Information and Computation **147**, 22–56 (1998)
Article No. IC982734

The Expressive Power of Stratified Logic

View metadata, citation and similar papers at core.ac.uk

Luca Cabibbo

Dipartimento di Informatica e Automazione, Università degli Studi di Roma Tre, Via della Vasca Navale 79, I-00146 Rome, Italy
E-mail: cabibbo@dia.uniroma3.it

The expressive power of the family wILOG$^{(\neg)}$ of relational query languages is investigated. The languages are rule based, with value invention and stratified negation. The semantics for value invention is based on Skolem functor terms. We study a hierarchy of languages based on the number of strata allowed in programs. We first show that, in presence of value invention, the class of stratified programs made of two strata has the expressive power of the whole family, thus expressing the computable queries. We then show that the language wILOG$^{\neq}$ of programs with nonequality and without negation expresses the monotone computable queries, and that the language wILOG$^{1/2,\neg}$ of semipositive programs expresses the semimonotone computable queries.
© 1998 Academic Press

1. INTRODUCTION

The study of query languages is a major issue in database theory. Beginning with the relational calculus and algebra query languages for the relational model of data (Codd, 1970), several extensions to both the languages and the model have been investigated, mainly with the goal of gaining in expressive power. A theory of queries originated from the definition of *computable queries* (Chandra and Harel, 1980) as a "reasonable" class of mappings from databases to databases. Computable queries are the class of partial recursive functions over finite relational structures that satisfy a criterion called *genericity*. The notion of genericity formalizes the *data independence principle* in databases; intuitively, it captures the fact that the only significant relationships among data are based on (non)equality of values. Genericity is a generalization of interesting properties enjoyed by the queries

* Part of the results in this paper appeared under the title "On the power of stratified logic programs with value invention for expressing database transformations" in *International Conference on Database Theory*, 1995. This work was partially supported by *MURST*, within the Project "Metodi formali e strumenti per basi di dati evoluti," and by *Consiglio Nazionale delle Ricerche*, within "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, Obiettivo LOGIDATA+."

0890-5401/98 $25.00
Copyright © 1998 by Academic Press
All rights of reproduction in any form reserved.

that are expressible in relational algebra and calculus (Aho and Ullman, 1979; Bancilhon, 1978; Paredaens, 1978).

Relational calculus and algebra express LOGSPACE queries only, and the addition of an iterative construct (a fixpoint operator or a *while* iterator) does not lead beyond PSPACE queries (Immerman, 1986; Vardi, 1982). In fact, queries in such languages may use only relations having fixed scheme and constants from the input database, so that their working space is polynomial in the size of the active domain of the input instance. Thus, a further mechanism is needed to fulfill completeness. There are (at least) two ways to overcome the PSPACE barrier: allowing for relations having variable scheme, or for the use of constants outside the active domain of the input database; several proposals in the literature consider indeed one of these approaches toward completeness. The former was pursued in (Chandra and Harel, 1980) with the language *QL*, introducing a modification to the data model to allow for unranked relations—intuitively, to simulate unbounded space on a Turing machine tape. The latter approach has been proposed in (Abiteboul and Vianu, 1990; Abiteboul and Vianu, 1991) using a mechanism of *value invention* as a means to introduce new domain elements in temporary relations during computations. They embedded value invention both in a procedural language (Abiteboul and Vianu, 1990) and in a rule-based one (Abiteboul and Vianu, 1991). It is worth noting that value invention does not increase the expressive power of relational calculus (Hull and Su, 1994); it is possible to achieve a complete language by extending the relational calculus with both value invention and an iterative construct.

A different mechanism to achieve completeness was proposed in (Hull and Su, 1989; Hull and Su, 1993), extending the data model with complex objects—built using the *set* and *tuple* constructors—to allow for *recursive types*. Unbounded value structures can thus be defined, which correspond essentially to hereditarily finite sets. The connection between hereditarily finite set construction and value invention was shown in (Van den Bussche *et al*., 1997), which reconciled the two approaches.

The idea of value invention originates from a proposal by Kuper and Vardi to choose arbitrary symbolic *object names* to manage new complex object values defined in their logical queries (Kuper and Vardi, 1984; Kuper and Vardi, 1993). The concept of object name is a refinement of Codd's notion of *surrogate* (Codd, 1979); nowadays, we use the term *object identity*. The mechanism of value invention has been recast into an object-oriented data model within a traditional database framework in the language IQL (Abiteboul and Kanellakis, 1989). There, value invention is more properly called object creation, because invented values are used to assign new object identifiers in correspondence to newly created objects. Object creation has been also incorporated in the rule-based language ILOG (Hull and Yoshikawa, 1990); ILOG adopts a different (and more declarative) semantics for object creation, using Skolem functor terms as suggested in previous proposals (Maier, 1986; Chen and Warren, 1989; Kifer *et al*., 1995; Kifer and Wu, 1993).

In this paper we study the expressive power of a family of query languages with value invention in the context of relational databases. The languages are rule based, extending the syntax and semantics of *Datalog*. The semantics of value invention is based on Skolem functors. Stratified negation is allowed in programs. We adopt the

formalism of ILOG$^{\neg}$ (Hull and Yoshikawa, 1990), which enjoys all the above charateristics. The language ILOG$^{\neg}$, originally proposed to express queries in the context of object databases, can be syntactically limited to specify relational queries only, that is, generic database mappings. (We do so by requiring *weak safety* in the use of value invention, that is, by allowing invented values in temporary relations only.) The language obtained in this way, called wILOG$^{\neg}$, expresses the computable queries of Chandra and Harel—this fact can be formally proved as a consequence of previous results (Hull and Su, 1989; Hull and Su, 1997). We strengthen this result, showing that the same expressive power can be achieved by means of a syntactically simpler language, obtained by limiting the use of negation to two strata programs, that is, programs made of a positive stratum followed by a semipositive one.

Starting from this first completeness result, we investigate languages with even more limited use of negation. We show that the language wILOG$^{\neq}$, in which the only form of negation allowed is nonequality, expresses the monotone queries, that is, all the computable queries that satisfy the monotonicity property. We then study the language wILOG$^{1/2,\neg}$ of semipositive programs, in which negation can be applied to input relations only. The language is shown to express the semimonotone queries, that is, the queries that satisfy a weak monotonicity property, also called "queries preserved under extensions" in the literature (Afrati *et al.*, 1991).

These results characterize the expressive power of stratified *Datalog* extended with value invention, with respect to a number of limitations in the use of negation. The languages considered express exactly some well-defined subclasses of the computable queries. In contrast, the classes of queries expressed by stratified *Datalog*$^{\neg}$, with the same limitations in the use of negation, are not always well-defined subclasses of the PTIME queries (Afrati *et al.*, 1991; Kolaitis, 1991; Kolaitis and Vardi, 1990; Papadimitriou, 1985). This highlights the profound impact that value invention has in database manipulation.

The paper is organized as follows. We recall some preliminary definitions in Section 2. Section 3 introduces the family ILOG$^{(\neg)}$ of languages, with some examples. Then, Sections 4, 5, and 6 are devoted to the study of the query languages wILOG$^{1,\neg}$, wILOG$^{\neq}$, and wILOG$^{1/2,\neg}$, which express the classes of computable queries, monotone queries, and semimonotone queries, respectively. In Section 7 we discuss why we do not have any expressiveness result concerning the language wILOG of positive programs. Concluding remarks are proposed in Section 8.

2. PRELIMINARIES

2.1. The Data Model

We assume the reader is familiar with the basic elements of *relational database theory* (Abiteboul *et al.*, 1995; Kanellakis, 1990). We now briefly review some database terminology and notation.

We assume the existence of three countably infinite and pairwise disjoint sets: the set $\mathscr{L}_r$ of *relation names*, the set $\mathscr{L}_a$ of *attribute names*, and the set Δ of *constants*, called the *domain*.

