Why Not Negation by Fixpoint?

PHOKION G. KOLAITIS

Computer and Information Sciences, University of California, Santa Cruz, Santa Cruz, California 95064

AND

CHRISTOS H. PAPADIMITRIOU*

Computer Science and Engineering, University of California, San Diego, La Jolla, California 92093

Received September 22, 1988; revised March 26, 1990

There is a fixpoint semantics for DATALOG programs with negation that is a natural generalization of the standard semantics for DATALOG programs without negation. We show that, unfortunately, several compelling complexity-theoretic obstacles rule out its efficient implementation. As an alternative, we propose Inflationary DATALOG, an efficiently implementable semantics for negation, based on inflationary fixpoints. © 1991 Academic Press, Inc.

1. INTRODUCTION

The history of negation in logic programming has been long and controversial. The implementation of the negation as failure semantics (Clark [Cl78], Apt and van Emden [AvE82]) depends on the order of the literals in a clause. In database applications, however, such dependence is unnatural and undesirable. To remedy the situation, Chandra and Harel [CH85] proposed a semantics for stratified programs with negation, whereby one can evaluate programs in which relational symbols are divided into layers and a relation may use negatively only relations on lower layers in its definition(s). The study of stratified programs has been pursued more recently by Apt, Blair, and Walker [ABW86], Van Gelder [VG86], and others. It should be pointed out, however, that not all DATALOG \neg programs (logic programs without function symbols, but with negation) can be assigned meaning under this semantics.

Any DATALOG \neg program can be thought of as a mapping from relation values to relation values. Given a set of appropriate values for both the database and the

* The research of this author was supported by an NSF grant.
non-database relations (where the non-database relations use elements that also appear in the database ones), we compute the non-database relations as follows: We start from the empty relations, iterating through all possible values for the variables in all rules, adding to the relation in the left-hand side each tuple that makes the right-hand side true. If we obtain precisely the non-database relations with which we started, we say that the present values of the non-database relations constitute a fixpoint for the given values of the database relations.

A DATALOG program is a DATALOG\(^-\) program with the additional condition that no negation occurs in the right-hand side of the rules. The standard semantics of a DATALOG program on a database is the least fixpoint of the program on the given database. This is guaranteed to exist, because the program gives rise to a monotone mapping from relation values to relation values. In contrast to this, given a DATALOG\(^-\) program and database values there may be no fixpoint, or a unique fixpoint (and hence a least fixpoint), or even several fixpoints. In the last case, a least fixpoint may or may not exist.

In view of the above, it is tempting to define the semantics of any DATALOG\(^-\) program to be, given the values for the database relations, the least fixpoint of the program for these values, if it exists, or some other standard value otherwise. Another natural extension of the DATALOG semantics to (non-deterministic) DATALOG\(^-\) semantics would be to assign some fixpoint of the program, if one exists, or a standard value otherwise.

Our main technical results reveal that there is no efficient way to implement either of these two semantics unless \(P=NP\). We show first that there are DATALOG\(^-\) programs for which determining whether a fixpoint exists is a NP-complete problem. Our result is actually a little stronger than establishing NP-completeness: We show that, for any problem in NP, there is a fixed DATALOG\(^-\) program such that, given the input to the problem as database relations, the program has a fixpoint if and only if the input is a "yes" instance. In other words, existence of fixpoints is a normal form for NP. After this, we examine the problem of telling whether or not a unique fixpoint exists (a desirable situation in which fixpoint semantics becomes deterministic). We show that there are fixed DATALOG\(^-\) programs for which existence of unique fixpoints on given data is a complete problem for the class US of problems having unique solution (typical problem: given a graph does it have a unique Hamilton circuit?). We also establish that existence of a least fixpoint, for fixed DATALOG\(^-\) programs, is even harder. At present, we can not pinpoint exactly the complexity of this problem, but we show that it lies between the class US from the lower end, and the class FO\(^{NP}\) (for first-order with NP oracles) from the upper end, a new subclass of \(A_2^{NP}\) that seems to be of interest in its own right. Finally, what if the program is not fixed, but is part of the input? We show that in this case the problem of telling whether a fixpoint exists becomes complete for nondeterministic exponential time (and thus, most probably, requires doubly exponential time). This last result illustrates one more difference between data complexity and expression complexity (cf. Vardi [VA82]).
The research reported here suggests that there are solid computational reasons why ordinary fixpoints should not be used as the semantics of negation, despite their naturalness. In the last section we propose an alternative semantics for DATALOG\textsuperscript{−} programs based on the inflationary semantics studied by Gurevich and Shelah [GS86] on finite databases and before that by several researchers on infinite structures under the name nonmonotone inductive definability (cf. Aczel [Ac77]). The advantages of this semantics are that it agrees with the standard one for DATALOG programs, it retains the computational intuition (bottom-up evaluation) of least fixpoints and, in contrast to the stratified semantics, it assigns meaning to all DATALOG\textsuperscript{−} programs. We show that the expressive power of DATALOG\textsuperscript{−} programs under this semantics (a query language which we call Inflationary DATALOG) is strictly greater than DATALOG and conclude by discussing certain recent developments concerning the expressive power of database query languages.

2. DATALOG\textsuperscript{−} Programs and Fixpoints

A DATALOG\textsuperscript{−} program $\pi$ is a finite set of rules. Each rule is of the form

$$t_0 \leftarrow t_1, t_2, ..., t_l,$$

where the $t_i$'s are literals. The literal $t_0$ is the head of the rule, the others make up the body. The literals in the body can be equalities $x_i = x_j$, inequalities $x_i \neq x_j$, atomic formulas $Q(x_1, ..., x_m)$, or negated atomic formulas $\neg Q(x_1, ..., x_m)$, where $Q$ is a relational symbol and the $x_i$'s are variables. The head $t_0$ of the rule is an atomic formula $S(x_1, ..., x_n)$, where $S$ is a relational symbol. The database relations of $\pi$ are those relational symbols that do not appear at the head of any rule; those that appear are called nondatabase relations. In the literature, the database relations are often called extensional database relations (EDBs) and the nondatabase ones are called intensional database relations (IDBs).

In order to illustrate these concepts, consider the following one-line DATALOG\textsuperscript{−} program $\pi_1$,

$$T(x) \leftarrow E(y, x), \neg T(y).$$

In this program $E$ is a database relation and $T$ is a nondatabase relation. Similarly, if $\pi_2$ is the DATALOG\textsuperscript{−} program,

$$S_1(x, y) \leftarrow E(x, y)$$

$$S_1(x, y) \leftarrow E(x, z), S_1(z, y)$$

$$S_2(x, y, z, w) \leftarrow S_1(x, y), \neg S_1(z, w),$$

then $E$ is a database relation, while $S_1$ and $S_2$ are non-database relations.
A DATALOG program is a DATALOG\neg program such that no literal in the body of a rule is an inequality or a negated atomic formula. Thus, the program \(\pi_3\),

\[
S(x, y) \leftarrow E(x, y) \\
S(x, y) \leftarrow E(x, z), S(z, y),
\]
is a DATALOG program, while the previous two programs \(\pi_1\) and \(\pi_2\) are not.

