decision problems that can be solved in polynomial time by a deterministic Turing machine using a constant number of calls to an $\mathcal{NP}$-oracle. 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 the nondeterminism of stable models in a disciplined fashion, (iii) the ability to bind a query to the lowest complexity level that includes the problem at hand, and (iv) a general algorithm that adapts its behavior to the desired level of complexity required by the query so that exponential time computation is only required for hard problems.

1 Introduction

Designing declarative, logic-oriented database languages for wider application domains has been a key motivation of much of the research on databases and knowledge bases in the past years. The introduction of DATALOG represented a major breakthrough in this line of work, due to DATALOG's ability to express 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 extended abstract of this paper was presented at the Fifth International Conference on Database Theory (ICDT95) [14]. Work partially supported by the Murst 40% projects Data-X and D21.

an operational semantics that is amenable to very efficient implementation as demonstrated by a number of prototypes of deductive database systems [25, 30, 22].

Unfortunately, the basic DATALOG language (without negation and function symbols) is severely limited in its expressive power and cannot express many of the queries of practical interest. While the exact expressive power of DATALOG has not been characterized completely, it has been shown that DATALOG only captures a proper subset of monotonic polynomial-time queries [4].

In order to support nonmonotonic queries, negation is allowed in the bodies of the rules (we will write DATALOG$^\neg$ to denote DATALOG with negation). Of particular interest is *stratified negation*, which avoids the semantic and implementation problems connected with the unrestricted use of nonmonotonic constructs in recursive definitions [5, 7, 38]. Simple, intuitive semantics leading to efficient implementation exists for stratified DATALOG$^\neg$; unfortunately, as shown in [20], this language has a reduced expressive power as it can only express a proper subset of fixpoint queries.

The simplest step toward greater expressive power is to remove the condition that negation must be stratified. To see an example of non-stratified negation, consider the following DATALOG$^\neg$ program P_{Tree}:

$$
\begin{array}{ll}
(r_1)\ reached(a) & \\
(r_2)\ reached(Y) \leftarrow & spanTree(X,Y) \\
(r_3)\ spanTree(X,Y) \leftarrow & reached(X), arc(X,Y), Y \neq a, \\
 & \neg\, diffChoice(X,Y) \\
(r_4)\ diffChoice(X,Y) \leftarrow & spanTree(Z,Y),\ Z \neq X \\
(r_5)\ nonTree \leftarrow & node(X),\ \neg\, reached(X) \\
(r_6)\ nonTree \leftarrow & arc(X,Y),\ \neg\, spanTree(X,Y), \\
 & \neg\, spanTree(Y,X)
\end{array}
$$

where arc and $node$, defined by a number of suitable facts, encode an undirected graph, say G. The negation in the body of rule r_3 is non-stratified because the negated predicate $diffChoice$ in the body is mutually recursive with the head predicate $spanTree$; on the other hand, the negations in the body of the rules r_5 and r_6 are stratified.

Unfortunately, the relaxation of the requirement that negation must be stratified opens a Pandora's box of semantic and computational problems. Take for instance the concept of *well-founded model* [39]. All programs have a unique well-founded model which can be computed in polynomial time; however, the well-founded models of many programs are not total, and thus the meaning of portions of these programs remains undefined. For example, the well-founded model of the program P_{Tree} above is total if and only if G_a is a tree, where G_a denotes the component of G containing the node

a: then the atom $nonTree$ is true if and only if the graph is not connected. On the other hand, if G_a is cyclic, the atom $nonTree$ remains undefined; then, the well-founded model does not provide a semantics of practical use for this program. Another problem with well-founded semantics is that it expresses exactly fixpoint queries [40], which are only a proper subset of polynomial-time queries.

A dramatic leap in expressive power is provided by the concept of *stable model* [10], which has emerged as the compendium of many concepts and theories developed over the years by AI researchers working on nonmonotonic reasoning, default theories and autoepistemic logic. This gain in expressive power, however, is not without complications. One is the nondeterministic nature of such semantics that follows from the fact that a program can have several stable models. Take, for example, the program P_{Tree} above. In general, P_{Tree} admits many total stable models, each one representing a spanning tree of G_a (the number of spanning trees may be exponential and so is the number of stable models). The atom $nonTree$ is true in a stable model M if and only if (i) G has at least one node which is not in the spanning tree retrieved by M (thus G is not connected) or (ii) G_a is cyclic.

Determinism for stable model semantics can be enforced by querying a ground literal (*query goal*) and returning the answer "true" if the literal holds true either for *some* stable model (*possible semantics*), or for *all* stable models (*certain semantics)*; these are known in AI as membership and entailment semantics, respectively. For instance, given the program P_{Tree} and the query goal $nonTree$, the two semantics coincide and the answer is "true" if and only if the graph is not a tree. These two semantics greatly extend the expressive power of DATALOG$^\neg$ queries; indeed, queries under possible semantics and certain semantics of stable models, respectively, express all decision problems in the classes $\mathcal{NP}$ and co$\mathcal{NP}$[24, 35]. However, there remain the following significant problems:

1. The usage of unrestricted negation in programs is often neither simple nor intuitive, and, for example, might lead to writing programs that have no total stable models (the same problem as for well-founded models). For instance, in the program P_{Tree}, in an attempt to simplify it, one could decide to modify the third rule into

   $$(r_3')\, spanTree(X, Y) \leftarrow reached(X), arc(X, Y), Y \neq a, \neg reached(Y)$$

   and remove the fourth rule. Then the resulting program will have no total stable models, thus losing its practical meaning.
2. Deterministic semantics may require the computation of an exponential number of stable models, not only for programs expressing $\mathcal{NP}$-hard problems but also for those solving polynomial-time problems. For instance, given the query goal $nonTree$ on P_{Tree}, if the graph is not a tree

then certain semantics must check that the goal is indeed true in every stable model; on the other hand, when the query goal is $\neg nonTree$, it will be the turn of possible semantics to search for all stable models before realizing that $\neg nonTree$ is never true.

3. Even computing a single stable model may require exponential time. For instance, if in our running program example we modify rule r_5 into:

$$(r_5')\ nonTree \leftarrow node(X),\ \neg\, reached(X), \neg\, nonTree$$

then the program will admit no total stable model when the graph G is not a tree but it will be necessary to spend an exponential amount of time before giving up the computation of a model.

With a great deal of 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. those based on the inflationary fixpoint computation procedure [3,2,21]. This paper is motivated by the opposite conviction that, because of the important advantages offered by a model-theoretic semantics, we should strive to preserve the declarative style of stable model semantics while trying to remove its limitations. This approach has already produced significant results that are briefly reviewed next.

A first seminal effect was defining the semantics of a nondeterministic construct, called *choice*, in terms of stable models [33]. To appreciate the usage of this construct, observe that the rules r_3 and r_4 above can be replaced with the following unique choice rule:

$$(r_3'')\ spanTree(X,Y) \leftarrow reached(X), arc(X,Y), Y \neq a, choice((Y),(X)).$$

The semantics of the choice construct can be informally explained in terms of functional dependencies (FDs): the FD $Y \rightarrow X$ must be enforced while deriving $spanTree$ literals and, then, an arc is to be discarded if another arc with the same endpoint has already been derived. An efficient implementation schema was proposed in [11] where it was shown that, for DATALOG with choice programs, the computation of a choice model can be done in polynomial time. The combination of choice with extrema aggregates is investigated in [15] and many other facets of logic programming with choice are detailed in [12]. It is worth noting that a construct related to choice is the $witness$ operator, introduced in [2] in the context of first order queries with inflationary fixpoint. In-depth relationships between choice and witness are investigated in [12].

The fact that the expressive power of total stable model semantics may go beyond $\mathcal{NP}$ and co$\mathcal{NP}$ was first shown in [31], where total stable models under the so-called definite semantics were shown to capture the class

$\mathcal{D}^p$[27]— a literal holds true in the definite semantics if it is true in all stable models and, besides, the program admits at least one stable model.

Stable models have also been studied in the domain of partial interpretations [29, 33]. Many types of partial stable models have been investigated in [34] and they have been classified according to their deterministic or nondeterministic properties. In particular, it was shown there that the presence of multiple partial stable models is not necessarily a manifestation of nondeterminism as it could just represent different degrees of undefinedness. According to [34], models are pairwise nondeterministic if a positive literal of one occurs negated in the other; a model is deterministic if it is pairwise nondeterministic w.r.t. no other stable model. The expressive powers of nondeterministic partial stable models, including *maximal stable models* (i.e. stable models which are maximal w.r.t. set containment) and *least-undefined stable models* (i.e. partial models with minimal set of undefined literals) have been analyzed in [32]. The expressive power of the deterministic ones, including the *maximum deterministic model* (the largest deterministic stable model) were investigated in [16].

Finally, properties of programs for which possible and certain semantics coincide (*"possible is certain"* semantics) have been discussed in [13], where a language based on the "possible is certain" semantics for least-undefined stable models is presented which turns out to be very expressive, whereas no practical language is conjectured to exist for other types of nondeterministic stable model semantics.