A *relation scheme* is a relation name $R \in \mathscr{L}_r$ together with a (possibly empty) finite set of attribute names in $\mathscr{L}_a$. We write $R(A_1 \cdots A_n)$ to denote a relation scheme with name R and set of attributes $\{A_1, ..., A_n\}$. The set of attribute names associated with R is denoted $sort(R)$; each $A \in sort(R)$ is called an *attribute of* R, and sometimes denoted as a pair (R, A). The *arity* of a relation R is the number $\alpha(R) = |sort(R)|$ of its attributes. A (*database*) *scheme* $\mathscr{S}$ is a finite set of relation schemes, having distinct names.

For a set X of attribute names, a *tuple* t *over* X is a total function $t: X \to \Delta$; we write $(A_1: d_1, ..., A_n: d_n)$ to denote a tuple t over $A_1 \cdots A_n$ such that $t(A_i) = d_i$, for $1 \leqslant i \leqslant n$. For a relation scheme R, a *relation instance over* R is a finite set of tuples over $sort(R)$. For a scheme $\mathscr{S}$, a (*database*) *instance* $\mathscr{I}$ *over* $\mathscr{S}$ is a function mapping each relation name R in $\mathscr{S}$ to a relation instance over R. The *active domain* of an instance $\mathscr{I}$, denoted by $adom(\mathscr{I})$, is the set of all domain elements occurring in $\mathscr{I}$. The set of all instances over a scheme $\mathscr{S}$ is denoted by $inst(\mathscr{S})$.

We now give an equivalent representation for instances, following the logic programming style. In doing so, we adopt a positional notation, omitting attribute names in tuples. Specifically, we assume the existence of a total order on $\mathscr{L}_a$ and use the convention of listing sets of attributes according to the total order. Then, if the list of attributes $A_1 \cdots A_n$ respects the total order, then we write $(d_1, ..., d_n)$ to represent a tuple $(A_1: d_1, ..., A_n: d_n)$.

Given a relation scheme R, a *fact over* R is an expression of the form $R(d_1, ..., d_n)$, where $(d_1, ..., d_n)$ is a tuple over $sort(R)$. A *relation instance over* R is a finite set of facts over R. For a scheme $\mathscr{S}$, a (*database*) *instance over* $\mathscr{S}$ is a finite set of facts over the relations in $\mathscr{S}$.

2.2. Queries and Query Languages

Given schemes $\mathscr{S}$ and $\mathscr{T}$, a *database mapping* f *from* $\mathscr{S}$ *to* $\mathscr{T}$, denoted $f: \mathscr{S} \to \mathscr{T}$, is a partial function from $inst(\mathscr{S})$ to $inst(\mathscr{T})$.

Let C be a set of constants, elements of the domain Δ. A database mapping f is *C-generic* if $f \cdot \rho = \rho \cdot f$ for each permutation ρ over Δ (extended in the natural way to instances) that is the identity on C. A database mapping is *generic* if it is C-generic for some finite C. A *query from* $\mathscr{S}$ *to* $\mathscr{T}$ is a generic database mapping $f: \mathscr{S} \to \mathscr{T}$.

The class $\mathscr{C}\mathscr{Q}$ of *computable queries* (Chandra and Harel, 1980) is the set of all queries f such that the mapping f is Turing computable.

The notion of genericity has been introduced to capture the fact that the only significant relationships among data are those based on (non)equality of values, that is, values have to be considered as *uninterpreted*, apart from a finite set C of domain elements, which may be fixed by the query. As a consequence of genericity, for a C-generic query q and an input instance $\mathscr{I}$, $adom(q(\mathscr{I})) \subseteq adom(\mathscr{I}) \cup C$. This property states that queries are essentially *domain-preserving* database mappings.

A *query language* is a formalism (i.e., a syntax together with a semantics) to formulate queries. Given a query language L, a query q is *expressible in* L if there exists an expression of L whose semantics coincides with the query q.

We can compare expressiveness of query languages, as follows. Given two languages L_1 and L_2, we say that L_1 *is weaker than* L_2, denoted $L_1 \sqsubseteq L_2$, if each query in L_1 is expressible in L_2 as well; L_1 and L_2 are *equivalent*, denoted $L_1 \equiv L_2$, if both $L_1 \sqsubseteq L_2$ and $L_2 \sqsubseteq L_1$ hold. We say that L_2 *is more expressive than* L_1, denoted $L_1 \sqsubset L_2$, if $L_1 \sqsubseteq L_2$ but not $L_1 \equiv L_2$.

We can compare query languages to classes of queries as well. Given a query language L and a class $\mathscr{C}$ of queries, we say that L *expresses* $\mathscr{C}$, denoted $L \equiv \mathscr{C}$, if any query in $\mathscr{C}$ is expressible in L.

3. THE LANGUAGE ILOG$^{(\neg)}$

In this section we briefly introduce the syntax and semantics of the language ILOG$^{(\neg)}$. The language was proposed by Hull and Yoshikawa; for a complete presentation we refer the reader to the papers (Hull and Yoshikawa, 1990; Hull and Yoshikawa, 1991). We will not consider here the object-based characteristics of the language, which motivated its introduction; instead, in this paper we focus only on the ability of the language to express queries in the relational framework.

The language is a variant of *Datalog*, with stratified negation and a mechanism for value invention, which is indicated by the use of a distinguished symbol "$*$" in atoms in heads of clauses.

3.1. Syntax

Let a database scheme $\mathscr{S}$ be fixed. Assume the existence of a further countably infinite set **Var** of *variables*, disjoint from the domain Δ.

A *term* is either a domain element $d \in \Delta$ or a variable $X \in \mathbf{Var}$. A *relation atom* is an expression of the form $R(A_1: t_1, ..., A_n: t_n)$, where R is a relation with $sort(R) = A_1 \cdots A_n$, and $t_1, ..., t_n$ are terms. An *invention atom* is an expression of the form $R(\text{ID}: *, A_1: t_1, ..., A_n: t_n)$, where R is a relation with $sort(R) = \text{ID}\, A_1 \cdots A_n$, ID is a distinguished attribute name called the *invention attribute*, "$*$" is a special symbol called the *invention symbol*, and $t_1, ..., t_n$ are terms. Intuitively, the invention symbol and invention atoms are used to introduce new domain elements throughout the computation of the model of a program. An *equality atom* is an expression of the form $t_1 = t_2$, where t_1, t_2 are terms. A *positive literal* is either a relation atom or an equality atom. A *negative literal* $\neg L$ is the negation of a positive literal L; a negative literal $\neg t_1 = t_2$ is called a *nonequality literal* and usually denoted $t_1 \neq t_2$. In the remainder of this paper we will not consider equality atoms anymore (because we can resort to multiple occurrences of the same term, instead), whereas nonequality literals will be used. A *literal* is either a positive or a negative literal.

Again, it is possible to adopt a positional notation by referring to a total order on $\mathscr{L}_a$ and omitting attribute names in literals. In particular, we assume that ID is the minimum element in $\mathscr{L}_a$ so that, if the invention attribute and symbol occur in an atom, then they occur in the first position.

A *clause* γ is an expression of the form

$$A \leftarrow L_1, ..., L_k.$$

where A is either a relation or an invention atom, called the *head of* γ and denoted *head*(γ), and $L_1, ..., L_k$ (with $k \geqslant 0$) is a (possibly empty) finite set of literals, called the *body of* γ and denoted *body*(γ). A clause is *range restricted* if each variable that occurs either in its head or in a negative literal in its body, occurs in a relation atom in its body as well. Hereinafter we will consider range-restricted clauses only. A *fact* is a clause with an empty body (that is, an atom A). A clause is an *invention* (*relation*, respectively) clause if its head is an invention (relation, respectively) atom.