In what follows we assume that all databases considered are over an arbitrary but fixed finite vocabulary \(\sigma\), i.e., we have a fixed sequence \(\sigma = (R_1, \ldots, R_j)\) of database relational symbols such that each \(R_i\) is of arity \(m_i\).

Suppose that \(D = (A, R_1, \ldots, R_j)\) is a database over \(\sigma\) having \(A\) as its universe and let \(S_1, \ldots, S_m\) be the nondatabase relations of a DATALOG\neg program \(\pi\). The program \(\pi\) then gives rise to an operator \(\Theta_\pi\), or simply \(\Theta\), which is a mapping from sequences \(\vec{S} = (S_1, \ldots, S_m)\) of relations on \(A\) whose arities match those of the non-database relations of \(\pi\) to sequences of relations of the same arities. Intuitively, for any such sequence \(\vec{S} = (S_1, \ldots, S_m)\) the operator \(\Theta\) returns as values a sequence \(\Theta(\vec{S})\) of relations that are obtained from the relations \(S_1, \ldots, S_m\) and the database relations \(R_1, \ldots, R_j\) by applying the rules of \(\pi\). When applying a rule of \(\pi\), the variables that occur in the body and not in the head of the rule are viewed as being existentially quantified, with the quantifiers positioned in the front of the body.

For simplicity, we give the formal definition of the operator \(\Theta\) for the case in which the program \(\pi\) has a single nondatabase relation \(S\) of arity \(m\). Let \(r_1, \ldots, r_k\) be the rules of \(\pi\) and assume that the rule \(r_i\) is

\[
S(\vec{x}) \leftarrow t_{i_1}, t_{i_2}, \ldots, t_{i_l}.
\]

Moreover, let \(\vec{z}\) be the sequence of all variables that occur in the body and not in the head of \(r_i\). Each rule \(r_i\), \(1 \leq i \leq k\), can be viewed as the formula

\[
(\forall \vec{z})(\forall\vec{x})(t_{i_1} \land t_{i_2} \land \cdots \land t_{i_l}) \rightarrow S(\vec{x}),
\]

which is equivalent to the formula

\[
(\forall \vec{z})[ (\exists \vec{x})(t_{i_1} \land t_{i_2} \land \cdots \land t_{i_l}) \rightarrow S(\vec{x})].
\]

We associate now with the rule \(r_i\) the existential formula \(\theta_i(\vec{x})\), where

\[
\theta_i(\vec{x}) \equiv (\exists \vec{z})(t_{i_1} \land t_{i_2} \land \cdots \land t_{i_l}).
\]

The operator \(\Theta\) is defined in terms of the existential formulas \(\theta_i(\vec{x})\), \(1 \leq i \leq k\), as

\[
\Theta(S) = \{ \bar{a} \in A^m : D \models \bigvee_{i=1}^{k} \theta_i(\bar{a}) \}.
\]

For example, for the program \(\pi_1\) and for a database \(D = (A, E)\), we have that

\[
\Theta(T) = \{ a \in A : (\exists y)(E(y, a) \land \neg T(y)) \}.
\]
Similarly, for the program \( n \), and for a database \( D = (A, E) \), we have that

\[
\Theta(S) = \{(a, b) \in A: E(a, b) \lor (\exists z)(E(a, z) \land S(z, b))\}.
\]

In the case of the program \( r_{c_2} \), the value of the operator \( \Theta \) on a pair of relations \( (S_1, S_2) \) is a pair of relations the first of which is

\[
\{(a, b) \in A^2: E(a, b) \lor \exists z(E(a, z) \land S_1(z, b))\}
\]

and the second is

\[
\{(a, b, c, d) \in A^4: S_1(a, b) \land \neg S_1(c, d)\}.
\]

Consider now a program \( \pi \), a database \( D = (A, R_1, \ldots, R_i) \), and a sequence \( \bar{S} = (S_1, \ldots, S_m) \) of relations on \( A \) whose arities match the arities of the nondatabase relations of \( \pi \). We say that the sequence \( \bar{S} = (S_1, \ldots, S_m) \) is a fixpoint of \( (\pi, D) \) if \( \Theta(\bar{S}) = \bar{S} \). If such a sequence exists, then we say that \( (\pi, D) \) has a fixpoint. When \( \pi \) is fixed and understood, we say that \( D \) has a fixpoint.

A sequence \( \bar{S} = (S_1, \ldots, S_m) \) is a least fixpoint of \( (\pi, D) \) if it is a fixpoint of \( (\pi, D) \) and for every fixpoint \( \bar{S}' = (S'_1, \ldots, S'_m) \) of \( (\pi, D) \), we have that \( S_i \subseteq S'_i \), \( 1 \leq i \leq m \).

If \( \pi \) is a DATALOG program, then \( (\pi, D) \) has a least fixpoint, for every database \( D \). The least fixpoint of \( (\pi, D) \) is the standard semantics of the DATALOG program \( \pi \) on \( D \). The reason DATALOG programs possess least fixpoints is that, since only positive literals appear in the bodies of the rules, the operator \( \Theta \) is monotone and, as is well known (cf. Tarski [Ta55]), every monotone operator has a least fixpoint. For example, the least fixpoint of the DATALOG program \( r_{c_2} \) above is the transitive closure \( TC \) of the binary relation \( E \).

The situation, however, changes dramatically when arbitrary DATALOG programs are considered. In fact, as mentioned in the Introduction, if \( \pi \) is a DATALOG program and \( D \) is a database, then \( (\pi, D) \) may have no fixpoints, or a unique fixpoint (which is also the least fixpoint), or even several different fixpoints.

These possibilities can be illustrated using the DATALOG program \( r_{c_1} \),

\[
T(x) \leftarrow E(y, x), \neg T(y).
\]

Indeed, let \( L_n \) be the directed path of length \( n \) with vertices \( \{1, 2, \ldots, n\} \) and edges \( E(i, i + 1) \) for \( 1 \leq i < n \), and let \( C_n \) be the directed cycle of length \( n \) with vertices \( \{1, 2, \ldots, n\} \) and edges \( E(i, i + 1) \) for \( 1 \leq i < n \) and \( E(n, 1) \). On each path \( L_n \) the program \( \pi \) has a unique fixpoint, namely the set \( \{2, 4, \ldots, 2i, \ldots\} \). On the other hand, on the cycles \( C_n \) the program \( \pi \) has no fixpoint if \( n \) is odd and has exactly two incomparable fixpoints (namely the sets \( \{1, 3, \ldots, n-1\} \) and \( \{2, 4, \ldots, n\} \)) if \( n \) is even. It follows that if \( G_n \) is the directed graph consisting of \( n \) disjoint copies of \( C_4 \), then \( \pi \) has exactly \( 2^n \) fixpoints on \( G_n \) and these fixpoints are pairwise incomparable. Thus, on \( G_n \), the program \( \pi \) has exponentially many fixpoints (in the size of the database), but no least fixpoint.