The current situation is that the main issues on non-monotonic logic programming with stable model semantics have been clarified as result of the previous research work, but unfortunately, success of this programming paradigm in practical application domains remain elusive. There are many reasons for the missed success[1]. One is that the approach is too complicated: we need to isolate the most relevant features of stable models and recast and package them into a simplified, intuitive framework, as it was done for stratified negation. Therefore, in this paper, we propose a simple framework for exploiting advances in model-theoretic semantics for non-monotonic DATALOG, without surrendering the naturalness and efficiency of stratified negation.

Our proposal is based on a language where the usage of stable model semantics is disciplined to avoid both undefinedness and unnecessary computational complexity, and to refrain from abstruse forms of unstratified negation. The core of the language is stratified DATALOG, that is extended with only one type of non-stratified negation, hardwired into the choice construct. The disciplined structure of negation in our language and its resulting

[1] The whole field of DATALOG programming "lost the boat," according to a metaphor by Jeff Ullman.

amenability to efficient implementation implies that a first important property follows: *every program in our language has at least one total stable model, and each stable model can be computed in polynomial time*.

A query goal in our language is a ground literal preceded by one of the symbols

$$!,\ \exists,\ \forall$$

which are used to select the semantics that matches the intrinsic complexity of the problem at hand. The quantifiers '$\exists$' and '$\forall$' activate, respectively, the possible semantics and the certain one, whereas the symbol '!' calls on the nondeterministic semantics, whereby just one of the total stable models expressed through the choice constructs is computed. For instance, given the program consisting of the rules r_1, r_2, r_3'', r_5 and r_6 above, the query goal $!(\neg\, nonTree)$ activates the nondeterministic semantics, so that the question of whether the graph is a tree can be answered in polynomial time. The answer of this query does not depend on the stable model selected in the computation as the possible and certain semantics coincide. Thus, we can now state a second desirable property of our language: *every decision problem in $\mathcal{P}$ can be expressed by a query that executes in polynomial time.*

The fact that our language captures $\mathcal{P}$ without assuming that the universe is ordered is made possible by the nondeterminism of choice. However, the programmer is now given the responsibility to write a query with "possible is certain" semantics to avoid multiple answers. Indeed, multiple answers are meaningful in search queries but not in queries expressing decision problems.

Let us now consider the program consisting of the rules r_1, r_2 above plus the following rules:

$$\begin{array}{ll} (r_3''')\ spanTree(X, Y) \leftarrow & reached(X), arc(X, Y), \\ & Y \neq a, choice((Y), (X)), choice((X), (Y)). \\ (r_5'')\ nonHpath \leftarrow & node(X), \neg reached(X). \end{array}$$

The spanning tree computed by a total stable model is now enforced by the second choice to be a simple path: indeed the additional FD $X \rightarrow Y$ imposes that any two arcs in $spanTree$ have distinct startpoints besides distinct endpoints. So the query goal $\exists(\neg\, nonHpath)$ solves the problem of whether the graph has a Hamiltonian simple path i.e. a path passing through all nodes exactly once – this problem is known to be $\mathcal{NP}$-complete [27]. On the other hand, $\forall(nonHpath)$ solves the complementary co$\mathcal{NP}$-complete problem of whether no Hamiltonian simple path exists in the graph. Note that the query goal $!nonHpath$ does not make sense: in fact, given a graph G with a Hamiltonian simple path, an answer could instead be "true" if the simple path constructed by the consulted stable model happens to be non-Hamiltonian.

By allowing conjunctions and/or disjunctions of quantified ground literals as query goals, the new language goes beyond $\mathcal{NP}$ and co$\mathcal{NP}$ and enables users to easily express and effectively solve all decision problems in the *query hierarchy* $\mathcal{QH}$[2], which is the class of all decision problems which can be solved in polynomial time by a deterministic Turing machine using any constant number of calls to an $\mathcal{NP}$-oracle [18,27]. We note that, as shown in [13], the query hierarchy is captured by DATALOG$^\neg$ also under the "possible is certain" semantics of least undefined stable models. Here, we have achieved the same expressive power with a much simpler language based on total interpretations.

In summary, our language incrementally extends stratified negation to capture decision problems ranging from $\mathcal{P}$ up to $\mathcal{QH}$, passing through various interesting subclasses of $\mathcal{QH}$ such as $\mathcal{NP}$, co$\mathcal{NP}$ and $\mathcal{D}^p$; the desired level of expressivity power is enabled in a controlled fashion and can be automatically adapted to the complexity of the problem at hand. So another important feature of the language is: *the ability to express many classical $\mathcal{NP}$-hard problems without gross inefficiencies in solving polynomial problems.* Indeed, the paper presents a unifying algorithm that automatically adapts to the complexity of the problem at hand, making stable model semantics amenable to effective implementation.

The paper is organized as follows. We give basic definitions and results on stable models and *bound* (i.e. ground) DATALOG$^\neg$ queries respectively in Sects. 2 and 3. In Sect. 4 we introduce the class of DATALOG$^\neg$ queries with stratified negation and $choice$ and we show that, under different total stable model semantics, such queries capture complexity classes ranging from $\mathcal{P}$ to $\mathcal{QH}$. In Sect. 5 we describe algorithms for the computation of queries at different levels of complexity. Finally, in Sect. 7, we present our conclusions.

2 Preliminary definitions

We assume the reader is familiar with the concepts of relational databases and of the DATALOG language [19,37] as well as of logic programming [23]. Non-standard or specific terminology and notation are presented below.

A *logic program* (or, simply, a *program*) P is a finite set of rules. Each *rule* of P has the form $A \leftarrow A_1, ..., A_m$, where A is an atom (the *head* of the rule) and $A_1, ..., A_m$ are literals (the *body* of the rule). A rule with an empty body is called a *fact*.

Given a logic program P, the Herbrand universe for P, denoted H_P, is the set of all possible ground terms recursively constructed by taking

[2] The class is better known as the Boolean hierarchy $\mathcal{BH}$, but we prefer to say that our query language captures the query hierarchy!

constants and function symbols occurring in P. The Herbrand Base of P, denoted B_P, is the set of all possible ground atoms whose predicate symbols occur in P and whose arguments are elements from the Herbrand universe. A *ground instance* of a rule r in P is a rule obtained from r by replacing every variable X in r by a ground term in H_P. The set of ground instances of r is denoted by $ground(r)$; accordingly, $ground(P)$ denotes $\bigcup_{r \in P} ground(r)$. An interpretation I of P is a subset of B_P. A ground positive literal A (resp. negative literal $\neg A$) is true w.r.t. an interpretation I if $A \in I$ (resp. $A \notin I$). A conjunction of literals is true in an interpretation I if all literals are true in I. A ground rule is true in I if either the body conjunction is false or the head is true in I. A *(Herbrand) model* M of P is an interpretation that makes each ground instance of each rule in P $true$. A model M for P is minimal if there is no model N for P such that $N \subset M$.

Let I be an interpretation for a program P. The *immediate consequence operator* $T_P(I)$ is defined as the set containing the heads of each rule $r \in ground(P)$ s.t. the body of r is true in I. The semantics of a *positive* (i.e. negation-free) logic program P is given by the unique minimal model; this minimum model coincides with the least fixpoint $T_P^{\infty}(\emptyset)$ of T_P [23]. Generally, the semantics of logic programs with negation can be given in terms of total stable model semantics [10] which we now briefly recall.

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

A logic program may have no, one or several stable models and deciding whether it admits at least one stable model is $\mathcal{NP}$-complete [24]. Positive programs have a unique stable model which coincides with the minimum model [10].

Given a program P and two predicate symbols p and q, we write $p \rightarrow q$ if there exists a rule where q occurs in the head and p in the body or there exists a predicate s such that $p \rightarrow s$ and $s \rightarrow q$. A program is *stratified* if there exists no rule where a predicate p occurs in a negative literal in the body, q occurs in the head and $q \rightarrow p$ i.e. there is no recursion through negation [5]. Stratified programs have a unique stable model which coincides with the *stratified model*, obtained by partitioning the program into an ordered number of suitable subprograms (called 'strata') and computing the fixpoints of every stratum from the lowest one up [5, 28, 39].

A DATALOG$^{\neg}$ program P is a function-free logic program whose predicates are partitioned into *extensional* and *intensional* predicates (EDB and IDB predicates, respectively). EDB predicates never occur in the rule heads as they are assumed to be defined by a number of facts stored in some

database D. Thus the set of EDB predicates is seen as a database scheme, denoted by DS_P. We assume that all relation attributes have as domain the same countable domain, say U. Let the set of all finite databases on DS_P be denoted $\mathbf{D}_P$; any database D in $\mathbf{D}_P$ will be seen as a set of facts $\{r(t)|t$ is a tuple in some relation r of $D\}$. Therefore, $P \cup D$ is a DATALOG$^\neg$ program and is denoted by $P(D)$.