A relation name occurring in the head of an invention clause is called an *invention relation*. We assume that the distinguished attribute name ID is used in invention relation schemes only.

An ILOG$^\neg$ *program* is a finite set of clauses, with the condition that no invention relation occurs in the head of a relation clause.

For a program $\mathscr{P}$, we denote by *adom*($\mathscr{P}$) the finite set of domain elements that occur explicitly in $\mathscr{P}$, and by *sch*($\mathscr{P}$) the database scheme made of the relation schemes occurring in $\mathscr{P}$. An *input–output scheme* (*or*, *simply*, *i–o scheme*) *for* $\mathscr{P}$ is a pair of schemes ($\mathscr{S}, \mathscr{T}$) such that: (i) $\mathscr{S}$ and $\mathscr{T}$ are disjoint subsets of *sch*($\mathscr{P}$), called the *input* and *output* schemes, respectively; and (ii) no relation name in $\mathscr{S}$ occurs in the head of a clause in $\mathscr{P}$. For a program $\mathscr{P}$ over an i–o scheme ($\mathscr{S}, \mathscr{T}$), denoted ($\mathscr{P}, \mathscr{S}, \mathscr{T}$), relations in the input scheme play the role of *extensional* relations, relations in the output scheme that of *intensional* (or *target*) relations, whereas relations in *sch*($\mathscr{P}$) but neither in $\mathscr{S}$ nor in $\mathscr{T}$ are viewed as *temporary* relations.

Let $\mathscr{P}$ be an ILOG$^\neg$ program. We say that $\mathscr{P}$ is *stratified* (Apt, 1990) if there exists a partition $P_1, ..., P_n$ of the clauses of $\mathscr{P}$ such that:

- if a relation name R occurs in a positive literal in the body of a clause in P_i, then each clause having R in its head is contained in $\bigcup_{j \leqslant i} P_j$; and
- if a relation name R occurs in a negative literal in the body of a clause in P_i, then each clause having R in its head is contained in $\bigcup_{j < i} P_j$.

If these conditions are satisfied, then the partition $P_1, ..., P_n$ is a *stratification* of $\mathscr{P}$, and each P_i is a *stratum*. In the remainder of the paper, unless stated explicitly, we will consider stratified ILOG$^\neg$ programs only.

A *positive* ILOG program is a program in which no negative literal occurs. An ILOG$^\neq$ program is a program in which the only negative literals allowed are non-equality literals. A *semipositive* ILOG$^{1/2, \neg}$ program is a program in which every relation name occurring in a negative literal is a relation of the input scheme.

3.2. Semantics

We now define the semantics of ILOG$^{(\neg)}$ programs; as the ordinary semantics of stratified logic programs, it is based on the notion of perfect model (minimal model for positive programs). The behavior of the symbol "$*$", used for value invention

in programs, remains to be specified; we follow the so-called *functional approach*, according to which its meaning is completely characterized resorting to Skolem functor terms. In this way, value invention in ILOG$^{\neg}$ programs corresponds essentially to a limited use of function symbols in ordinary logic programs. As a consequence, as noted in (Hull and Yoshikawa, 1990), positive programs have a monotonic semantics, which is equivalently characterized in a model-theoretic as well as a fixpoint semantics. This is in contrast with the "operational" semantics of value invention adopted in $Datalog^{\neg}_{\infty}$ (Abiteboul and Vianu, 1991), where there exist positive programs defining non-monotone queries (see (Hull and Yoshikawa, 1990, Example 7.6)).

The semantics of an ILOG$^{\neg}$ program $(\mathscr{P}, \mathscr{S}, \mathscr{T})$ is a binary relation between $inst(\mathscr{S})$ and $inst(\mathscr{T})$, which is defined here in terms of a four-step process, described informally as follows:

1. Replace the occurrences of the invention symbol "$*$" by appropriate Skolem functor terms, thus obtaining the Skolemization $Skol(\mathscr{P})$ of $\mathscr{P}$.

2. For an instance $\mathscr{I}$, consider its representation as a set of facts.

3. $Skol(\mathscr{P}) \cup \mathscr{I}$ is essentially a logic program with function symbols; a preferred model $\mathscr{M}_{Skol(\mathscr{P}) \cup \mathscr{I}}$ of $Skol(\mathscr{P}) \cup \mathscr{I}$ (minimal if $\mathscr{P}$ is either a positive ILOG or an ILOG$^{\neq}$ program, perfect if it is a stratified ILOG$^{\neg}$ program) can be found via a fixpoint computation; if $\mathscr{M}_{Skol(\mathscr{P}) \cup \mathscr{I}}$ is finite, call it the *model of* $\mathscr{P}$ *over* $\mathscr{I}$.

4. If the model of $\mathscr{P}$ over $\mathscr{I}$ is defined, it is something similar to a set of facts of the language, apart from the presence of Skolem functor terms. In order to obtain an instance of the output scheme, we must coherently replace distinct Skolem functor terms by distinct new values (that is, values that do belong neither to $adom(\mathscr{I})$ nor to $adom(\mathscr{P})$), thus obtaining an instance $\mathscr{J}$ over $sch(\mathscr{P})$. Then, the *semantics* $\mathscr{P}(\mathscr{I})$ *of* $(\mathscr{P}, \mathscr{S}, \mathscr{T})$ *over* $\mathscr{I}$ is the restriction of $\mathscr{J}$ to the relation names in $\mathscr{T}$. Otherwise (i.e., if $\mathscr{M}_{Skol(\mathscr{P}) \cup \mathscr{I}}$ is infinite) the *semantics* is *undefined.*

We now formalize concepts related to "Skolem functor terms", in order to make precise the definition of the semantics of ILOG$^{\neg}$ programs.

Assume the existence of a countable set $\mathscr{L}_f$ of *Skolem functor names.* For each different relation name $R \in \mathscr{L}_r$, the set $\mathscr{L}_f$ contains a distinct functor name f_R, called the *Skolem functor associated with R.*

Consider an invention relation $R(\text{ID}A_1 \cdots A_n)$, having f_R as the associated Skolem functor; a *Skolem functor term for R* is an expression of the form $f_R(A_1: t_1, ..., A_n: t_n)$ (or simply $f_R(t_1, ..., t_n)$, adopting a positional notation) where $t_1, ..., t_n$ are terms; note how the scheme constrains the functor f_R to have arity $n = \alpha(R) - 1$. Then, extend the notion of *term* by also including Skolem functor terms. The *Skolemization* of a program $\mathscr{P}$, denoted by $Skol(\mathscr{P})$, is obtained by replacing the head of each invention clause in $\mathscr{P}$ of the form $R(*, t_1, ..., t_n)$ by $R(f_R(t_1, ..., t_n), t_1, ..., t_n)$, where $f_R(t_1, ..., t_n)$ is the Skolem functor term for R built using the terms already present in the head of the clause.

It is possible to generalize the notion of instance taking Skolem functor terms into account. Given a program $\mathscr{P}$, the *Herbrand universe* $\mathscr{U}_{\mathscr{P}}$ *for* $\mathscr{P}$ is the set of all

ground terms built using domain elements from Δ and Skolem functors for invention relations in $sch(\mathscr{P})$. The *Herbrand base* $\mathscr{H}_{\mathscr{P}}$ *for* $\mathscr{P}$ is the set of all ground facts built using relation names in $sch(\mathscr{P})$ and terms in $\mathscr{U}_{\mathscr{P}}$. A *Herbrand interpretation over* $\mathscr{P}$ is a finite subset of $\mathscr{H}_{\mathscr{P}}$. Then, the notions of *Skolemized tuple*, *Skolemized relation instance*, and *Skolemized* (*database*) *instance over* $\mathscr{S}$ with respect to a program $\mathscr{P}$ are defined in the natural way, referring to the universe $\mathscr{U}_{\mathscr{P}}$ instead of the domain Δ. The set of all Skolemized database instances over a scheme $\mathscr{S}$ with respect to a program $\mathscr{P}$ is denoted $S\text{-}inst_{\mathscr{P}}(\mathscr{S})$.