In the next section we investigate the computational complexity of the existence of fixpoints, unique fixpoints, and least fixpoints.
3. The Complexity of Fixpoints

For any fixed DATALOG\textsuperscript{\neg} program \( \pi \), the problem of telling whether an input database \( D \) has a fixpoint is in NP: One has to guess relations of size \( n' \), where \( n \) is the size of the input relations and \( s \) is the size of the (fixed) program \( \pi \), and verify (also in time \( n' \)) that the relations guessed indeed constitute a fixpoint. We show that problems in NP and DATALOG\textsuperscript{\neg} programs are in a tight correspondence.

To achieve this we need first some preliminaries. An existential second-order formula \( \Psi \) over the vocabulary \( \sigma \) is an expression of the form \( \exists \hat{S} \phi(\hat{S}) \), where \( \hat{S} = (S_1, \ldots, S_m) \) is a sequence of relational symbols (different from those in \( \sigma \)) and \( \phi(S) \) is an arbitrary first-order formula with relational symbols among those in \( \sigma \) and \( \hat{S} \). We use the following well-known result of Fagin [Fa74], which establishes a connection between computability and second-order definability.

**Theorem.** A collection \( C \) of finite databases over the vocabulary \( \sigma \) is in NP if and only if it is definable by an existential second-order formula over \( \sigma \), i.e., if and only if there is a formula \( \exists \hat{S} \phi(\hat{S}) \) such that for any database \( D \) over \( \sigma \),

\[ D \in C \iff D \models \exists \hat{S} \phi(\hat{S}). \]

We now can state and prove the following result.

**Theorem 1.** For any NP computable collection \( C \) of finite databases over \( \sigma \) there is a DATALOG\textsuperscript{\neg} program \( \pi_C \) such that a database \( D \) is in \( C \) if and only if \( (\pi_C, D) \) has a fixpoint.

**Proof.** By Fagin's theorem there is an existential second-order formula \( \Psi \) that defines the collection \( C \) on finite databases. It is known that every existential second-order formula is equivalent to one of the form

\[(\exists \hat{S})(\forall \bar{x})(\exists \bar{y})(\theta_1(\bar{x}, \bar{y}) \lor \cdots \lor \theta_k(\bar{x}, \bar{y})) \]

where \( \theta_1, \ldots, \theta_k \) are conjunctions of literals involving the relational variables in \( \sigma \) and \( \hat{S} \). This is called the Skolem normal form for existential second-order formulas. It is established by first bringing the first-order part of \( \Psi \) in prenex normal form and then applying repeatedly the equivalence

\[ (\forall \bar{u})(\exists \bar{v}) \chi(\bar{u}, \bar{v}) \leftrightarrow (\exists X)\{(\forall \bar{u})(\forall \bar{v})[X(\bar{u}, \bar{v}) \rightarrow \chi(\bar{u}, \bar{v})] \land (\forall \bar{u})(\exists \bar{v})X(u, v)] \}\]

In effect, this transformation "Skolemizes" the first-order part of \( \Psi \). The only difference from ordinary "Skolemization" is that here we do not introduce function symbols, but instead we encode functions by their graphs.

In constructing the program \( \pi_C \) we will use the DATALOG\textsuperscript{\neg} rule \( T(z) \leftarrow \neg T(w) \). This rule makes \( T \) "toggle" and in particular it has no fixpoint,
because it puts every constant in \( T \) if and only if there is a constant that is not in \( T \). It follows that the DATALOG' program

\[
T(z) \leftarrow \neg Q(u), \neg T(w)
\]

has the property that it has \( T = \emptyset \) as its unique fixpoint if and only if the complement of \( Q \) is empty. The desired program \( \pi_C \) consists of the following rules:

\[
S_j(w_j) \leftarrow S_j(w_j) \quad (1 \leq j \leq m)
\]

\[
Q(x) \leftarrow \theta_i(x, y) \quad (1 \leq i \leq k)
\]

\[
T(z) \leftarrow \neg Q(u), \neg T(w).
\]

The sole purpose of the first \( m \) rules is to make the relational symbols of \( \bar{S} \) into nondatabase relations. The effect of the next \( k \) rules is that

\[
Q \neq A^n \Leftrightarrow \neg (\forall \bar{x})(\exists \bar{y})(\theta_1(\bar{x}, \bar{y}) \lor \cdots \lor \theta_k(\bar{x}, \bar{y}))
\]

where \( n \) is the arity of the sequence \( \bar{x} \) and \( A \) is the universe of the database \( D \) under consideration.

We now claim that, for any finite database \( D \) over the vocabulary \( \sigma \), the database \( D \) is in \( C \) if and only if \( (\pi_C, D) \) has a fixpoint. Indeed, assume first that a finite database \( D \) is in \( C \). Then, by Fagin's theorem, there are relations \( \bar{S} = (S_1, \ldots, S_m) \) such that

\[
(D, \bar{S}) \models (\forall \bar{x})(\exists \bar{y})(\theta_1(\bar{x}, \bar{y}) \lor \cdots \lor \theta_k(\bar{x}, \bar{y})).
\]

It is now easy to verify that the sequence of relations

\[
S_1, \ldots, S_m, Q = A^n, \ T = \emptyset
\]

constitutes a fixpoint of \( (\pi_C, D) \). In the other direction, assume that \( (S_1, \ldots, S_m, Q, T) \) is a fixpoint of \( (\pi_C, D) \). Then \( Q \) must be equal to \( A^n \) or else \( T \) would not be a fixpoint of the last rule. It follows that

\[
(D, \bar{S}) \models (\forall \bar{x})(\exists \bar{y})(\theta_1(\bar{x}, \bar{y}) \lor \cdots \lor \theta_k(\bar{x}, \bar{y}))
\]

and hence, again by Fagin's theorem, \( D \) is in \( C \).

**Example 1.** We illustrate the proof of Theorem 1 by finding the DATALOG' program associated with the SATISFIABILITY problem.