If a DATALOG$^\neg$ program is positive (resp., stratified) then it is simply called a DATALOG (resp., DATALOG$^{\neg s}$) program. A DATALOG$^{\neg s}$ program P is called $semipositive$ if negation is only applied to EDB atoms [1]. In this case, for each D in $\mathbf{D}_P$, $P(D)$ can be partitioned into just two strata: D and P.

3 Semantics and complexity of DATALOG$^\neg$ queries

We assume that the reader is familiar with the basic notions of complexity classes [18,27] and of query language complexity evaluation (see for example [2,3,6,8,17,19–21,36,41]).

Definition 1. *A (*bound*)* DATALOG$^\neg$ query Q *is a pair* $(v\, G, P)$ *where* G *is a ground literal, called* query goal, P *is a* DATALOG$^\neg$*program and* v *is a symbol denoting the type of semantics adopted for the query: '!' for nondeterministic semantics, '*$\exists$*' for possible semantics, '*$\forall$*' for certain semantics and* $\exists\forall$ *for definite semantics [31]. The* answer *to a query* $Q = (v\, G, P)$ *on a database* D *in* $\mathbf{D}_P$*, denoted by* $Q(D)$*, is defined as follows:*

1. *under the* nondeterministic semantics *(*$v = !$*):*
   - *(a)* $true$ *if there is a stable model* M *of* $P \cup D$ *for which* G *is true in* M*, and*
   - *(b)* false *if either there exists no stable model or there is a stable model* M *of* $P(D)$ *for which* G *is false in* M*;*
2. *under the* possible semantics *(*$v = \exists$*):*
   - *(a)* $true$ *if there is a stable model* M *of* $P(D)$ *for which* G *is true in* M*, and*
   - *(b)* false *otherwise;*
3. *under the* certain semantics *(*$v = \forall$*):*
   - *(a)* $true$ *if for each stable model* M *of* $P(D)$*,* G *is true in* M*, and*
   - *(b)* false *otherwise;*
4. *under the* definite semantics *(*$v = \exists\forall$*):*
   - *(a)* $true$ *if* $P(D)$ *has at least one stable model and for each stable model* M *of* $P(D)$*,* G *is true in* M*, and*
   - *(b)* false *otherwise.*

$!\mathbf{Q}^\neg$ *(resp.,* $\exists\mathbf{Q}^\neg$*,* $\forall\mathbf{Q}^\neg$*,* $\exists\forall\mathbf{Q}^\neg$*) denotes the set of all possible* DATALOG$^\neg$ *queries under nondeterministic (resp., possible, certain, definite) semantics.* □

It turns out that any query $Q = (v\,G, P)$ in $\exists\mathbf{Q}^{\neg}$, $\forall\mathbf{Q}^{\neg}$ or $\exists\forall\mathbf{Q}^{\neg}$ represents a function from $\mathbf{D}_P$ to $\{true, false\}$. On the other hand, $Q = (!\,G, P)$ in $!\mathbf{Q}^{\neg}$ represents a relation on $\mathbf{D}_P \times \{true, false\}$ as the answer on a given database can be both $true$ and $false$; we say that Q is *deterministic* if it has exactly one answer for each database $D \in \mathbf{D}_P$. Note that, as queries correspond to decision problems and any decision problem with multiple answers is meaningless, the only queries of practical interest are the deterministic ones – nondeterminism is only used to increase the expressive power and/or to simplify the formulation. For instance, as shown in the example of Sect. 1, in order to test whether an undirected graph is a tree, we can write a program which nondeterministically selects a spanning tree of the graph component containing a given node; then the graph is a tree if and only if the spanning tree coincides with the graph. This program has as many stable models as the number of spanning trees for the above graph component but the query result remains the same no matter which stable model is selected.

The deterministic nature of a query in $!\mathbf{Q}^{\neg}$ can be characterized in terms of possible and certain semantics as follows.

Proposition 1. *A query $Q = (!G, P)$ in $!\mathbf{Q}^{\neg}$ is deterministic if and only if for each $D \in \mathbf{D}_P$, either $P(D)$ has no total stable model or $(\exists G, P)(D) = (\forall G, P)(D)$.*

Proof. If $P(D)$ has no total stable model then the answer under nondeterministic semantics can only be false by definition. On the other hand, if a stable model exists, two possibilities may arise:

1. the answer is true under certain semantics; so, as G is true in every stable model and there exists at least one stable model, the answer is true under both possible and nondeterministic semantics;
2. the answer is false under certain semantics; so, as G is false in at least one stable model, the answer will be unique under the nondeterministic semantics if and only if G is false in every stable model i.e. if and only if the answer is false also under possible semantics.

This concludes the proof. □

From now on, we shall only consider deterministic queries and, with a little abuse of notation, $!\mathbf{Q}^{\neg}$ will denote the subclass of all queries which are deterministic – note that this subclass is not recursively enumerable as the problem of whether a query is deterministic is not decidable [2]. On the other hand, $\exists\mathbf{Q}^{\neg}$ and $\forall\mathbf{Q}^{\neg}$ are recursive as they continue to denote the sets of all possible queries under the two associated semantics.

Given a query $Q = (v\,G, P)$, where $v\{\,!, \exists, \forall, \exists\forall\}$, the *database collection* of Q is the set of all databases $D \in \mathbf{D}_P$ for which $Q(D)$ has a true answer and is denoted by $\mathcal{EXP}(Q)$. It is well known that for each query Q, $\mathcal{EXP}(Q)$ is indeed a *generic* database set [6, 1], i.e. it is closed under renaming of constants in $U - C$, where C is the set of constants occurring in P and in G – thus the constants not in C are not interpreted and relationships among them are only those explicitly provided by the databases. From now on any generic set of databases on the same scheme will be called a *database collection*.

According to the *data complexity* approach of [6, 41] for which the program is assumed to be constant while the database is variable, the complexity of a query under a given semantics coincides with the complexity of the problem of recognizing the associated database collection.

The *expressive power* of a type of semantics (i.e. nondeterministic, possible, certain or definite version) is given by the class of the database collections of all queries, i.e. $\mathcal{EXP}(v\,\mathbf{Q}^{\neg}) = \{\mathcal{EXP}(Q) | Q \in v\,\mathbf{Q}^{\neg}\}$, where $v \in \{!, \exists, \forall\}$. The expressive power of each semantics will be compared with database complexity classes, defined as follows. Given a Turing machine complexity class C (for instance $\mathcal{P}$ or $\mathcal{NP}$), a relational database scheme DS, and a database collection $\mathbf{D}$ on DS, $\mathbf{D}$ is *C-recognizable* if the problem of de-ciding whether D is in $\mathbf{D}$ is in C. The *database complexity class $\mathcal{DB}$-C* is the family of all C-recognizable database collections (for instance, $\mathcal{DB}$-$\mathcal{P}$ is the family of all database collections that are recognizable in polynomial time). If the expressive power of a given semantics coincides with some complexity class $\mathcal{DB}$-C, we say that the given semantics captures (or expresses all queries in) $\mathcal{DB}$-C.

Let $v\,\mathbf{Q}^{\neg s}$, where $v \in \{!, \exists, \forall, \exists\forall\}$, be the set of all queries in $v\,\mathbf{Q}^{\neg}$ whose programs are stratified. The following results are known in the literature. Recall that $\mathcal{D}^p$ is the class of all possible conjunctions of two decision problems, one in $\mathcal{NP}$ and the other in co$\mathcal{NP}$ [18, 27].

Fact 1

1. $\mathcal{EXP}(!\mathbf{Q}^{\neg s}) = \mathcal{EXP}(\exists\mathbf{Q}^{\neg s}) = \mathcal{EXP}(\forall\mathbf{Q}^{\neg s}) \subset \mathcal{P}$ (see [20]);
2. $\mathcal{EXP}(\exists\mathbf{Q}^{\neg}) = \mathcal{DB}$-$\mathcal{NP}$ (see [24]);
3. $\mathcal{EXP}(\forall\mathbf{Q}^{\neg}) = \mathcal{DB}$-co$\mathcal{NP}$ (see [24, 35]);
4. $\mathcal{EXP}(\exists\forall\mathbf{Q}^{\neg}) = \mathcal{DB}$-$\mathcal{D}^p$ (see [31]). □

Possible, certain and definite total stable model semantics capture two complementary high complexity classes. One would then expect that nondeterministic semantics captures some lower complexity class. The next result shows that this is not the case.

Theorem 1. $\mathcal{EXP}(!\mathbf{Q}^{\neg}) = \mathcal{DB}\text{-}\mathcal{NP}$.