In defining the semantics of a program $\mathscr{P}$, if the model of $\mathscr{P}$ over an instance $\mathscr{I}$ exists and is finite, then it is a Herbrand interpretation over $sch(\mathscr{P})$, that is, a Skolemized instance over $sch(\mathscr{P})$. Thus, focusing on steps 1–3 of the definition of the semantics, we define the *pre-semantics of a program* $(\mathscr{P}, \mathscr{S}, \mathscr{T})$ as a partial function $\psi_{\mathscr{P}}: inst(\mathscr{S}) \to S\text{-}inst_{\mathscr{P}}(\mathscr{T})$ that maps $\mathscr{I}$ to the Skolemized instance corresponding to $\mathscr{M}_{Skol(\mathscr{P}) \cup \mathscr{I}}$ restricted to the relation names in $\mathscr{T}$.

The replacement of different Skolem functor terms by distinct new values (Step 4) is defined in a nondeterministic fashion; therefore, if Skolem functor terms appear in the model of $\mathscr{P}$ over $\mathscr{I}$, then the semantics of $\mathscr{P}$ includes several possible outcomes (related to different choices of new values) and (by considering *all* possible replacements) it is in general a binary relation rather than a function.

3.3 *Safe Programs*

We now introduce syntactical sub-languages of ILOG$^{(\neg)}$ that limit the use of "invention" in programs; we follow analogous definitions in (Abiteboul and Vianu, 1991).

As we have seen, the semantics of an ILOG$^{(\neg)}$ program over an instance may lead to the introduction of new values, which are not in the active domain of the input database nor of the program itself; this fact contrasts with the notion of genericity, and the semantics of ILOG$^{(\neg)}$ programs, in general, is not a query in the usual sense. A program $(\mathscr{P}, \mathscr{S}, \mathscr{T})$ is *safe* if, for each instance $\mathscr{I}$ of $\mathscr{S}$, $\mathscr{P}(\mathscr{I})$ does not contain invented new values. It can be shown that safety of ILOG$^{(\neg)}$ programs is an undecidable property (even limiting our attention to positive ILOG). Hence, we consider two syntactical restrictions to ensure safety of programs.

A program is *strongly safe* if no invention clause occurs in it. It is apparent that the language of strongly safe ILOG$^{(\neg)}$ programs, denoted sILOG$^{(\neg)}$, syntactically and semantically corresponds to stratified *Datalog*$^{(\neg)}$.

Weak safety is defined with respect to an i–o scheme, using the auxiliary notion of "invention-attribute set." Given a program $\mathscr{P}$ over an i–o scheme $(\mathscr{S}, \mathscr{T})$, the *invention attributes for* $(\mathscr{P}, \mathscr{S}, \mathscr{T})$ are the smallest set of attributes such that:

- if R is an invention relation name in $sch(\mathscr{P})$, then (R, ID) is an invention attribute for $(\mathscr{P}, \mathscr{S}, \mathscr{T})$;
- if (R, A) is an invention attribute for $(\mathscr{P}, \mathscr{S}, \mathscr{T})$, $R(..., A : X, ...)$ is a positive literal in the body of a clause γ in $\mathscr{P}$, and $Q(..., A' : X, ...)$ is the head of γ, then (Q, A') is an invention attribute for $(\mathscr{P}, \mathscr{S}, \mathscr{T})$.

A program $\mathscr{P}$ is *weakly safe with respect to* $(\mathscr{S}, \mathscr{T})$ if no invention attribute for $(\mathscr{P}, \mathscr{S}, \mathscr{T})$ has the form (R, A), where R is a relation name in the output scheme $\mathscr{T}$. The language of weakly safe ILOG$^{(\neg)}$ programs is denoted by wILOG$^{(\neg)}$.

Intuitively, a program is weakly safe if invented values appear only in temporary relations, under suitable conditions, and do not appear in target relations. Specifically, the definition ensures that invented values are never "mixed-up" with values from the input active domain. Formally, it can be verified that in defining the semantics of a wILOG$^{(\neg)}$ program $(\mathscr{P}, \mathscr{S}, \mathscr{T})$, new values may appear only in tuples of the temporary relations $sch(\mathscr{P}) - \mathscr{T}$; since the temporary relations do not contribute to the result of the program, the assignment of distinct new values has no influence on the possible outcomes, and thus it is not strictly necessary. In particular, the semantics of wILOG$^{(\neg)}$ programs coincides with their pre-semantics.

3.4. Introductory Examples

The following three examples show the main features of the language; these examples are interesting because they illustrate techniques that will be used to prove results of this paper.

EXAMPLE 3.1. A (*total*) *enumeration of a finite set* R is a listing of the elements of R in any order, without repeats, and enclosed by brackets "[" and "]". For example, if $R = \{\mathtt{a}, \mathtt{b}\}$, then the enumerations of R are the lists `[ab]` and `[ba]`.

We now define an ILOG$^{\neg}$ program $\mathscr{P}_{code}$ that produces a representation of all the enumerations of a unary input relation R. Program $\mathscr{P}_{code}$ uses invention relations $list^{nil}(\text{ID})$ and $list^{cons}(\text{ID}, first, tail)$; values invented in these relations correspond to empty and nonempty lists, respectively; the target relation of the program is $list^{out}$, with the same scheme as $list^{cons}$. The program uses an auxiliary relation $misses(list, element)$ to denote which R's elements a list is still missing to obtain a total enumeration; relation $misses^{proj}$ is the restriction of $misses$ to the $list$ attribute, and it denotes the lists having at least an element missing. (In what follows, we will use the variable Nil to highlight terms that are intended to unify with values corresponding to an empty list.)

$$
\begin{aligned}
list^{nil}(*) &\leftarrow .\\
list^{cons}(*, \text{"]"}, Nil) &\leftarrow list^{nil}(Nil).\\
misses(RB, X) &\leftarrow list^{cons}(RB, \text{"]"}, Nil), list^{nil}(Nil), R(X).\\
list^{cons}(*, X, L) &\leftarrow misses(L, X).\\
misses(L, Y) &\leftarrow list^{cons}(L, X, L'), misses(L', Y), X \neq Y.\\
misses^{proj}(L) &\leftarrow misses(L, X).\\
list^{out}(*, \text{"[}", L) &\leftarrow list^{cons}(L, X, L'), \neg misses^{proj}(L).
\end{aligned}
$$

For an instance $\mathscr{I} = \{R(a), R(b)\}$, the pre-semantics for $\mathscr{P}_{code}$ on $\mathscr{I}$ contains in $list^{out}$ the functor terms $f^{out}(\text{"[}", f^{cons}(a, f^{cons}(b, f^{cons}(\text{"]"}, f^{nil}()))))$ and $f^{out}(\text{"[}", f^{cons}(b, f^{cons}(a, f^{cons}(\text{"]"}, f^{nil}()))))$, where f^{out}, f^{cons}, and f^{nil} are the

Skolem functor names associated with $list^{out}$, $list^{cons}$, and $list^{nil}$, respectively. These terms are the required representations for the enumerations of the relation R in $\mathscr{I}$.

Note that program $\mathscr{P}_{code}$ is stratified, with a stratification made of two strata (the second stratum being composed of the last clause only).

The following example shows that the construction of the "partial" enumerations of a set requires nonequality as the only form of negation.

EXAMPLE 3.2. A *partial enumeration of a finite set* R is an enumeration of a (possibly empty) subset of R.