Consider a vocabulary \( \sigma \) consisting of a unary relation symbol \( V \) and two binary relation symbols \( P \) and \( N \). Let \( \mathcal{S} \) be the class of all finite databases \( D = (A, V, P, N) \) over the vocabulary \( \sigma \) such that \( V \subseteq A, \ P \subseteq (A - V) \times V \) and \( N \subseteq (A - V) \times V \). It is easy to see that there is a one-to-one onto correspondence between databases in \( \mathcal{S} \) and instances of SATISFIABILITY. Indeed, with every instance \( I \) of SATISFIABILITY we associate a database \( D(I) \) in \( \mathcal{S} \) by taking the universe \( A \) of
$D(I)$ to be the union $V \cup C$ of the set of variables and the set of clauses of $I$, taking $V$ to be the set of variables of $I$, and using the binary relations $P$ and $N$ to encode the positive and the negative occurrences of the variables in the clauses. Thus, $P(c, v)$ holds on $D(I)$ if and only if the variable $v$ occurs positively in the clause $c$ (and analogously for $N$). Conversely, every database $D = (A, V, P, N)$ in $\mathcal{S}$ gives rise to a unique instance $I(D)$ of SATISFIABILITY with variables $V$, clauses $A \rightarrow V$, and such that a variable $v$ occurs positively (negatively) in a clause $c$ if and only if $P(c, v)$ holds on $D$ (resp., $N(c, v)$ holds on $D$).

Consider now the following existential second-order formula $\Psi$ over the vocabulary $\sigma$:

\[
(\exists S)(\forall x)(\exists y)[(S(x) \rightarrow V(x)) \land \neg V(x) \rightarrow (P(x, y) \land S(y)) \\
\lor (N(x, y) \land \neg S(y))].
\]

It is clear that for every instance $I$ of SATISFIABILITY there is a satisfying assignment for $I$ if and only if $D(I) \models \Psi$. Similarly, for every database $D$ in $\mathcal{S}$ we have that $D \models \Psi$ if and only if $I(D)$ has a satisfying assignment. Moreover, every relation $S$ on $D(I)$ (or on $D$) witnessing the existential second-order quantifier in $\Psi$ encodes a satisfying assignment of $I$ (resp., a satisfying assignment of $I(D)$).

In order to find the DATALOG $^-$ program $\pi$ associated with $\Psi$, we need to put the quantifier part of $\Psi$ in disjunctive normal form. By distributing conjunctions over disjunctions and removing redundant disjuncts, we see that $\Psi$ is equivalent to the following existential second-order formula:

\[
(\exists S)(\forall x)(\exists y)[V(x) \lor [\neg S(x) \land P(x, y) \land S(y)] \\
\lor [\neg S(x) \land N(x, y) \land \neg S(y)]].
\]

From the proof of Theorem 1, we extract now the DATALOG $^-$ program $\pi_\text{SAT}$ below.

\[
\begin{align*}
S(x) & \leftarrow S(x) \\
Q(x) & \leftarrow V(x) \\
Q(x) & \leftarrow \neg S(x), P(x, y), S(y) \\
Q(x) & \leftarrow \neg S(x), N(x, y), \neg S(y) \\
T(z) & \leftarrow \neg Q(u), \neg T(w).
\end{align*}
\]

This program has the property that for every instance $I$ of SATISFIABILITY a satisfying assignment exists for $I$ if and only if $(\pi_\text{SAT}, D(I))$ has a fixpoint.

We examine next the logical and computational complexity of determining whether or not a DATALOG $^-$ program has a unique fixpoint.

Let $\text{US}$ (unique solution) be the class of languages accepted by nondeterministic polynomial-time bounded Turing machines with the convention that a string is accepted if there is exactly one accepting computation (cf. [BG82]). It is known that $\text{US}$ contains every co-NP problem and is in turn contained in the class $\text{D}^p$. 
NEGATION BY FIXPOINT

introduced by Papadimitriou and Yannakakis [PY82], of languages that are a conjunction of NP and co-NP problems.

UNIQUE SATISFIABILITY is the prototypical problem in US. Given a set of variables and a set of clause, the question is: does the set of clauses have a unique satisfying assignment? This problem is also complete for US via polynomial time reductions. The collection of graphs having a unique Hamilton circuit is another typical member of US.

In general, if a collection $C$ of databases is defined using an existential second-order formula with unique witnesses for the second-order quantifiers, then $C$ is in US. More formally, assume that $\sigma$ is a vocabulary, $\varphi(\vec{S})$ is a first-order formula with relational symbols among those in $\sigma$ and $\vec{S}$, and $C$ is a collection of finite databases over $\sigma$ such that

$$D \in C \iff (\exists! \vec{S}) \varphi(\vec{S}),$$

where $(\exists! \vec{S})$ means that "there are unique relations $S_1, \ldots, S_m$." In this case, testing for membership in $C$ is a problem in US. We should point out, however, that not every problem in US can be written in the above logical form (cf. [Ko90]).

If $\pi$ is a DATALOG program, then $\pi$-UNIQUE FIXPOINT is the following problem: Given a database $D$, does $(\pi, D)$ have a unique fixpoint? It is easy to verify directly that this problem is in US. Moreover, it is not hard to show that $\pi$-UNIQUE FIXPOINT can be expressed in terms of an existential second-order formula with unique witnesses for the second-order quantifiers. Indeed, let $\vec{S} = (S_1, \ldots, S_m)$ be the sequence of the nondatabase relations of $\pi$. Notice that the operator $\Theta$ associated with $\pi$ can be defined using first-order formulas. Actually, the analysis of $\Theta$ given in Section 2 shows that $\Theta$ is definable using existential first-order formulas. In particular, there are existential first-order formulas $\varphi_i(\vec{x}_i, \vec{S})$, $1 \leq i \leq m$, such that for any database $D$ and any sequence of relations $\vec{S}$ on $D$ we have that

$$\Theta(\vec{S}) = (\{a_1 : D \models \varphi_1(\vec{a}, \vec{S})\}, \ldots, \{a_m : D \models \varphi_m(\vec{a}, \vec{S})\}).$$

Let $\varphi_\pi(\vec{S})$ be the first-order formula

$$\varphi_\pi(\vec{S}) \equiv \bigwedge_{i=1}^m (\forall \vec{x}_i)[S_i(\vec{x}_i) \iff \varphi_i(\vec{x}_i, \vec{S})].$$

This formula has the property that $\vec{S}$ is a fixpoint of $(\pi, D) \iff D \models \varphi_\pi(\vec{S}),$ for any database $D$ over $\sigma$ and any sequence $\vec{S}$ of relations on $D$. It follows that $\pi$-UNIQUE FIXPOINT is definable by the formula $(\exists! \vec{S}) \varphi_\pi(\vec{S})$, since

$(\pi, D)$ has a unique fixpoint $\iff D \models (\exists! \vec{S}) \varphi_\pi(\vec{S}).$

A close examination of the proof of Theorem 2 reveals that $\pi$-UNIQUE FIXPOINT is actually a normal form for collections of databases that are definable
by existential second-order formulas with unique witnesses for the second-order quantifiers. More formally, we can show that if a collection $C$ of databases is definable by an expression of the form $(3 ! S) \varphi(S)$, then there is a DATALOG' program $\pi$ such that for every database $D$ we have that $D$ is in $C$ if and only if $(\pi, D)$ has a unique fixpoint.