Proof. (Membership to $\mathcal{DB}$-$\mathcal{NP}$.) Take any query $Q = (!G, P)$ in $!\mathbf{Q}^{\neg}$ and a database D over DS_P; we want to recognize whether $Q(D)$ is $true$. To this end, we simply guess an interpretation M of $P(D)$ and test in (deterministic) polynomial time whether (i) M is a total stable model and (ii) G is in M. The check that M is a stable model can be done in polynomial time as we only need to compute the least fixpoint of P^M and check whether it coincides with M.
(Completeness.) We use Fagin's result [9] that every $\mathcal{NP}$ recognizable database collection, say $\mathbf{D}$ on the database scheme DS, is defined by an existential second order formula $\exists R \Phi$, where R is a list of predicate symbols distinct from those in DS and Φ is a first-order formula involving predicate symbols in DS and in R. As shown in [21], 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 list of predicate symbols distinct from those in DS, $\theta_1, ..., \theta_k$ are conjunctions of literals involving variables in X and Y, and predicate symbols in S and DS_P. Consider the query $(!\neg g, P)$, where P is the following $\mathtt{DATALOG}^{\neg}$ program with $DS_P = DS$ and $\mathbf{D} \subseteq \mathbf{D}_P$:

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

Consider any database $D \in \mathbf{D}_P$. The first two groups of rules nondeterministically select a relation for each predicate in S – by viewing S as a database scheme, we can say that the rules select a database Z for S. The third group of rules derives $q(x)$ for each x such that there exists y, for which some formula $\theta_i(x,y)$ is true w.r.t. Z and D. The fourth rule checks whether there is some x for which the formula is not satisfied; if this will happen then g will be set to true. The fifth rule forces the global program to have a total stable model only if g is false. Thus, the program $P(D)$ has a total stable model iff $(!\neg g, P)(D)$ is true and, then, iff the above second order Skolem normal form is satisfied by D, i.e. the database D belongs to the database collection $\mathbf{D}$. □

Example 1 Consider the following $\mathtt{DATALOG}^{\neg}$ program P computing a Hamiltonian simple path:

$$
\begin{array}{ll}
(r_1)\ reached(a) & \\
(r_2)\ reached(Y) \leftarrow & pathArc(X,Y) \\
(r_3)\ pathArc(X,Y) \leftarrow & reached(X),\ arc(X,Y), \\
 & Y \neq a,\ \neg\, diffChoice(X,Y) \\
(r_4)\ diffChoice(X,Y) \leftarrow & pathArc(Z,Y),\ Z \neq X \\
(r_5)\ diffChoice(X,Y) \leftarrow & pathArc(X,Z),\ Z \neq Y \\
(r_6)\ nonHpath \leftarrow & node(X),\ \neg\, reached(X) \\
(r_7)\ p \leftarrow & nonHpath,\ \neg\, p.
\end{array}
$$

The query $(!\neg\, nonHpath, P)$ expresses the $\mathcal{NP}$-complete problem of whether an undirected graph has a Hamiltonian simple path. □

The previous result highlights the fact that the complexity of stable model semantics derives not only from the large number of stable models but also from the intrinsic difficulty of finding one stable model. Therefore even nondeterministic semantics, which computes only one stable model, may require exponential time (of course unless $\mathcal{P}$=$\mathcal{NP}$). To remove this problem, in the next section we shall propose a restricted form of negation and a combined usage of stable model semantics to extend stratified negation in an incremental, disciplined fashion: the power can be controlled by the user to select the desired level of complexity which may range from $\mathcal{P}$ up to the entire query hierarchy which includes a large number of meaningful problems.

4 Choice and stratification in DATALOG queries

The *choice* construct (supported in $\mathcal{LDL}$++ [25] and, in some form, in Coral [30]), is used to enforce functional dependencies (FD) constraints on rules of a logic program. A rule r with choice constructs, called a ***choice rule***, has the following general format:

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

where $B(Z)$ denotes the conjunction of all the literals in the body of r that are not choice constructs, and $X_i,\ Y_i,\ Z$ for $1 \leq i \leq k$, denote vectors of variables occurring in the body of r such that $X_i \cap Y_i = \emptyset$ and $X_i, Y_i \subseteq Z$. Each construct $choice((X_i),(Y_i))$ prescribes that the set of all consequences derived from r, say R, must respect the FD $X_i \rightarrow Y_i$. Let $FD_r = \{X_i \rightarrow Y_i | i = 1, ..., k\}$. The enforcing of all dependencies is done as follows. We start from a first consequence, say a, and we set $R = \{a\}$ – obviously R satisfies FD_r. Then, while deriving a subsequent consequence, say b, this is included in R only if $R \cup \{b\}$ satisfies FD_r.

The formal semantics of choice is given in terms of stable models by replacing the above choice rule with the following rules:

1. Replace r with a rule r' (called *modified choice rule*) obtained by substituting the choice atoms with the atom $chosen_r(W)$:

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

   where $W \subseteq Z$ is the list of all variables appearing in the choice goals, i.e. $W = \bigcup_{1 \leq j \leq k} X_j \cup Y_j$.
2. Add the new rule (called *chosen rule*)

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

3. For each $choice((X_i), (Y_i))$ $(1 \leq i \leq k)$, add the new rule *diffChoice rule*

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

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

   For instance, a rule r of the form

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

is rewritten as:

$$\begin{array}{ll}
r_1 : & p(X, Y, W) \leftarrow q(X, Y, Z, W),\ chosen_r(X, Y, Z) \\
r_2 : & chosen_r(X, Y, Z) \leftarrow q(X, Y, Z, W),\ \neg diffChoice_r(X, Y, Z) \\
r_3 : & diffChoice_r(X, Y, Z) \leftarrow chosen_r(X, Y', Z),\ Y \neq Y' \\
r_4 : & diffChoice_r(X, Y, Z) \leftarrow chosen_r(X', Y, Z'),\ Z \neq Z'
\end{array}$$

where the $choice$ predicates have been substituted by the $chosen$ predicate and for each $choice$ predicate there is a *diffChoice* rule. We have $FD_r = \{XZ \rightarrow Y, Y \rightarrow Z\}$, where X, Y and Z denote respectively the first, the second and the third attribute of $chosen_r$. The dependencies in FD_r hold in the $chosen_r$ relation as they are enforced in the derivations of the rule r_2 by means of the *diffChoice* predicates. The dependencies in FD_r also hold in the p relation only if there are no other rules with p in the head which could add additional tuples violating FD_r.

A DATALOG$^\neg$ program P with choice rules is called a *choice program*. The *standard version* $sv(P)$ of P is the program obtained from P by applying the above transformation to every choice rule. Given a database D, any stable model of $sv(P)(D)$ is called a *choice model* of $P(D)$. Moreover, we say that P is *stratified modulo choice* if, by considering choice atoms as extensional atoms, the program results stratified. If P is stratified modulo choice, then the choice models of $P(D)$ are in general multiple but the existence of at least one as well as its computation in polynomial time is guaranteed [34, 11, 15].

Let $\mathsf{DATALOG}^{\neg s,c}$ denote the set of all $\mathsf{DATALOG}^{\neg}$ programs that are stratified modulo choice and $!\mathbf{Q}^{\neg s,c}$, $\exists\mathbf{Q}^{\neg s,c}$, $\forall\mathbf{Q}^{\neg s,c}$ and $\exists\forall\mathbf{Q}^{\neg s,c}$ denote the sets of all queries with P in $\mathsf{DATALOG}^{\neg s,c}$. The definition of answer for a query $(v\,G, P)$ in $v\,\mathbf{Q}^{\neg s,c}$, $v \in \{\,!, \exists, \forall, \exists\forall\}$ is obtained from Definition 1 by simply replacing "stable model" with "choice model". Again, under the nondeterministic semantics, we restrict our attention to deterministic queries; therefore, $!\mathbf{Q}^{\neg s,c}$ is the subset of all deterministic queries and is not recursively enumerable.

Theorem 2.

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

Proof.

1. (Membership). Let $Q = (!G, P)$ be any query in $\mathbf{Q}^{\neg s,c}$. It is known that if P is semipositive then the computation of a stable model for $sv(P)$ can be done in polynomial time [11]. Generally, it is sufficient to determine a stratification for P and to compute one stratum at a time, following the ordering of strata. The computation of a stable model at each stratum proceeds as for a semipositive program; hence, as the number of strata is constant, the overall computation is done in polynomial time.
(Completeness). It is known that semipositive $\mathsf{DATALOG}^{\neg}$ plus an ordering on the domain captures the complexity class $\mathcal{P}$ [26]. Therefore, it is sufficient to introduce an ordering on the domain and 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:

$$
\begin{aligned}
r_1 &: \mathit{domain}(X) \\
r_2 &: \mathit{min}(X) \leftarrow \mathit{domain}(X),\ \mathit{choice}((),(X)) \\
r_3 &: \mathit{reached}(X) \leftarrow \mathit{min}(X) \\
r_4 &: \mathit{reached}(X) \leftarrow \mathit{succ}(Y,X) \\
r_5 &: \mathit{succ}(X,Y) \leftarrow \mathit{reached}(X),\ \mathit{domain}(Y),\ \mathit{min}(Z),\ Y \neq Z, \\
&\qquad\qquad \mathit{choice}((X),(Y)),\ \mathit{choice}((Y),(X)) \\
r_6 &: \mathit{nonLast}(X) \leftarrow \mathit{succ}(X,W) \\
r_7 &: \mathit{max}(X) \leftarrow \mathit{domain}(X),\ \neg\mathit{nonLast}(X)
\end{aligned}
$$

where domain is the active domain for it collects all constants appearing in the database. In the above program, rule r_2 is used to select nondeterministically an element from the domain whereas rule r_5 is used to construct a chain. In particular, the atoms $\mathit{choice}((X),(Y))$ and $\mathit{choice}((Y),(X))$ are used to enforce the constraint that every element is preceded and followed by a unique element; the conjunction

$min(Z), Y \neq Z$ is used to avoid reintroducing the first element in the chain (this condition is necessary since the first element in the chain is selected by means of a different choice rule). Finally, rule r_7 computes the maximum as the last element in the chain.