An ILOG$^{\neq}$ program $\mathscr{P}_{pcode}$ that computes all the partial enumerations of an input unary relation R can be obtained from program $\mathscr{P}_{code}$ of Example 3.1 by replacing its last clause with the following one, in which the negated relation atom $\neg misses^{proj}(L)$ has been dropped.

$$list^{out}(*, \text{“[”}, L) \leftarrow list^{cons}(L, X, L').$$

The following example shows how to perform a transformation that is the inverse of the previous ones.

EXAMPLE 3.3. Consider a relation $list^{out}$ that represents lists of domain elements, enclosed by brackets; the representation uses relations $list^{nil}$ and $list^{cons}$ to encode intermediate lists, as indicated in Example 3.1.

The following program $\mathscr{P}_{decode}$ computes a unary relation R containing the domain elements occurring in the input lists. It uses a relation $toDecode(list)$ to select the lists to be decomposed. Relation R will contain the *union* of what we obtain by decomposing each list in $list^{out}$.

$$toDecode(L') \leftarrow list^{out}(L, \text{“[”}, L').$$
$$R(X) \leftarrow toDecode(L), list^{cons}(L, X, L'), X \neq \text{“]”}.$$
$$toDecode(L') \leftarrow toDecode(L), list^{cons}(L, X, L'), X \neq \text{“]”}.$$

$\mathscr{P}_{decode}$ is an ILOG$^{\neq}$ program. An equivalent but positive ILOG program can be obtained observing that the right bracket "]" is always followed by an empty list; thus, the test $X \neq$ "]" can be performed by testing instead for a tail which is a non-empty list, as follows:

$$toDecode(L') \leftarrow list^{out}(L, \text{“[”}, L').$$
$$R(X) \leftarrow toDecode(L), list^{cons}(L, X, L'), list^{cons}(L', X', L'').$$
$$toDecode(L') \leftarrow toDecode(L), list^{cons}(L, X, L'), list^{cons}(L', X', L'').$$

3.5. *Expressiveness of* sILOG$^{\neg}$ *and* wILOG$^{\neg}$

We conclude the section by recalling known results concerning expressiveness of ILOG$^{(\neg)}$ in the context of relational queries. In this respect, we should not consider

all ILOG$^{(\neg)}$ programs, because their semantics is not always a function, thus they do not always define a query in the strict sense. We consider instead sILOG$^{(\neg)}$ and wILOG$^{(\neg)}$ programs, in which the use of value invention is limited.

Strongly safe ILOG$^{(\neg)}$ corresponds syntactically and semantically to stratified *Datalog*$^{(\neg)}$, hence it inherits a lot of well-known results. Among others, we recall that *Datalog*$^{(\neg)}$ expresses (total) queries in PTIME; it would express the *fixpoint queries* if the inflationary semantics for negation were adopted instead (Abiteboul and Vianu, 1991); in *Datalog*$^{(\neg)}$, the stratified semantics is weaker than the inflationary one (Kolaitis, 1991).

On the other hand, wILOG$^{(\neg)}$ allows for value invention in temporary relations, in such a way that the semantics of every weakly safe program is always a query. The following result can be proved as the analogous result stated for weakly safe $Datalog_{\infty}^{\neg}$ programs in (Abiteboul and Vianu, 1991), even with a different semantics for negation and value invention.

Fact 3.4. Let $\mathscr{P}$ be a wILOG$^{\neg}$ program over an i-o scheme $(\mathscr{S}, \mathscr{T})$. The semantics of $(\mathscr{P}, \mathscr{S}, \mathscr{T})$ is a C-generic database mapping from $\mathscr{S}$ to $\mathscr{T}$, with $C = adom(\mathscr{P})$.

In presence of value invention (or a similar mechanism), it has been shown (Hull and Su, 1989; Hull and Su, 1997) that the expressive power of the stratified semantics is the same as the inflationary semantics. The following result characterizes the expressive power of weakly safe stratified ILOG$^{\neg}$ programs; it can be proven as the analogous result about the deductive language COL extended with recursive types (Hull and Su, 1997). There, hereditarily finite set construction is used instead of value invention; furthermore, COL programs must be stratified also with respect to set construction.

Fact 3.5. wILOG$^{\neg}$ expresses the computable queries.

4. EXPRESSIVENESS OF TWO-STRATA wILOG$^{\neg}$ PROGRAMS

In this section we introduce a syntactic hierarchy of ILOG$^{(\neg)}$ languages, based on limitations in the use of stratified negation. We then strengthen the result stated as Fact 3.5 by showing that the class of wILOG$^{\neg}$ programs made of two strata has the same expressive power of the whole wILOG$^{\neg}$, thus expressing the computable queries.

Let ILOG$^{i,\neg}$ be the class of ILOG$^{\neg}$ stratified programs having a stratification composed of *at most* $i+1$ strata, that is, programs that use i "groups" of negations. Then, consider the hierarchy consisting of the languages ILOG$^{i,\neg}$, for $i \geqslant 0$, and including also the following languages with a very limited use of negative literals: ILOG, the class of positive programs, with no negative literals at all; ILOG$^{\neq}$, the class of programs allowing for non-equality as the only form of negation; and ILOG$^{1/2,\neg}$, the class of *semipositive* programs, in which negation can be applied on

input relations only (nonequality is still allowed). Analogous hierarchies can be defined with respect to the languages sILOG$^{\neg}$ and wILOG$^{\neg}$.

With respect to the expressive power of the languages in these hierarchies, we have the following trivial relationships, based upon syntactical considerations:

$$\text{ILOG} \sqsubseteq \text{ILOG}^{\neq} \sqsubseteq \text{ILOG}^{1/2,\neg} \sqsubseteq \text{ILOG}^{1,\neg} \sqsubseteq \cdots \sqsubseteq \text{ILOG}^{\neg}$$

As shown in (Kolaitis, 1991), the above chain of containments is proper for the family sILOG$^{\neg}$. Interestingly, it collapses at level '1' for the family wILOG$^{\neg}$, as the following main result of the section shows.

THEOREM 4.1. wILOG$^{1,\neg}$ *expresses the computable queries.*

A comment is useful here. The language wILOG$^{1,\neg}$ is syntactically much simpler than the language wILOG$^{\neg}$: programs in wILOG$^{1,\neg}$ contain (at most) a positive stratum "followed" by a semipositive one, whereas wILOG$^{\neg}$ allows for an unbounded use of stratified negation. At the same time, Corollary 6.4 in Section 6 shows that the simpler language wILOG$^{1/2,\neg}$ of semipositive programs is less expressive than wILOG$^{1,\neg}$. The two facts together imply the "minimality" of wILOG$^{1,\neg}$ among the complete languages in this family.

Before proving the theorem (which strengthens Fact 3.5) we highlight the crucial points. The idea is to use wILOG$^{1,\neg}$ programs to simulate computations of queries as performed by *domain Turing Machines* (*domTMs*), a formal tool introduced in (Hull and Su, 1993) and which is already known to express the computable queries. Specifically, we show that for every query q there is a wILOG$^{1,\neg}$ program that: (i) encodes the input instance into a set of enumerations; (ii) performs the simulation of a domTM that computes q on these enumerations, yielding a number of enumerations of the output instance; and (iii) decodes the result enumerations to obtain the actual output of the query.

We now briefly describe domain Turing Machines and their properties. The main point in using domTMs to implement queries is that, unlike conventional Turing Machines, domTMs allow for a countable alphabet of symbols to be used on tapes. This alphabet includes both our domain Δ and a finite set W of connectives such as parentheses "(" and ")" and brackets "[" and "]". A domTM has a two-way infinite tape, and is equipped with a *register*, capable of storing a single symbol of the alphabet. This register, with the help of two special symbols η and κ, is used to express transitions that are (essentially) "generic" and to keep finite the control of the machine. Specifically, moves may only refer to a finite subset of the domain (corresponding to a set of interpreted domain elements) and to working symbols; in addition, it is possible to specify moves based on the (non)equality between the content of the register and that of the tape cell under the head. The possible effect of a move, apart from changing the internal state of the machine, is to change the content of the register and that of the tape cell under the head, then to move the head.