The preceding analysis pinpoints the logical complexity of the existence of unique fixpoints. Turning now to the computational complexity of this problem, we saw above that $\pi$-UNIQUE FIXPOINT is a problem in US. Our next result yields a matching lower bound.

**Theorem 2.** There is a DATALOG' program $\pi$ such that $\pi$-UNIQUE FIXPOINT is a US-complete problem.

**Proof.** Let $\pi$ be the DATALOG' program $\pi_{SAT}$ associated with SATISFIABILITY in the preceding Example 1. An analysis of Example 1 and of the proof of Theorem 1 shows that if $I$ is an instance of SATISFIABILITY, then there is a one-to-one correspondence between satisfying assignments for $I$ and fixpoints of $(\pi_{SAT}, D(I))$. It follows that $I$ has a unique satisfying assignment if and only if $(\pi_{SAT}, D(I))$ has a unique fixpoint.

Uniqueness of fixpoint is only one case in which fixpoint semantics of DATALOG' programs becomes deterministic. A different and more relaxed condition is that the program and data have a least fixpoint. We are thus led to ask, what is the complexity of determining for a fixed program, whether given data have a least fixpoint? The proofs of Theorems 1 and 2 actually show that there are fixed programs for which deciding whether the given data has a least fixpoint is a US-hard problem. As for upper bounds, the obvious definition of least fixpoint shows that for any fixed DATALOG' program the problem whether a database has a least fixpoint is in $\Sigma^p_2$, i.e., in the second level of the polynomial time hierarchy. With a little extra work it can be proved that the least fixpoint problem is in the class $\Sigma^p_2$ (also denoted by $\text{P}^\text{NP}$) of polynomial time computations relative to NP oracles.

In fact, the least fixpoint problem belongs to a finer complexity class that does not seem to have been isolated before in complexity theory. We say that a collection $C$ of finite databases over $\sigma$ is in $\text{FO}^\text{NP}$ (first-order with NP oracles) if it is definable by a first-order formula involving NP predicates. In view of Fagin's theorem, $\text{FO}^\text{NP}$ can be also defined to be the closure of existential second-order formulas under negation, disjunction, conjunction, and first-order quantification. In other words, $\text{FO}^\text{NP}$ can be described succintly as the first-order closure of NP. Thus, $\text{FO}^\text{NP}$ contains not only the class $\text{D}^\text{p}$, but also the entire Boolean hierarchy BH, the Boolean closure of NP, studied recently by Wechsung [We85], Cai and Hemachandra [CaH86], Kadin [Ka87], and others. Also, the class $\text{FO}^\text{NP}$ itself is contained in $\Delta^p_2$. We conjecture that the inclusions

$$BH \subseteq \text{FO}^\text{NP} \subseteq \Delta^p_2,$$
are proper, but this appears to be yet another "hard" complexity question. The problem: "Given a graph $G = (V, E)$, is there an edge $E(x, y)$ such that if this edge is removed, then the resulting graph is 3-colorable, but not Hamiltonian?" is an example of a problem in $\text{FO}^{\text{NP}}$ that does not seem to be in $D^p$, or even in some higher level of the Boolean hierarchy BH. The least fixpoint problem provides us with another such example, according to the following

**Theorem 3.** If $\pi$ is a DATALOG$^-$ program and $C_\pi$ is the collection of finite databases $D$ such that $(\pi, D)$ has a least fixpoint, then $C_\pi$ is in the class $\text{FO}^{\text{NP}}$.

**Proof.** Assume that $\bar{S} = (S_1, \ldots, S_m)$ are the nondatabase relations that occur in the DATALOG$^-$ program $\pi$. Let $\varphi_\pi(\bar{S})$ be a first-order formula with relation symbols among those in $\sigma$ and $\bar{S}$ and such that for any database $D$ over $\sigma$ and any sequence $\bar{S}$ of relations on $D$,

$$\bar{S}$$

is a fixpoint of $(\pi, D) \iff D \models \varphi_\pi(\bar{S})$.

Observe now that, given a database $D$, the program $(\pi, D)$ has a least fixpoint if and only if the (coordinatewise) intersection of all fixpoints is a fixpoint. Let $\varphi^*_\pi$ be the formula obtained from $\varphi_\pi$ as follows: For every relational symbol $S_j$, $1 \leq j \leq m$, replace each positive occurrence $S_j(\bar{w}_j)$ by the expression

$$(\forall \bar{S}^*)(\varphi_\pi(\bar{S}^*) \rightarrow \bar{S}^*_j(\bar{w}_j))$$

(this says that $\bar{w}_j$ is in the intersection of all fixpoints) and also replace each negative occurrence $\neg S_j(\bar{w}_j)$ by the negation

$$(\exists \bar{S}^*)(\varphi_\pi(\bar{S}^*) \land \neg \bar{S}^*_j(\bar{w}_j))$$

of the above expression. Notice that $\varphi^*_\pi$ is obtained from a first-order formula by substituting existential second-order and universal second-order formulas (i.e., NP and co-NP predicates) for some of its relation variables. Since $\varphi^*_\pi$ defines the collection $C_\pi$ of finite databases on which $\pi$ has a least fixpoint, it follows that $C_\pi$ is in $\text{FO}^{\text{NP}}$.

We have, thus, established lower and upper bounds for the complexity of the least fixpoint problem for DATALOG$^-$ programs. The exact complexity, however, of this problem remains an interesting open question.

Finally, we consider the version of the problem in which both the program and the data are part of the input. For an input of size $n$, one has to guess an alleged fixpoint that is potentially of size $n^n$ (since both the cardinality of the universe and the arity of the relations could be as large as $n$). Thus, the problem is in $\text{NTIME}(n^n)$. We shall show that it is hard for $\text{NEXP} (= \bigcup_{c \geq 1} \text{NTIME}(2^{cn}))$ and thus, most probably, requires doubly exponential time. This is true even when the universe is binary, in which case the problem is in $\text{NEXP}$.