2. Membership derives from Fact 1 since for each $(\exists G, P)$ in $\mathbf{Q}^{\neg_s,c}$, $(\exists G, sv(P))$ is in $\mathbf{Q}^{\neg}$ and $\mathcal{EXP}(\exists G, sv(P)) = \mathcal{EXP}(\exists G, P)$. For the completeness, we take any $\mathcal{NP}$ recognizable database collection, say $\mathbf{D}$ on the database scheme DS, and we follow the lines of the proof of Theorem 1 but this time we use a $\mathsf{DATALOG}^{\neg_s,c}$ program P defined in the following way:

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

Obviously, $DS_P = DS$ and $\mathbf{D} \subseteq \mathbf{D}_P$. It is easy to see that, given any database D in $\mathbf{D}_P$, the query $(\exists \neg g$,P) on D is true iff D belongs to $\mathbf{D}$.

3. Membership can be proved as in the proof of Part 2. For the completeness, consider any co$\mathcal{NP}$ recognizable database collection, say $\mathbf{D}'$ on the database scheme DS. Let $\mathbf{D}$ be the data collection of all databases on DS that are not in $\mathbf{D}'$; then $\mathbf{D}$ is recognizable. Consider now the program P in the above proof of Part (2) and the query $Q = (\forall g, P)$. Then, given any database D in $\mathbf{D}_P$, $Q(D)$ is true under certain semantics iff D does not belong to $\mathbf{D}$, i.e. D is in $\mathbf{D}'$.
4. It follows from the fact that definite semantics coincides with certain semantics as the existence of at least one total stable model is guaranteed.

□

The above results fix precise bounds on the expressive powers of $\mathsf{DATALOG}^{\neg_s,c}$. Observe that the proof of Part (1) of Theorem 2 strongly depends on the restriction that $!\mathbf{Q}^{\neg_s,c}$ only contains deterministic queries. Thus this language captures $\mathcal{P}$ but it has the drawback of not being recursively enumerable. This is actually the price we pay in order to capture the whole class $\mathcal{P}$ without having to introduce either an order (thus, loosing genericity) or an exponential-time execution algorithm (thus, loosing efficiency). We leave the programmer with the responsibility of ensuring determinism while writing a query, in the same way s/he is required to guarantee its semantic correctness. We are positive that the simplicity and immediacy of our language will make this task easier.

Example 2 *Tree graph.* We are given an undirected graph whose nodes and arcs are stored in the database relations `node` and `arc`, respectively. A spanning tree starting from any source node can be defined by the following program ST:

$$
\begin{array}{ll}
root(X) \leftarrow & node(X),\ choice((),(X)) \\
reached(X) \leftarrow & root(X) \\
reached(X) \leftarrow & spanTree(Y,X) \\
spanTree(X,Y) \leftarrow & reached(X),\ arc(X,Y),\ root(S), \\
 & Y \neq S,\ choice(Y,X) \\
nonTree \leftarrow & node(X),\ \neg\, reached(X) \\
nonTree \leftarrow & arc(X,Y),\ \neg\, spanTree(X,Y), \\
 & \neg\, spanTree(Y,X).
\end{array}
$$

The choice in the first rule enables the nondeterministic selection of the root of the spanning tree, while the choice in the fourth rule ensures that no node has two incoming arcs in $spanTree$. It turns out that the graph is a tree if and only if the query $(!\neg\texttt{nonTree}, ST)$ has answer $true$. It is easy to recognize that the program shown in Sect. 1 is the standard version of ST, simplified by the application of some foldings and the usage of a deterministic selection of the root.

The query of the above example is obviously deterministic. To appreciate the relevance of nondeterministic semantics, observe that the same query also returns the correct answer under the possible semantics. But computation under possible semantics is efficient only if the graph is a tree. Otherwise, the possible semantics insists in trying to select all other possible spanning trees (and there can be an exponential number of these), while the nondeterministic semantics stops after the first failure.

Example 3 *Hamiltonian path.* The following $\mathsf{DATALOG}^{\neg_s,c}$ program HPP computes a simple path in G (predicate $pathArc$) and checks that all nodes are in the path (predicates hp and $nonHpath$).

$$
\begin{array}{ll}
firstNode(X) \leftarrow & node(X),\ choice((),(X)) \\
reached(X). \leftarrow & firstNode(X) \\
reached(X) \leftarrow & pathArc(Y,X) \\
pathArc(X,Y) \leftarrow & reached(X),\ arc(X,Y),\ firstNode(S),\ Y \neq S, \\
 & choice((X),(Y)),\ choice((Y),(X)) \\
nonHpath \leftarrow & node(X),\ \neg\, reached(X).
\end{array}
$$

The graph has a Hamiltonian path iff the query $(\exists\neg nonHpath, HPP)$ has answer $true$. Observe that the program shown in Example 1 is a "simplified" standard version of HPP.

Hamiltonian circuit. Let us now verify whether there exists a Hamiltonian circuit in the graph i.e. there is both a Hamiltonian path and an arc from the last node in the path to the first node. Consider the following program HCP, obtained from the program HPP by simply adding the following two rules:

$$\begin{array}{ll} nonLastNode(X) \leftarrow & pathArc(X, Y) \\ hCircuit \leftarrow & \neg\, nonHpath,\ node(E),\ \neg\, nonLastNode(E), \\ & firstNode(S),\ arc(E, S) \end{array}$$

where the rule defining the predicate $hCircuit$ checks whether the simple path can be extended into a circuit. Thus, the query $(\exists hCircuit, HCP)$ is $true$ iff the graph has a Hamiltonian circuit. □

As our language guarantees the existence of stable models, definite semantics collapses into certain semantics and, then, it can no longer capture an interesting class such as $\mathcal{D}^p$, which includes many practical decision problems. We next show that a high expressive power can be preserved and even increased by combining the first three semantics: nondeterministic, possible and certain.

Definition 2. *A* compound query goal *is defined inductively as follows:*

1. *any query goal* $!A$, $\exists A$ *or* $\forall A$, *is also a compound query goal;*
2. *if* G_1 *and* G_2 *are two compound query goals, then* $\neg(G1)$, $(G_1 \wedge G_2)$ *and* $(G_1 \vee G_2)$ *are compound query goals.* □

Definition 3. *A (*$\mathsf{DATALOG}^{\neg s,c}$*)* compound query *is a pair* $Q = (G, P)$ *where* G *is a compound query goal and* P *is a* $\mathsf{DATALOG}^{\neg s,c}$ *program. Given a database* $D \in \mathbf{D}_P$, *the answer of* Q *on* D, *denoted by* $Q(D)$, *is* $true$ *if*

- G *is a query goal and* $(G, P)(D)$ *has answer* $true$, *or*
- $G = \neg(G_1)$ *and* $(G_1, P)(D)$ *has answer* $false$, *or*
- $G = (G_1 \wedge G_2)$ *and both* $(G_1, P)(D)$ *and* $(G_2, P)(D)$ *have answer* $true$, *or*
- $G = (G_1 \vee G_2)$ *and* $(G_1, P)(D)$ *or* $(G_2, P)(D)$, *have answer* $true$,

otherwise $Q(D)$ *has answer* $false$. □

Let $(!\exists\forall)^\star\mathbf{Q}^{\neg s,c}$ be the set of all compound queries that are deterministic, i.e. queries yielding exactly one answer. We next show that the expressive power of $(!\exists\forall)^\star\mathbf{Q}^{\neg s,c}$ captures the whole query hierarchy $\mathcal{QH}$, defined as $\mathcal{QH} = \cup_{i=0}^{\infty}\mathcal{QH}_i$, where $\mathcal{QH}_i = \mathcal{P}^{\mathcal{DB}\text{-}\mathcal{NP}[i]}$ is the class of all languages which can be recognized by an $\mathcal{NP}$-oracle Turing machine with i queries

of the oracle [18,27]. Thus, the query hierarchy consists of all database collections that can be recognized by an $\mathcal{NP}$-oracle Turing machine with a constant number of queries to the oracle. It is well known that $\mathcal{QH}$ coincides with the Boolean hierarchy $\mathcal{BH}$, defined as follows [18,27]: i) $\mathcal{BH}_0 = \mathcal{P}$; ii) $\mathcal{BH}_i$, $i > 0$, is the set of all languages expressible as $X - Y$, where $X \in \mathcal{NP}$ and $Y \in \mathcal{BH}_{i-1}$; iii) $\mathcal{BH} = \cup_{i=0}^{\infty}\mathcal{BH}_i$.