Formally, a (*deterministic*) *domain Turing machine* (*domTM*) (with respect to a domain Δ) is a 6-tuple $M = (K, W, C, \delta, q_s, q_h)$, as follows (Hull and Su, 1997).

- K is a finite set of *states*;
- W is a finite set of *working symbols* (in the discussion we shall assume that W includes the distinguished symbols "(", ")", "[", "]", used for encoding input and output, and also the blank symbol "#");
- $C \subseteq \Delta$ is a finite set of *constants*;
- $q_s \in K$ is the *starting state*;
- $q_h \in K$ is the unique *halting state*;
- δ is the *transition function*, a partial function from $(K - \{q_h\}) \times (W \cup C \cup \{\eta\}) \times (W \cup C \cup \{\eta, \kappa\})$ to $K \times (W \cup C \cup \{\eta, \kappa\})^2 \times \{\leftarrow, \rightarrow, -\}$. In a transition value $\delta(q, a, b) = (q', a', b', dir)$, q denotes the current domTM state, a the register content, and b the content of the tape cell under the head. Further, q' denotes the new domTM state, a' the new register content, b' the new content of the tape cell under the head, and *dir* the direction to move the head. Here, apart from constants in $W \cup C$, the two distinguished symbols η and κ can be used to model templates for infinite sets on transitions. It can be used $b = \kappa$ only if $a = \eta$; $\eta \in \{a', b'\}$ only if $\eta \in \{a, b\}$; and $\kappa \in \{a', b'\}$ only if $b = \kappa$.

An *instantaneous description* of a domTM is a 5-tuple (q, a, α, b, β), where $q \in K$, $a \in W \cup \Delta \cup \{\text{"\#"}\}$ is the *register content*, $\alpha, \beta \in (W \cup \Delta)^*$ and $b \in W \cup \Delta$ such that the *tape content* is $\alpha b \beta$, where the *tape head position* is the specified occurrence of b. As usual, it is assumed that neither the first symbol of α nor the last of β is "#".

A transition value $\delta(q, a, b) = (q', a', b', dir)$ is *generic* if $\eta \in \{a, b\}$. In generic transition values, the symbols η and κ are intended to range over distinct elements of $\Delta - C$: a transition value $\delta(q, \eta, \eta) = \cdots$ is "applicable" when the domTM is in state q and the content of the register and that of the tape cell under the head are equal, whereas $\delta(q, \eta, \kappa) = \cdots$ is applicable in the complementary situation in which they differ. A *computation* of M is defined in the usual fashion, assuming that the initial value held in the register is "#".

Given an instance $\mathscr{I}$, an *enumeration* of $\mathscr{I}$ is a sequential representation of $\mathscr{I}$ on a domTM tape, where domain elements are separated by connectives in W: tuples are enclosed within parentheses "(" and ")", and sets of tuples of different relations within brackets "[" and "]". The difference between instances and enumerations is essentially that instances are *sets* of tuples, whereas enumerations are *sequences*. We denote by $enum(\mathscr{I})$ the set of all enumerations of an instance $\mathscr{I}$. For example, if $\mathscr{I}$ is the instance $\{R_1(a), R_2(a, b), R_2(b, c)\}$ over $\{R_1, R_2\}$, then the enumerations of $\mathscr{I}$ (assuming the listing of R_1 precedes that of R_2) are $e_1 =$ `[(a)][(ab)(bc)]` and $e_2 =$ `[(a)][(bc)(ab)]`.

The correspondence between domTMs and queries is not yet complete. A domTM tape is used for input and processing; hence, the mapping computed by a domTM may depend on the input order. Fortunately, it is possible to restrict the attention to domTMs whose computations do not depend on the input order, in a sense made precise as follows. A domTM M_q, used to compute a query q from $\mathscr{S}$ to $\mathscr{T}$, is *order independent* if for each instance $\mathscr{I}$ of the input scheme $\mathscr{S}$, either: (i) for every enumeration of $\mathscr{I}$, M_q does not halt (meaning that q is undefined on input $\mathscr{I}$); or (ii) there is an instance $\mathscr{J}$ of the output scheme $\mathscr{T}$ such that, for every

enumeration e of $\mathscr{I}$, M_q halts and its output tape, denoted by $M_q(e)$, contains nothing but some enumeration of the instance $\mathscr{J}$, with the head of the domTM on the leftmost symbol of the tape (meaning that $q(\mathscr{I}) = \mathscr{J}$). For example, if $M_q(e_1) = [(ab)(bc)]$ and $M_q(e_2) = [(bc)(ab)]$, then M_q is order independent on the instance $\mathscr{I}$, and we can assume $q(\mathscr{I}) = \{(a, b), (b, c)\}$.

The following important result states that order independent domain Turing Machines provide an effective way to implement queries.

Fact 4.2. (Hull and Su, 1993). For every computable query q there is an order independent domTM M_q which computes q.

EXAMPLE 4.3. We now define part of the transition function δ of a domTM that computes $\sigma_{1 \neq 2}(R_2)$, having $\{R_1, R_2\}$ as input scheme, where R_2 is a binary relation. Function δ includes:

(1) $\delta(q_s, \text{“\#”}, \text{“[”}) = (q_1, \text{“\#”}, \text{“\#”}, \rightarrow)$
(2) $\delta(q_1, \text{“\#”}, \text{“(’)”}) = (q_1, \text{“\#”}, \text{“\#”}, \rightarrow)$
(3) $\delta(q_1, \text{“\#”}, \text{“)”}) = (q_1, \text{“\#”}, \text{“\#”}, \rightarrow)$
(4) $\delta(q_1, \text{“\#”}, \eta) = (q_1, \text{“\#”}, \text{“\#”}, \rightarrow)$
(5) $(q_1, \text{“\#”}, \text{“]”}) = (q_2, \text{“\#”}, \text{“\#”}, \rightarrow)$
(6) $\delta(q_2, \text{“\#”}, \text{“[”}) = (q_3, \text{“\#”}, \text{“]”}, \rightarrow)$
(7) $\delta(q_3, \text{“\#”}, \text{“(”}) = (q_4, \text{“\#”}, \text{“)”}, \rightarrow)$
(8) $\delta(q_4, \text{“\#”}, \eta) = (q_5, \eta, \eta, \rightarrow)$
(9) $\delta(q_5, \eta, \eta) = (q_6, \text{“\#”}, \text{“\#”}, \rightarrow)$
(10) $\delta(q_5, \eta, \kappa) = (q_6, \eta, \kappa, \rightarrow)$
(11) $\delta(q_6, \text{“\#”}, \text{“)”}) = (q_3, \text{“\#”}, \text{“)”}, \rightarrow)$
(12) $\delta(q_6, \text{“\#”}, \text{“]”}) = (q_7, \text{“\#”}, \text{“]”}, -)$

Here, transition values from (1) to (5) erase the encoding of the relation R_1 from the input tape. Then, transition (6) moves to the first tuple of the relation R_2. Transition (7) reads over the “(” of a tuple. Transition (8) remembers the first coordinate of a tuple in the register, which is then compared with the second coordinate in transitions (9) and (10). Specifically, transition (9) considers the case in which they are equal, marking the second coordinate of the tuple with “#”, so that it can be deleted later in the computation. Transition (10) considers the case in which they differ, and thus the tuple must be kept in the result. Transition (11) reads over the “)”, getting to the next tuple. Finally, transition (12) reads over the “]”, turning the control to state q_7. Starting from this state, the domTM should erase the tuples having “#” as second coordinate from the tape, and arrange the remaining tuples so that they are listed without separation (the details are not included here).