The 3-COLORING problem is the following: We are given a graph $G = (V, E)$,
and we are asked whether the set $V$ of nodes can be partitioned into subsets $R$, $B$, and $G$, such that all three sets are independent (there is no edge with both endpoints in the same set). It is well known that 3-COLORING is NP-complete [Kar72]. The following DATALOG program $\pi_{\text{COL}}$ relates to the 3-colorability of the graph represented by the database relation $E$:

$$
R(x) \leftarrow R(x)
$$

$$
B(x) \leftarrow B(x)
$$

$$
G(x) \leftarrow G(x)
$$

$$
P(x) \leftarrow E(x, y), R(x), R(y)
$$

$$
P(x) \leftarrow E(x, y), B(x), B(y)
$$

$$
P(x) \leftarrow E(x, y), G(x), G(y)
$$

$$
P(x) \leftarrow G(x), B(x)
$$

$$
P(x) \leftarrow B(x), R(x)
$$

$$
P(x) \leftarrow R(x), G(x)
$$

$$
P(x) \leftarrow \neg R(x), \neg B(x), \neg G(x)
$$

$$
T(z) \leftarrow P(x), \neg T(w).
$$

This program is essentially an instantiation of the program $\pi_C$ in the proof of Theorem 1 (the negations are pushed one level inside). Thus, the sole purpose of the first three rules is to make $R$, $B$, and $G$ into nondatabase relations. The next three rules make sure that there is no edge joining two nodes both in $R$ (or $B$, or $G$) (at the penalty of making $P$ nonempty, thus triggering the last rule). The next three rules make sure that no node has two colors, and the next one that all nodes have some color. As a result, we have:

**Lemma 1.** Program $\pi_{\text{COL}}$ has a fixpoint on $E$ if and only if $E$ represents a 3-colorable graph.

With any graph-theoretic problem, we can define its "succinct version." Imagine that the nodes of the graph are the elements of $\{0, 1\}^n$, and, instead of an explicitly given edge relation, there is a Boolean circuit with $2n$ inputs and one output such that, the value output by the circuit is 1 if and only if the inputs form two $n$-tuples that are connected by an edge. A Boolean circuit is, of course, a finite set of triples $\{(a_i, b_i, c_i): i = 1, \ldots, k\}$, where $a_i \in \{\text{OR, AND, NOT, IN}\}$ is the kind of the gate, and $b_i, c_i < i$ are the inputs of the gate, unless the gate is an input gate ($a_i = \text{IN}$), in which case, say, $b_i = c_i = 0$. For NOT gates, $b_i = c_i$. Given values in $\{0, 1\}$ for the input gates, we can compute the values of all gates one by one in the obvious way. The value of the circuit is the value of the last gate. The SUCCINCT
3-COLORING problem is the following: Given a Boolean circuit with $2n$ inputs and one output, is the graph thus presented 3-colorable?

**Lemma 2.** SUCCINCT 3-COLORING is complete for NEXP.

**Proof.** It was shown in [PY86] that the succinct version of any NP-complete problem to which 3SAT is reducible by projection, is NEXP-complete. The reduction from 3SAT to 3-COLORING in [GJS76] is indeed a projection.

**Theorem 4.** The problem of determining, given a DATALOG$^-$ program and database relations with domain $\{0, 1\}$, whether it has a fixpoint, is NEXP-complete.

**Proof.** The problem is certainly solvable in NEXP, since the bound $n^n$ becomes $2^n$ in the binary case.

To prove completeness, we shall show that SUCCINCT 3-COLORING reduces to the fixpoint existence problem. Given a Boolean circuit $\{(a_i, b_i, c_i): i = 1, \ldots, k\}$, we shall construct a DATALOG$^-$ program and database relations such that the program has a fixpoint if and only if the graph represented by the given circuit is 3-colorable. Our proof uses the DATALOG$^-$ program $\pi_{\text{COL}}$ constructed above for 3-COLORING. Notice that the rules of this program make sense even in the succinct context, if the $x$ and $y$ are thought of as $n$-tuples of variables.

For each gate $g_i = (a_i, b_i, c_i)$ of the circuit, we shall have a new nondatabase relation $G_i(x, y)$, where $x$ and $y$ are $n$-tuples of variables. The intention is that $G_i(x, y)$ will contain all $2n$-tuples of bits that make $g_i$ output 1. The edge predicate $E(x, y)$ in $\pi_{\text{COL}}$ will no longer be a database relation, but the nondatabase relation $G_e(x, y)$, the output of the circuit. Thus, $E(x, y)$ will hold for exactly those $2n$-tuples of bits that correspond to adjacent nodes of the graph represented by the circuit.

There are rules which define each of the $G_i$'s. If $a_i = \text{AND}$, then we have the rule: 
“$G_i(x, y) \leftarrow G_h(x, y), G_c(x, y)$.” If $a_i = \text{OR}$, we have the rules: “$G_i(x, y) \leftarrow G_h(x, y)$” and “$G_i(x, y) \leftarrow G_c(x, y)$.” If $a_i = \text{NOT}$, then we have the rule “$G_i(x, y) \leftarrow \neg G_h(x, y)$.” Finally, if $a_i = \text{IN}$ and $g_i$ is the $j$th input of the circuit (in the order of the inputs implied by their arrangement as $x$, $y$), then we have the rule: “$G_i(z_1, \ldots, z_{j-1}, 1, z_{j+1}, \ldots, z_n) \leftarrow$.” Notice that the program has no database relations, but we have fixed the domain of all variables to be $\{0, 1\}$ (fixing the universe is not a departure from our framework, as it can be achieved by introducing a new database relation $D(\cdot)$ with value $\{0, 1\}$). Finally, we identify relation $G_e(x, y)$ with $E(x, y)$ in the program, and add the rules of program $\pi_{\text{COL}}$. We call the resulting program $\pi_{\text{SC}}$.

We claim that $\pi_{\text{SC}}$ has a fixpoint if and only if the graph represented by the circuit is 3-colorable. In any fixpoint of $\pi_{\text{SC}}$, relation $G_i$ will contain precisely those $2n$-tuples that, considered as inputs of the circuit, make the value of gate $g_i$ equal to 1. Thus, relation $E$ will contain precisely those pairs of $n$-tuples that are adjacent nodes of the graph represented by the circuit. The result now follows from Lemma 1.
4. INFLATIONARY DATALOG

The results of the previous section suggest that any reasonable version of fixpoint semantics for logic programs with negation is faced with computational obstacles. In this section we study a new database query language called Inflationary DATALOG, which assigns inflationary semantics to logic programs with negation. Inflationary semantics has the advantage that it gives meaning to all DATALOG\(\neg\) programs; it is a natural extension of the standard DATALOG semantics and is computable in polynomial time.