Theorem 3. $\mathcal{EXP}((!\exists\forall)^\star\mathbf{Q}^{\neg s,c}) = \mathcal{DB}\text{-}\mathcal{QH}$.

Proof. Take any query $Q = (G, P)$ in $(!\exists\forall)^\star\mathbf{Q}^{\neg s,c}$. Let k be the number of simple query goals in G; according to the data complexity assumption, k is to be considered a constant. Take now any database D in $\mathbf{D}_P$. Answering $Q(D)$ can be done by (i) answering the k subqueries and (ii) replacing each simple query goal in G with the answer of the corresponding subquery and, then, computing the resulting boolean expression. As every subquery is either in $\mathcal{NP}$ or in co$\mathcal{NP}$ by Theorem 2, Step (i) can be done in polynomial time by consulting an $\mathcal{NP}$oracle k times. Since Step (ii) is easily implemented in polynomial time, it turns out that Q is in $\mathcal{QH}$; so $\mathcal{EXP}((!\exists\forall)^\star\mathbf{Q}^{\neg s,c}) \subseteq \mathcal{DB}\text{-}\mathcal{QH}$.

Let us now prove that $\mathcal{DB}\text{-}\mathcal{QH} \subseteq \mathcal{EXP}((!\exists\forall)^\star\mathbf{Q}^{\neg s,c})$. To this end, as $\mathcal{QH} = \mathcal{BH}$, it is sufficient to show that $\mathcal{DB}\text{-}\mathcal{BH}_i \subseteq \mathcal{EXP}((!\exists\forall)^\star\mathbf{Q}^{\neg s,c})$ for each $i > 0$. We proceed by induction on i:

($i = 1$ – *basis of the induction*) The fact that $\mathcal{DB}\text{-}\mathcal{BH}_1 = \mathcal{DB}\text{-}\mathcal{NP} \subseteq \mathcal{EXP}((!\exists\forall)^\star\mathbf{Q}^{\neg s,c})$ follows from Theorem 2.

($i > 1$ – *induction*) By *induction hypothesis*, $\mathcal{DB}\text{-}\mathcal{BH}_{i-1} \subseteq \mathcal{EXP} \times((!\exists\forall)^\star\mathbf{Q}^{\neg s,c})$; we have to prove that also $\mathcal{DB}\text{-}\mathcal{BH}_i \subseteq \mathcal{EXP}((!\exists\forall)^\star\mathbf{Q}^{\neg s,c})$. Consider any database collection $\mathbf{D}$ in $\mathcal{DB}\text{-}\mathcal{BH}_i$ and say that DS is the scheme of the databases in $\mathbf{D}$. By definition of $\mathcal{DB}\text{-}\mathcal{BH}_i$, there exist two database collections on DS, say $\mathbf{D}'$ and $\mathbf{D}''$, such that $\mathbf{D}' \in \mathcal{DB}\text{-}\mathcal{NP}$, $\mathbf{D}'' \in \mathcal{DB}\text{-}\mathcal{BH}_{i-1}$ and $\mathbf{D} = \mathbf{D}' - \mathbf{D}''$. Hence there exist two queries, say $Q' = (G', P')$ and $Q'' = (G'', P'')$, such that $DS_{P'} = DS_{P''} = DS$, $\mathcal{EXP}(Q') = \mathbf{D}'$ and $\mathcal{EXP}(Q'') = \mathbf{D}''$. Without loss of generality, assume that IDB predicates in P' and P'' are distinct. Consider the query $Q = (G, P)$, where $P = P' \cup P''$ and $G = G' \wedge \neg(G'')$. Let $\overline{\mathbf{D}''}$ be the data collection of all databases on DS that are not in $\mathbf{D}''$. Then $\mathcal{EXP}((\neg(G''), P)) = \overline{\mathbf{D}''}$. It follows that $\mathcal{EXP}(Q) = \mathbf{D}' \cap \overline{\mathbf{D}''}$. But $\mathbf{D}' \cap \overline{\mathbf{D}''} = \mathbf{D}' - \mathbf{D}''$ by definition; so $\mathcal{EXP}(Q) = \mathbf{D}$. Hence, $\mathcal{DB}\text{-}\mathcal{BH}_i \subseteq \mathcal{EXP}((!\exists\forall)^\star\mathbf{Q}^{\neg s,c})$. This concludes the proof. □

As shown next, a recursive subclass of $(!\exists\forall)^\star\mathbf{Q}^{\neg s,c}$ which preserves the same expressive power is $(\exists\forall)^\star\mathbf{Q}^{\neg s,c}$, consisting of all compound queries with no nondeterministic subqueries. Let $(\exists\forall)^1\mathbf{Q}^{\neg s,c}$ be the set of all compound queries of the form $((\exists G_1 \wedge \forall G_2), P)$. Obviously, also $(\exists\forall)^1\mathbf{Q}^{\neg s,c}$ is recursive.

Corollary 1. *$\mathcal{EXP}((\exists\forall)^1\mathbf{Q}^{\neg_s,c}) = \mathcal{DB}$-$\mathcal{QH}$ and $\mathcal{EXP}((\exists\forall)^1\mathbf{Q}^{\neg_s,c}) = \mathcal{DB}$-$\mathcal{D}^p$.*

Proof. It follows from the facts that (i) nondeterministic subqueries are never used in the proof of Theorem 3 and (ii) $\mathcal{D}^p = \mathcal{BH}_2$. □

Example 4 *Unique Hamiltonian path.* We want to verify whether a graph has a unique Hamiltonian simple path. We modify the DATALOG$^{\neg_s,c}$ program of Example 3 into the program UHP below by enabling the selection of two simple paths in the graph rather than one. The two paths are identified by the labels 1 and 2, respectively, which are stored in the first argument of the predicate $pathArc$ and are defined by the first two rules. The third rule nondeterministically selects a node which will be the first node of both paths. The rules $r_4 - r_7$ select the two simple paths. Rule r_8 verifies whether the two paths are distinct. The last rule checks whether we have selected two distinct Hamiltonian simple paths.

$(r_1)\ label(1)$
$(r_2)\ label(2)$
$(r_3)\ firstNode(X) \leftarrow \quad node(X),\ choice((),(X))$
$(r_4)\ reached(L,X) \leftarrow \quad label(L),\ firstNode(X)$
$(r_5)\ reached(L,X) \leftarrow \quad pathArc(L,Y,X)$
$(r_6)\ pathArc(L,X,Y) \leftarrow reached(L,X),\ arc(X,Y),$
$\qquad firstNode(L,S),\ Y \neq S,$
$\qquad choice((X),(Y)),\ choice((Y),(X))$
$(r_7)\ nonHpath(L) \leftarrow \quad label(L),\ node(X),\ \neg reached(L,X)$
$(r_8)\ distinct \leftarrow \quad pathArc(1,X,Y1),\ pathArc(2,X,Y2),$
$\qquad Y1 \neq Y2$
$(r_9)\ twoHpaths \leftarrow \quad \neg nonHpath(1),\ \neg nonHpath(2),\ distinct.$

The query $(\exists(\neg nonHpath(1)), UHP)$ solves the $\mathcal{NP}$-complete problem of whether the graph has a Hamiltonian path. The query $(\forall(\neg twoHpaths), UHP)$ solves the co$\mathcal{NP}$-complete problem of checking whether the graph has at most one Hamiltonian path. Finally the query $((\exists\neg nonHpath(1)) \wedge \forall(\neg twoHpaths), UHP)$ solves the $\mathcal{D}^p$ problem of whether the graph has a unique Hamiltonian path – note that this problem has not been proven to be $\mathcal{D}^p$-complete and it belongs to the class $\mathcal{US}$, i.e. problems with a unique solution [18, 27].

Proposition 2. *Given any query $Q = (G,P) \in (!\exists\forall)^\star\mathbf{Q}^{\neg_s,c}$ and any database $D \in DS_P$,*

1. *$Q(D)$ can be solved in time $\mathcal{O}(n \times poly(||D||))$, and*
2. *if $G =!A$ then $Q(D)$ can be solved in time $O(poly(||D||))$,*

where $poly(||D||)$ is a polynomial in the size of D and $n = \mathcal{O}(2^{||D||})$ is the number of choice models of the program $P(D)$.

Proof. A choice model of $P(D)$ can be computed in polynomial time (see the proof of Part 1 of Theorem 2). Finding one choice model is sufficient to solve the query when $G = !A$. In the other cases, at worst we have to find all choice 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 choice model in polynomial time. Hence, as the number n of choice models is exponential in the size of the data, $Q(D)$ is computed in $\mathcal{O}(n \times poly(||D||))$ time. □

Proposition 2 clarifies the relevance 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 with no restriction on the usage of unstratified negation. However, computing a choice model is polynomial-time, while computing a stable model in an unstratified program is exponential-time. Thus, by using choice models, answering queries under the possible or certain semantics remains exponential, but polynomial problems can now be computed in polynomial time, by requesting a nondeterministic semantics. Again, we point out that we are considering deterministic queries.