The computations of a query q from $\mathscr{S}$ to $\mathscr{T}$ can be simulated as follows:

1. Given an input instance $\mathscr{I}$ over $\mathscr{S}$, generate the family $enum(\mathscr{I})$ of all enumerations of $\mathscr{I}$. Note that, referring to an (essentially) deterministic language such as wILOG$^{(\neg)}$, it is not possible to generate a *single* enumeration of $\mathscr{I}$, so that *all* of them must be generated.

2. Let M_q be an order independent domain Turing Machine that computes q. Then, for every enumeration $e \in enum(\mathscr{I})$, simulate the computation of M_q on e, denoting by $M_q(e)$ the corresponding output tape. The various threads of simulations are performed simultaneously, and eventually result in an output enumeration for every enumeration of $\mathscr{I}$.

3. Decode the various output enumerations into instances over the output scheme; denote the result of decoding an output enumeration o by $decode(o)$. Then, take the union of such instances as the result of the overall process.

Following the above approach, starting from q, and so from M_q, our goal is to build a wILOG$^{1,\neg}$ program Q that computes the following query (on input instance $\mathscr{I}$):

$$Q(\mathscr{I}) = \bigcup_{e \in enum(\mathscr{I})} decode(M_q(e)).$$

The hypothesis of order independence on M_q guarantees that, for every enumeration e of $\mathscr{I}$, $decode(M_q(e)) = q(\mathscr{I})$; hence, $Q(\mathscr{I}) = q(\mathscr{I})$.

Proof of Theorem 4.1. We will show the following:

1. All the enumerations of an instance can be computed by a two-strata ILOG$^{\neg}$ program, where each enumeration is represented by means of a different invented value (and its corresponding Skolem term). We essentially use the technique shown in Example 3.1, albeit more complicated since, in general, we must deal with n-ary tuples, to be enclosed in parentheses; moreover, we must also concatenate enumerations of the various input relations.

2. The simulation of a domTM, starting from an enumeration and producing an output enumeration, can be done by an ILOG$^{\neq}$ program (without negation). Invented values are used to represent *strings* stored on the tape of the domTM. Termination of a computation happens within a finite number of acrossed instantaneous descriptions (which are represented by a finite number of strings), whereas a nontermination involves an infinite number of description crossings. Therefore, termination and nontermination correspond to a finite or an infinite number of invented values, respectively, hence to a finite or an infinite model of the program.

3. The decoding phase can be done by an ILOG (positive) program. The technique used is that of Example 3.3; intuitively, we perform the *union* of the results of decoding the various output enumerations.

4. The overall program, obtained by putting together the above three subprograms, is in wILOG$^{1,\neg}$, that is, is weakly safe and has a stratification made of two strata.

Note how we claim that, during the simulation, the only phase that needs stratified negation is the construction of the enumerations of the input instance. We use the technique of program $\mathscr{P}_{code}$ of Example 3.1; there, $\mathscr{P}_{code}$ is made of two strata: the output of the first stratum (which contains ILOG$^{\neq}$ clauses only) contains all partial enumerations of the input relation R; $\mathscr{P}_{code}$ resorts once to stratified negation (second stratum) to distinguish the *total* enumerations from the "incomplete" ones, selecting those lists for which *no* R's element is missing. Intuitively, we can build a two-strata program that computes the enumerations of the various input relations; then, to obtain the enumerations of the input instance, enumerations of the different relations need to be concatenated. We claim that we can do so without resorting to negation anymore.

Let q be a computable query from $\mathscr{S}$ to $\mathscr{T}$, and let M_q be an order independent domTM which computes q. We now define three programs Q_{in}, Q_{simu}, and Q_{out} based on M_q, and then we show that the program $Q = Q_{in} \cup Q_{simu} \cup Q_{out}$ is in wILOG$^{1,\neg}$ and that it indeed simulates the computations of the domTM M_q, that is, the semantics of Q coincides with q.

Enumeration of the input instance. The following program Q_{in} assigns to a unary relation ENC invented values corresponding to the possible encodings of enumerations of the input database, which is an instance over the input scheme $\mathscr{S} = \{R_1, ..., R_n\}$. To this end, we first encode each input relation R_i in $\mathscr{S}$ independently, and then concatenate them.

For the first part, assume that we must enumerate a binary relation R_i. We use invention relations ENC_i^{nil}(ID) (for the empty string), ENC_i^*(ID, *first*, *tail*) (for strings containing entire tuples), $ENC_i^)$, ENC_i^2, and ENC_i^1 (for strings containing part of tuples), and relation ENC_i (for the complete enumerations).

$$
\begin{aligned}
ENC_i^{nil}(*) &\leftarrow .\\
ENC_i^*(*, \text{“]”}, Nil) &\leftarrow ENC_i^{nil}(Nil).\\
misses_i(S, X_1, X_2) &\leftarrow ENC_i^*(S, \text{“]”}, Nil), ENC_i^{nil}(Nil), R_i(X_1, X_2).\\
ENC_i^)(*, \text{“)”}, S) &\leftarrow ENC_i^*(S, C, T), misses_i(S, X_1, X_2).\\
ENC_i^2(*, X_2, S_1) &\leftarrow ENC_i^)(S_1, \text{“)”}, S), ENC_i^*(S, C, T),\\
&\qquad misses_i(S, X_1, X_2).\\
ENC_i^1(*, X_1, S_2) &\leftarrow ENC_i^2(S_2, X_2, S_1), ENC_i^)(S_1, \text{“)”}, S),\\
&\qquad ENC_i^*(S, C, T), misses_i(S, X_1, X_2).\\
ENC_i^*(*, \text{“(”}, S_3) &\leftarrow ENC_i^1(S_3, X_1, S_2), ENC_i^2(S_2, X_2, S_1),\\
&\qquad ENC_i^)(S_1, \text{“)”}, S), ENC_i^*(S, C, T),\\
&\qquad misses_i(S, X_1, X_2).\\
misses_i(S', Y_1, Y_2) &\leftarrow ENC_i^*(S', \text{“(”}, S_3), ENC_i^1(S_3, X_1, S_2),\\
&\qquad ENC_i^2(S_2, X_2, S_1), ENC_i^)(S_1, \text{“)”}, S),\\
&\qquad ENC_i^*(S, C, T), misses_i(S, Y_1, Y_2), X_1 \neq Y_1.\\
misses_i(S', Y_1, Y_2) &\leftarrow ENC_i^*(S', \text{“(”}, S_3), ENC_i^1(S_3, X_1, S_2),\\
&\qquad ENC_i^2(S_2, X_2, S_1), ENC_i^)(S_1, \text{“)”}, S),\\
&\qquad ENC_i^*(S, C, T), misses_i(S, Y_1, Y_2), X_2 \neq Y_2.\\
misses_i^{proj}(S) &\leftarrow misses_i(S, X_1, X_2).\\
ENC_i(*, \text{“[”}, S) &\leftarrow ENC_i^*(S, C, T), \neg misses_i^{proj}(S).
\end{aligned}
$$

Note how we use stratified negation in the last clause only.

We now define relation $ENC_i^{cons}(string\_id, first, last)$ to represent all non-empty strings used in enumerating the relation R_i. We will use it to concatenate enumerations of the various relations.

$$ENC_i^{cons}(S, C, T) \leftarrow ENC_i^{*}(S, C, T).$$
$$ENC_i^{cons}(S, C, T) \leftarrow ENC_i^{)}(S, C, T).$$
$$ENC_i^{cons}(S, C, T) \leftarrow ENC_i^{2}(S, C, T).$$
$$ENC_i^{cons}(S, C, T) \leftarrow ENC_i^{1}(S, C, T).$$
$$ENC_i^{cons}(S, C, T) \leftarrow ENC_i(S, C, T).$$

It is worth noting that the foregoing set of clauses can be adapted to work with every input relation in $\mathscr{S}$, where a different number of invention relations must be used depending on the arity of the relation to be encoded.