We consider first DATALOG\(\neg\) programs \(\pi\) over the vocabulary \(\sigma\) having a single non-database relation \(S\). Let \(k\) be the arity of \(S\) and let \(D\) be any database over \(\sigma\) having universe \(A\). As described in Section 2, the program \(\pi\) gives rise to a mapping \(\Theta\) from \(k\)-ary relations on \(A\) to \(k\)-ary relations on \(A\). The inflationary semantics of the DATALOG\(\neg\) program \(\pi\) on \(D\) is defined by iterating the mapping \(\Theta\) in the following way: we define first the sequence \(Q^n\), \(n \geq 1\), of \(k\)-ary relations on \(A\) by the equations:

\[
Q^1 = \Theta(\emptyset), \quad Q^{n+1} = Q^n \cup \Theta(Q^n)
\]

and then we put

\[
\Theta^\infty = \bigcup_{n=1}^{\infty} \Theta^n.
\]

The inflationary semantics of the DATALOG\(\neg\) program \(\pi\) on the database \(D\) is the \(k\)-ary relation \(\Theta^\infty\). For DATALOG\(\neg\) programs with more than one non-database relations the inflationary semantics is defined in a similar way by simultaneous induction in the defining equations. One of the non-database relations is identified as the carrier (often also called the goal predicate) of the program and \(\Theta^\infty\) in this case denotes the relation corresponding to the carrier at the end of the iteration.

Several remarks are in order. Notice first that if \(\pi\) is a DATALOG program (without negations), then \(\Theta^{n+1} = \Theta(\Theta^n)\), because in this case \(\Theta\) is a monotone mapping. It follows that for DATALOG programs the relation \(\Theta^\infty\) is the least fixpoint of \((\pi, D)\) and therefore in this case the inflationary semantics coincides with the standard DATALOG semantics. For general DATALOG\(\neg\) programs, however, the relation \(\Theta^\infty\) need not be the least fixpoint of \((\pi, D)\) or even a fixpoint of it, since \((\pi, D)\) may have no fixpoint whatsoever. Notice next that the sequence of relations \(\Theta^n, n \geq 1\), is increasing. This in turn implies that on any finite database \(D\) there is a number \(n_0 \leq |A|^k\) (where \(|A|\) is the cardinality of the universe \(A\) of \(D\)) such that

\[
\Theta^\infty = \Theta^{n_0} = \Theta^n
\]

for every \(n \geq n_0\). As a result, the relation \(\Theta^\infty\) is computable in polynomial time (in the size of the database \(D\)) and consequently the inflationary semantics is, at least in principle, efficiently implementable.
Let us examine the inflationary semantics of some DATALOG\(^{-}\) programs we encountered earlier. For the program \(T(x) \leftarrow \neg T(y)\) we have that \(\Theta^\infty = \Theta^1 = \mathcal{A}\), for any database \(D\) with universe \(\mathcal{A}\). Similarly, if \(\pi_1\) is the program

\[
T(x) \leftarrow E(y, x), \neg T(y)
\]

studied in Section 2 and \(G = (\mathcal{V}, \mathcal{E})\) is any graph, then \(\Theta^\infty = \Theta^1 = \{x \colon \exists y \in \mathcal{E}(y, x)^\}\). The relations computed by these two programs are obviously first-order. Later on we will see examples of DATALOG\(^{-}\) programs whose inflationary semantics are relations that are not first-order definable. In such cases there is no uniform bound on the number of iterations after which the sequence \(\Theta^n\) becomes constant on finite databases.

We should point out that neither the term *inflationary* nor this particular concept of iterating mappings between relations is new. The term *inflationary operator* was coined by Gurevich and Shelah [GS86], where an operator \(H\) (a mapping) from \(k\)-ary relations to \(k\)-ary relations is said to be inflationary if \(S \subseteq H(S)\) for every \(k\)-ary relation \(S\). For any operator \(H\), let

\[
H' = H(\emptyset) \quad \text{and} \quad H^{n+1} = H(H^n).
\]

Notice that an inflationary operator \(H\) has at least one fixpoint on every database \(D\), since \(\mathcal{A}^k\) is a fixpoint of \(H\), where \(\mathcal{A}\) is the universe of \(D\). On the other hand, exactly one of the fixpoints of \(H\) is of the form \(H^n\) for some \(n \geq 1\). This is called the *inductive fixpoint* of \(H\). Observe also that if \(H\) is a monotone operator, then the least fixpoint of \(H\) coincides with the inductive fixed point of \(H\). If \(F\) is a mapping from \(k\)-ary relations to \(k\)-ary relations, then the operator \(\hat{F}(S) = S \cup F(S)\) is inflationary; as a result, it has an inductive fixpoint, which, by an abuse of terminology, is sometimes also called the *inductive fixpoint* of \(F\) or more often (and more accurately) the relation *inductively definable* by \(F\). What is going on here is that the relation \(\Theta^\infty\) associated with a DATALOG\(^{-}\) program \(\pi\) is the inductive fixpoint of the inflationary operator

\[
\hat{\Theta}(S) = S \cup \Theta(S).
\]

Gurevich and Shelah [GS86] studied the expressive power of the logic FO + IFP (for first-order + inductive fixpoint) on finite structures. FO + IFP is first-order logic augmented with the inductive fixpoint formation rule for inflationary operators \(\hat{F}\) that are obtained from first-order definable operators \(F\). An operator \(F\) between \(k\)-ary relations is said to be first-order definable if there is a first-order formula \(\varphi(x_1, ..., x_k, S)\) such that on any finite database \(D\),

\[
F(S) = \{(a_1, ..., a_k) : D \models \varphi(a_1, ..., a_k, S)\}.
\]

Long before that, however, inductive fixpoints and inflationary operators had been studied on infinite structures under the name nonmonotone inductive definability. There is an extensive literature on this subject that goes back to the 50's. A nice
summary of the main results in this area can be found in Section 3.5 of Aczel [Ac77]. We see, therefore, that inflationary semantics is a natural concept that has been explored with much success in other contexts. We feel that it also deserves consideration and study in logic programming, as an alternative semantics for negation.

Let \( \pi \) be a DATALOG\( ^\triangledown \) program whose carrier is a \( k \)-ary nondatabase relation \( S \). Then the inflationary semantics of \( \pi \) gives rise to a query, i.e., a mapping assigning to every finite database \( D \) the \( k \)-ary relation \( \Theta^\infty \). It turns out that the collection of queries definable by DATALOG\( ^\triangledown \) programs under the inflationary semantics corresponds to the existential fragment of FO + IFP according to the following result.

**Proposition 1.** A query is expressible in Inflationary DATALOG if and only if it is expressible in FO + IFP using operators definable by existential first-order formulas.

**Sketch of Proof.** In Section 2 we showed that if \( \pi \) is a DATALOG\( ^\triangledown \) program, then the operator \( \Theta \) associated with \( \pi \) is definable using an existential first-order formula. From this it follows easily that if a query is expressible in Inflationary DATALOG, then it is in the existential fragment of FO + IFP. For the other direction, assume that \( H \) is an operator definable by an existential first-order formula \( \varphi \). It is easy to show now that \( H \) can be simulated by a DATALOG\( ^\triangledown \) program \( \pi \), which is obtained by bringing the existential formula \( \varphi \) in disjunctive normal form and associating a DATALOG\( ^\triangledown \) rule with every disjunct of \( \varphi \).