5 Implementation

In this section we shall provide an implementation scheme for the computation of any $\mathsf{DATALOG}^{\neg_s,c}$ query $Q = (G, P)$. For the sake of presentation, we first consider the case that P is a semipositive program.

Given a semipositive $\mathsf{DATALOG}^{\neg_s,c}$ program P, we denote with P_C the set of chosen rules in $sv(P)$ i.e.

$$P_C = \{chosen_r(W) \leftarrow B(Z), \neg diffChoice_r(W).|r \text{ is a choice rule in } P\} .$$

Given an interpretation I, we say that I is P_C*-consistent* if for each predicate $chosen_r$ in P_C, the relation $\{w|\ chosen_r(w) \in I\}$ satisfies FD_r i.e. all the functional dependencies defined by the choice constructs in the rule r. Moreover, P_D denotes the set of remaining rules in $sv(P)$ i.e. $P_D = sv(P) - P_C$ consists of the modified choice rules, the diffChoice rules plus all the non-choice rules of P. Thus non-stratified negations are now isolated into P_C.

We define an operator Z_P for P as

$$\forall I \subseteq B_{sv(P)},\ Z_P(I) = Q_P(I) \cup any(\gamma_P(Q_P(I))),$$

```
GoalHandler GH;
bool solveSPQuery ( Program P, Goal G, SetOf⟨Atom⟩ D ) {
        GH.setGoal(G);
        computeZ(P, D);
        GH.done();
        return GH.verified();
};
void computeZ ( Program P, SetOf⟨Atom⟩ M ) {
        SetOf⟨Atom⟩ M1, M2;
        M1 = Q(P, M);
        ListOf⟨Atom⟩ LA = γ(P, M1);
        if ( LA.empty() )
                GH.verify(M1);
        else
            while( !LA.empty() && !GH.verified() && !GH.verified()) {
                        SetOf⟨Atom⟩ M2 = M1; M2.insert(LA.first());
                        computeZ(P, M2);
                        LA.pop();
                }
};
```

Fig. 1. Query evaluation for a semipositive DATALOG$^{\neg s,c}$ program

where $Q_P(I) = I \cup T_{P_D}^{\uparrow\omega}(I)$ is the inflationary all-immediate-consequences operator for P_D (and, as such, it captures the deterministic part of the program P), γ returns the set of all possible choices in P_C, and any is a nondeterministic operator returning any of the above choices.

The operator γ is defined as:

$$\gamma_P(I) = \{A \mid A \in T_{P_C}(I) - I \text{ s. t. } I \cup \{A\} \text{ is } P_C\text{-consistent}\}$$

where T_{P_C} is the immediate consequence operator for P_C. Note that implementing the operator γ_P can be done by storing the $chosen_r$ relations and the functional dependencies in FD_C; from these, the *diffChoice* predicates can be generated on the fly, thus eliminating the need to store them explicitly. Moreover, this implementation scheme provides a simple enough metaphor for a programmer to make effective use of this construct without having to become cognizant of the subtleties of nonmonotonic semantics [11, 12].

Because of its inflationary structure, the least fixpoint $Z_P^{\uparrow\omega}(\emptyset)$ exists; moreover, it coincides with a choice model of P [11]. Thus a choice model of P can be computed by alternating between the application of a nondeterministic selection of one of the choices returned by the operator γ_P (i.e. the firing of an instance of a $chosen$ rule) and the firing of all consequences of such a choice by means of Q_P till saturation. In order to compute all choice models, we have to backtrack to each choice selection and perform a different selection until no other choices are available.

The computation of a semipositive DATALOG$^{\neg s,c}$ query is done by the two C++ functions in Fig. 1. The two functions use the global object GH which handles the compound query goal. The method $setGoal(G)$ of GH stores the compound goal G in a suitable data structure and constructs a three-valued ($true$, *false*, *undefined*) expression E_G by replacing each simple subquery with a three-valued variable and assigning *undefined* to each variable; $verify(M)$ assigns $true$ (resp., *false*) to the *undefined* variables corresponding to all "!B" or "$\exists B$" (resp., "$\forall B$") subqueries of G for which B is true (resp., false) in M; $done()$ assigns *false* (resp., $true$) to the *undefined* variables corresponding to all "!B" or "$\exists B$" (resp., "$\forall B$") subqueries of G; $verified()$ returns $true$ if E_G is made true by the current variable assignments (thus, the compound goal is already satisfied), and $unverified()$ returns $true$ if E_G is made false (thus, the compound goal cannot be satisfied by any subsequent assignment). Recall that $t \vee u = t$, $f \vee u = u$, $t \wedge u = u$, and $f \wedge u = f$, where $t = true$, $f =$ *false* and $u =$ *undefined* [29].

The function $solveSPQuery$ initializes the goal handler and calls the function $computeZ$ for computing the choice models that are necessary to answer the query; then, soon after having notified the end of the choice models computation to the goal handler, it returns the query result. The function *computeZ* computes $M1 = Q_P(M)$ and then collects all elements of $\gamma_P(M1)$ into the list LA. If LA happens to be empty then $M1$ is a choice model and it is submitted for evaluation to the goal handler. Otherwise, all atoms in LA are taken out, one after the other; at each step, an atom A in the list is added to all atoms in $M1$ thus obtaining $M2 = Z_P(M1)$ and, then, $computeZ$ is recursively invoked to compute $Z_P(M2)$, and so on until a choice model is eventually determined. When the function takes back the control, it moves to the next element in the list LA to compute further choice models. The function is terminated when the list LA becomes empty or as soon as the goal handler realizes that the compound goal has already been satisfied or can never be satisfied.

We are now ready to show the implementation of a general DATALOG$^{\neg s,c}$ query $Q = (G, P)$, thus removing the assumption that P is semipositive. The new computation is shown in Fig. 2. The function $solveQuery$ finds a stratification for P [1] and divides P into a number of subprograms $P_0, \ldots, P_{n-1}$, each corresponding to a stratum – the subprograms are stored in the array AP. Then the fixpoint computation is started, one stratum after the other, from the lowest stratum 0 up. The computation for each subprogram $AP[i]$ is carried out as for a semipositive program except for the statement just after no more choice is detected: the function $computeZ$ now invokes the method $GH.verify(M1)$ only if $AP[i]$ is the last stratum (i.e. $i = n - 1$) or otherwise it activates the fixpoint computation of the next stratum.

```
GoalHandler GH;
bool solveQuery ( Program P, Goal G, SetOf⟨Atom⟩ D ) {
        ArrayOf⟨Program⟩ AP = computeStrata(P);
        GH.setGoal(G);
        computeZ(AP, D, 0);
        GH.done();
        return GH.verified();
};
void computeZ ( ArrayOf⟨Program⟩ AP, SetOf⟨Atom⟩ M, int n ) {
        SetOf⟨Atom⟩ M1, M2;
        M1 = Q(AP[i], M);
        ListOf⟨Atom⟩ LA = γ(AP[i], M1);
        if ( LA.empty() )
            if ( i == AP.length() − 1 )
                GH.verify(M1);
            else
                computeZ(AP, M1, i + 1);
        else
            while( !LA.empty() && !GH.verified() && !GH.verified()) {
                SetOf⟨Atom⟩ M2 = M1; M2.insert(LA.first());
                computeZ(AP, M2, i);
                LA.pop();
            }
};
```

Fig. 2. Query evaluation for a semipositive DATALOG$^{\neg s,c}$ program

Proposition 3. *Let any query $Q = (G, P) \in (!\exists\forall)^\star\mathbf{Q}^{\neg s,c}$ be fixed. Then, given any database $D \in DS_P$,*

1. *$solveQuery(P, G, D)$ is computed in time $\Theta(n \times poly(||D||))$, and*
2. *if $G = !A$ then $solveQuery(P, G, D))$ is computed in time $\Theta(poly(||D||))$,*

where $poly(||D||)$ is a polynomial in the size of D and $n = O(2^{||D||})$ is the number of choice models of the program $P(D)$.

Proof. Computing a stratification of a program P can be done in time polynomial in the size of P [1]. Therefore, the complexity of $solveQuery$ is determined by the complexity of the function $computeZ$. Each simple execution of $computeZ$ (i.e. without counting the time spent by the nested recursive executions) is done in time polynomial in the size of the Herbrand base of $sv(P)$, which is in turn polynomial in the size of D. So, in the general case, the overall time complexity is $\Theta(k \times p_1(||D||))$, where $p_1(||D||)$ is some polynomial in the size of D and k is the number of executions of $computeZ$. Obviously k is equal to $p_2(||D||) \times l$, where $p_e(||D||)$ is another polynomial in the size of D and l is the number of leaf nodes in the backtracking search tree of $computeZ$. As each leaf produces a distinct choice model,

Query	Complexity	Reference
$!\mathbf{Q}^{\neg}$	$\mathcal{DB}$-$\mathcal{NP}$	this paper
$\exists\mathbf{Q}^{\neg}$	$\mathcal{DB}$-$\mathcal{NP}$	[24]
$\forall\mathbf{Q}^{\neg}$	$\mathcal{DB}$-co$\mathcal{NP}$	[24, 35]
$\exists\forall\mathbf{Q}^{\neg}$	$\mathcal{DB}$-$\mathcal{D}^p$	[31]
$!\mathbf{Q}^{\neg s,c}$	$\mathcal{DB}$-$\mathcal{P}$	this paper
$\exists\mathbf{Q}^{\neg s,c}$	$\mathcal{DB}$-$\mathcal{NP}$	this paper
$\forall\mathbf{Q}^{\neg s,c}$	$\mathcal{DB}$-co$\mathcal{NP}$	this paper
$\exists\forall\mathbf{Q}^{\neg s,c}$	$\mathcal{DB}$-co$\mathcal{NP}$	this paper
$(\exists\forall)^1\mathbf{Q}^{\neg s,c}$	$\mathcal{DB}$-$\mathcal{D}^p$	this paper
$(!\exists\forall)^\star\mathbf{Q}^{\neg s,c}$	$\mathcal{DB}$-$\mathcal{QH}$	this paper

Fig. 3. Expressive power of DATALOG$^{\neg}$ and DATALOG$^{\neg s,c}$

k coincides with n. So $solveQuery$ is computed in time $\Theta(n \times poly(||D||))$, where $poly(||D||)$ is the polynomial $p_1(||D||) \times p_2(||D||)$. Let us now assume that $G =!A$. In this case, the execution stops as soon as the first leaf node is reached in the search; so $k = 1$ and, then, the time complexity becomes $\Theta(poly(||D||))$.

Thus, the choice construct allows us to capture a special subclass of DATALOG$^{\neg}$ programs that have a very expressive stable model semantics but are amenable to efficient implementation and are appealing to intuition.

6 Conclusion

The table in Fig. 3 summarizes the results here obtained on 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, an intuitive appeal and have an efficient implementation algorithm. In fact, the class of queries for which we have presented an effective implementation scheme achieves the same level of expressive power as the query hierarchy $\mathcal{QH}$; this is higher than the expressive power of classical queries under the possible or certain semantics.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases, Reading, MA: Addison-Wesley, 1995.
2. Abiteboul, S., E. Simon, E., V. Vianu, V.: Non-Deterministic Languages to Express Deterministic Transformations In: Proc. of the Ninth ACM PODS Conference, pp 215–229, 1990.
3. Abiteboul, S., V. Vianu, V.: Datalog Extensions for Databases Queries and Updates. J Comput Syst Sci 43, 62–124, 1991.
4. Afrati, F., Cosmadakis, S.S., Yannakakis, M.: On Datalog vs. Polynomial Time. In: Proc. of the Tenth ACM PODS Conference, 13–25, 1991.
5. Apt, C., Blair, H., Walker, A.: Towards a Theory of Declarative Knowledge. In: Minker, J. (ed.) Proc. Work. on Found. of Deductive Database and Logic Prog. 89–148, 1988.
6. Chandra, A., Harel, D.: Structure and Complexity of Relational Queries. J Comput Syst Sci 25(1), 99–128, 1982.
7. Chandra, A., Harel, D.: Horn Clauses Queries and Generalizations. J Logic Program 2(1), 1–15, 1985.
8. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Trans Database Syst 22(3), 364–418, 1997.
9. Fagin, R.: Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In: Karp, R. (ed.) Complexity of Computation, SIAM-AMS Proceedings, Vol. 7, 43–73, 1974.
10. Gelfond, M., Lifschitz, V.: The Stable Model Semantics of Logic Programming. Proc. of the Fifth Intern. Conf. on Logic Programming, 1070–1080, 1988.
11. Giannotti, F., Pedreschi, D., Saccà, D., Zaniolo, C.: Nondeterminism in Deductive Databases. Proc. 2nd DOOD Conference, pp 129–146, 1991.
12. Giannotti, F., Pedreschi, D., Zaniolo, C.: Semantics and Expressive Power of Non-Deterministic Constructs in Deductive Databases. J Comput Syst Sci, 2001 (to appear).
13. Greco S., Saccà D.: "Possible is Certain" is Desirable and Can Be Expressive. Ann Math Artif Intell 019(1-2), 147–168, 1997.
14. Greco S., Saccà D., Zaniolo, C.: DATALOG Queries with Stratified Negation and Choice: from $\mathcal{P}$ to $\mathcal{D}^p$, In *Proc. of the Fifth Int. Conf. on Database Theory*, pp 82–96, 1995.
15. Greco, S., Zaniolo, C., Ganguly, S.: Greedy by Choice. Proc. of the Eleventh ACM PODS Conference, 105–163, 1992.
16. Greco, S., Saccá, D.: Complexity and Expressive Power of Deterministic Semantics for DATALOG$^{\neg}$. Inform Comput 153(1), 81–98, 1999.
17. Immerman, N.: Languages that Capture Complexity Classes. SIAM J Comput 16(4), 760–778, 1987.
18. Johnson, D. S.: A Catalog of Complexity Classes. In: van Leeuwen, J. (ed) Handbook of Theoretical Computer Sciences, Vol. A, North-Holland, pp 67–161, 1990.
19. Kanellakis, P. C.: Elements of Relational Databases Theory. In: van Leeuwen, Z. (ed) Handbook of Theoretical Computer Sciences, Vol. B, North-Holland, 1075–1155, 1990.
20. Kolaitis, P.: The Expressive Power of Stratified Logic Programs. Infor Comput 90, 50–66, 1990.
21. Kolaitis, P., Papadimitriou, C.: Why not Negation by Fixpoint? J Comput Syst Sci 43, 125–144, 1991.
22. Eiter, T., Leone, N., Mateis, C., Pfeifer G., Scarcello, F.: A Deductive System for Nonmonotonic Reasoning. Proceedings International on Logic Programming and Nonmonotonic Reasoning, pp 363–374, 1997.
23. Lloyd, J.L.: Foundations of Logic Programming. Berlin: Springer 1987.

24. Marek, W., Truszczynski, M.: Autoepistemic Logic. Journal of the ACM 38(3), 588–619, 1991.
25. Naqvi, S., Tsur, S.: A Logic Language for Data and Knowledge Bases. Computer Science Press, 1989.
26. Papadimitriou, C.: A Note on the Expressive Power of Prolog. Bull of the EATCS 26, 21–23, 1985.
27. Papadimitriou, C.: Computational Complexity. Reading, MA: Addison-Wesley, 1994.
28. Przymusinska, A., Przymusinski, T.: Weakly Perfect Model Semantics for Logic Programs. Proceedings of the Fifth Intern. Conference on Logic Programming, 1106–1122, 1988.
29. Przymusinski T.C.: Well-founded Semantics Coincides with Three-valued Stable Semantics. Foundamenta Informaticae 13, 445–463, 1990.
30. Ramakrisnhan, R., Srivastava, D., Sudanshan, S.: CORAL – Control, Relations and Logic. Proc. of 18th Conf. on Very Large Data Bases, 238–250, 1992.
31. Saccà D.: Multiple total stable models are definitely needed to solve unique solution problems. Inform Process Lett 58, 249–254, 1996.
32. Saccà, D.: The Expressive Powers of Stable Models for Bound and Unbound DATALOG Queries. J Comput Syst Sci 54(3), 441–464, 1997.
33. Saccà, D., Zaniolo, C.: Stable Models and Non-Determinism in Logic Programs with Negation. Proc. ACM PODS Symp., 205–218, 1990.
34. Saccà, D., Zaniolo, C.: Stable models and nondeterminism in logic programs with negation. *J Logic Comput* 7(5), 555–579, 1997.
35. Schlipf, J.S.: The Expressive Powers of the Logic Programming Semantics. J Comput Syst Sci 51(1), 64–86, 1995.
36. Schlipf, J.S.: Complexity and Undecidability Results for Logic Programming. Ann Math Artif Intell 15(3-4), 257–288, 1995.
37. Ullman, J.D.: Principles of Databases and Knowledge Base Systems, Vol. I and II. Computer Science Press, 1988.
38. Van Gelder, A.: Negation as failure using tight derivations for general logic programs. J Log Program 6(1), 109–133, 1989.
39. Van Gelder, A., Ross, K.A., Schlipf, J.S.: The well-founded semantics for general logic programs. Journal of the ACM 38(3), 620–650, 1991.
40. Van Gelder, A.: The Alternating Fixpoint of Logic Programs with Negation. J Comput Syst Sci 43(1), 185–221, 1992.
41. Vardi, M.: The Complexity of Relational Query Languages. Proceedings of the 14th ACM Symposium on Theory of Computing, 137–146, 1982.