We should point out that Chandra and Harel [CH85] obtained a similar result for DATALOG programs and positive existential formulas.

We show next that Inflationary DATALOG has higher expressive power than DATALOG (without negation). We will exhibit, in particular, a natural query on finite graphs which is neither first-order definable nor expressible by a DATALOG program, but which is expressible by a DATALOG\( ^\triangledown \) program under inflationary semantics.

The transitive closure \( \text{TC}(x, y) \) query, namely "is there a path from \( x \) to \( y \)?," is the canonical example of a natural query on finite graphs \( G = (V, E) \) which is not first-order definable (cf. [AU79]), but which is expressible as the least fixpoint of the DATALOG program:

\[
S(x, y) \leftarrow E(x, y) \\
S(x, y) \leftarrow E(x, z), S(z, y).
\]

Closely related to this is the distance query \( D(x, y, x^*, y^*) \), namely "is there a path from \( x \) to \( y \) that is shorter than or equal to any path from \( x^* \) to \( y^* \)?" (it is understood that the answer to this query is "yes" if there is a path from \( x \) to \( y \), but no path from \( x^* \) to \( y^* \)).
Proposition 2. There is an Inflationary DATALOG program whose carrier expresses the distance query on finite graphs. There is no first-order formula or DATALOG program expressing the distance query on finite graphs.

Proof. The transitive closure query is reducible to the distance query, since \( \text{TC}(x, y) \) if and only if \( D(x, y, x, y) \). From this it follows immediately that the Distance query is not first-order definable. Notice that the distance query is not monotone, in the sense that if in a graph \( G = (V, E) \) we have that \( D(a, b, a^*, b^*) \) and \( G' = (V, E') \) with \( E \subseteq E' \), then it is not necessarily true that \( D(a, b, a^*, b^*) \) holds in \( G' \). Since DATALOG programs give rise to monotone queries only, we conclude that the distance query is not expressible by a DATALOG program.

Consider now the following Inflationary DATALOG program \( r_c \) with carrier \( S_3 \):

\[
S_1(x, y) \leftarrow E(x, y) \\
S_1(x, y) \leftarrow E(x, z), S_1(z, y) \\
S_2(x^*, y^*) \leftarrow E(x^*, y^*) \\
S_2(x^*, y^*) \leftarrow E(x^*, z^*), S_2(z^*, y^*) \\
S_3(x, y, x^*, y^*) \leftarrow E(x, y), S_1(z, z), S_1(z, y), \neg S_2(x^*, y^*) \\
S_3(x, y, x^*, y^*) \leftarrow E(x, z), S_1(z, y), \neg S_2(x^*, y^*).
\]

The first four rules generate two synchronous copies of the transitive closure. During the iteration, the last two rules assign to the carrier \( S_3 \) quadruples such that the first pair enters in the first copy of the transitive closure by the next level of the iteration, while the second pair is not in the second copy of the transitive closure built so far. As a result, in the \( n \)th level of iteration, the new quadruples \((x, y, x^*, y^*)\) entering the carrier \( S_3 \) are the ones for which the shortest path from \( x \) to \( y \) is of length exactly \( n \) and the shortest path from \( x^* \) to \( y^* \) is of length at least \( n \). It follows that, under the inflationary semantics, the carrier \( S_3 \) of \( \pi \) computes the distance query on finite graphs.

It is perhaps interesting to note that \( \pi \) is also a stratified logic program, since there is no "recursion through negation" in it (cf. [CH85, VG86, or ABW86] for the exact definitions). More precisely, \( \pi \) can be viewed as a stratified logic program with two strata, where the lower stratum consists of the first four rules and the higher one of the last two rules of \( \pi \). Under the semantics of stratified logic programs, each stratum is computed separately, proceeding in order from the lower to the higher. Thus, if the above program \( \pi \) is viewed as a stratified logic program, then it does not compute the distance query. Instead, it computes the query

\[
\{(x, y, x^*, y^*) : \text{TC}(x, y) \land \neg \text{TC}(x^*, y^*)\}.
\]

In particular, we see that inflationary semantics differs from the semantics of stratified logic programs. It is an open problem to determine whether or not there is a stratified logic program computing the distance query (cf. also [Ko89]).
5. Concluding Remarks

We have demonstrated here that any treatment of negation using fixpoint semantics is impeded by complexity-theoretic obstacles. As a remedy, we have proposed a new database query language, Inflationary DATALOG. We should point out that Abiteboul and Vianu [AV88] have independently introduced a language called SdetDL, identical in expressive power to Inflationary DATALOG.

What is the exact expressive power of Inflationary DATALOG (or SdetDL)? The preceding Propositions 1 and 2 imply that the queries expressible in Inflationary DATALOG on finite databases properly contain the DATALOG queries and are in turn contained in the queries expressible in the existential fragment of FO + IFP.

Abiteboul and Vianu [AV88] obtained two interesting results that provide a precise characterization of the expressive power of Inflationary DATALOG. More specifically, they established first that on finite databases the queries expressible in Inflationary DATALOG are closed under complement. As a consequence, on finite databases Inflationary DATALOG contains all queries expressible by stratified logic programs. In addition, [AV88] showed that on finite databases Inflationary DATALOG has the same expressive power as fixpoint logic FP. Fixpoint logic is a natural extension of first-order logic obtained by augmenting first-order logic with the least fixpoint formation rule for operators definable by positive first-order formulas. It has been studied in depth on both infinite structures [Mo74] and finite databases [CH82, Va82, Im86, GS86]. In particular, the main theorem in Gurevich and Shelah [GS86] is that on finite databases FO + IFP coincides with fixpoint logic FP in terms of expressive power.

In addition, [Ko89] obtained a separation between stratified logic programs and fixpoint logic on finite databases, building on work of [Da87]. Thus, by combining all the results described above, we arrive at the following picture on finite databases (\(\subset\) denotes proper inclusion in terms of expressive power):

\[
\text{DATALOG} \subset \text{Stratified Logic Programs} \\
\subset \text{Inflationary DATALOG} = \text{FP} = \text{FO} + \text{IFP}.
\]

Moreover,

\[
\text{Relational Calculus} \subset \text{Stratified Logic Programs},
\]

but

\[
\text{Relational Calculus} \not\subset \text{DATALOG}.
\]

We conclude by pointing out that the above picture is rather special to finite structures, since the results of [GS86] and [AV88] do not hold in general on infinite structures.
Many thanks to Stavros Cosmadakis, Alex Schaffer, Larry Stockmeyer, Allen Van Gelder, and Moshe Vardi for several useful suggestions and comments on an earlier draft of this paper.

REFERENCES