To concatenate enumerations of the input relations, we use relation $ENC(string\_id)$, and invention relations $ENC^{nil}(\text{ID})$, $ENC^{cons}(\text{ID}, first, tail)$. All the clauses are positive.

$$ENC^{nil}(*) \leftarrow .$$
$$represents(Nil, Nil') \leftarrow ENC^{nil}(Nil), ENC_n^{nil}(Nil').$$
$$ENC^{cons}(*, C, S) \leftarrow ENC_n^{cons}(L, C, S'), represents(S, S').$$
$$represents(S, S') \leftarrow ENC^{cons}(S, C, T), ENC_n^{cons}(S', C, T'), represents(T, T').$$
$$represents(S, Nil) \leftarrow ENC_{n-1}^{nil}(Nil), ENC_n(S', C, T), represents(S, S').$$
$$ENC^{cons}(*, C, S) \leftarrow ENC_{n-1}^{cons}(L, C, S'), represents(S, S').$$
$$represents(S, S') \leftarrow ENC^{cons}(S, C, T), ENC_{n-1}^{cons}(S', C, T'), represents(T, T').$$
$$\cdots \leftarrow \cdots$$
$$ENC(S) \leftarrow ENC^{cons}(S, C, T), ENC_1(S', C', T'), represents(S, S').$$

Note that, denoting by $enum(R_i)$ the enumerations of input relation R_i, in this way we define an encoding for each element in the Cartesian product $enum(R_1) \times \cdots \times enum(R_n)$.

Simulation of the domTM M_q. We now define the program Q_{simu}, whose goal is to simulate computations of M_q on enumerations encoded in relation *ENC*.

We represent the instantaneous descriptions of M_q by means of invention relations ID^{nil} and ID^{cons}: the former stores starting descriptions and the latter successive ones. The scheme of ID^{nil} is (ID, *state*, *register*, *ltape*, *head*, *rtape*), and that of ID^{cons} includes also an attribute *previous_ID*. We use invention relations rather than ordinary relations in such a way that the simulating program has a finite model if and only if the simulation halts. It is worth noting that a cycling computation would across a finite number of different descriptions, and thus an ordinary relation

containing them would give rise to a finite model even in presence of such an infinite computation; in contrast, in the same situation, an invention relation would contain a tuple for each instantaneous description crossing, hence an infinite number of them, thus leading to an infinite model.

We use also a relation $ID(id, state, register, ltape, head, rtape)$ to summarize all acrossed instantaneous descriptions. Relations ID^{nil} and ID are defined as follows (where q_s is the unique starting state of the domTM):

$$ID^{nil}(*, q_s, \text{“\#”}, Nil, \text{“\#”}, X) \leftarrow ENC(X), ENC^{nil}(Nil).$$
$$ID(I, S, N, L, H, R) \leftarrow ID^{nil}(I, S, N, L, H, R).$$
$$ID(I, S, N, L, H, R) \leftarrow ID^{cons}(I, S, N, L, H, R, P).$$

Relations $first(string, first)$ and $tail(string, tail)$ are used to get the “first” character of a string and its “tail,” respectively. These relations are needed mainly to deal with “expansions” of the empty string, in case the head reaches an end of the tape.

$$first(X, F) \leftarrow ENC^{cons}(X, F, T).$$
$$first(Nil, \text{“\#”}) \leftarrow ENC^{nil}(Nil).$$
$$tail(X, T) \leftarrow ENC^{cons}(X, F, T).$$
$$tail(Nil, Nil) \leftarrow ENC^{nil}(Nil).$$

We are now ready to describe how to manage the transition function δ of M_q.

For each triple (q, a, b) such that $\delta(q, a, b) = (q', a', b', -)$, that is, for each nonmoving nongeneric transition value, we have a clause:

$$ID^{cons}(*, q', a', L, b', R, P) \leftarrow ID(P, q, a, L, b, R).$$

For a nonmoving generic transition value $\delta(q, \eta, \kappa) = (q', \kappa, a, -)$, we have a clause:

$$ID^{cons}(*, q', K, L, a, R, P) \leftarrow ID(P, q, N, L, K, R), N \neq K.$$

For a right-moving generic transition value $\delta(q, \eta, a) = (q', a', \eta, \rightarrow)$, we have clauses:

$$expand(L, N) \leftarrow ID(I, q, N, L, a, R).$$
$$ID^{cons}(*, q', a', L', A', R', P) \leftarrow ID(P, q, N, L, a, R), first(L', N), tail(L', L), first(R, A'), tail(R, R').$$

For a left-moving generic transition value $\delta(q, \eta, \eta) = (q', \eta, a, \leftarrow)$, we have clauses:

$$expand(R, a) \leftarrow ID(I, q, N, L, N, R).$$

$$ID^{cons}(*, q', N, L', A', R', P) \leftarrow ID(P, q, N, L, N, R), first(L, A'), tail(L, L'), first(R', a), tail(R', R).$$

Similarly for the other types of moves.

The following clause is used to invent new values to represent new needed strings, according to requests made using the relation $expand(string, element)$:

$$ENC^{cons}(*, A, S) \leftarrow expand(S, A).$$

During the computation of the least fixpoint of the foregoing set of clauses, threads of simulations associated with different input enumerations evolve independently. Looking at a single thread of simulation, at each stage of the fixpoint computation all the previous IDs related to the thread are re-derived (but without generating any new value or fact), in addition to a single new ID (possibly with an associated new string). When the halting state is reached, no new ID is generated; a fixpoint is reached when all independent threads of simulations reach the halting state.

Because M_q computes q, the output tapes of halting computations of M_q correspond to enumerations of the output instance (that is, legal instances of the target scheme). We store in relation ENC^{out} the values representing strings in the output tapes (we have a different output tape for each different enumeration of the input instance) using the following clause (where q_h is the unique halting state of the domTM):

$$ENC^{out}(S) \leftarrow ID(I, q_h, \text{“\#”}, Nil, \text{“\#”}, S), ENC^{nil}(Nil).$$

Decoding the output. The following program Q_{out} decodes the various output enumerations into an instance of the target scheme $\mathscr{T}$. For the sake of simplicity and without loss of generality, we assume $\mathscr{T} = \{T\}$, that is, the result of q is a single relation T.

We decode the strings corresponding to enumerations of the output instance, which we stored in relation ENC^{out}. For example, assume that the target relation T is binary; as in Example 3.3, we use an auxiliary relation *toDecode*:

$$toDecode(S') \leftarrow ENC^{out}(S), ENC^{cons}(S, \text{“[”}, S').$$

$$T(X_1, X_2) \leftarrow toDecode(S), ENC^{cons}(S, \text{“(”}, S_1), ENC^{cons}(S_1, X_1, S_2), ENC^{cons}(S_2, X_2, S_3).$$

$$toDecode(S') \leftarrow toDecode(S), ENC^{cons}(S, \text{“(”}, S_1), ENC^{cons}(S_1, X_1, S_2), ENC^{cons}(S_2, X_2, S_3), ENC^{cons}(S_3, \text{“)”}, S').$$

Correctness of the simulation. Consider the overall program $Q = Q_{in} \cup Q_{simu} \cup Q_{out}$, and recall that it has been build starting from a query q from $\mathscr{S}$ to $\mathscr{T}$.

The membership of program Q, with i–o scheme $(\mathscr{S}, \mathscr{T})$, in wILOG$^{1,\neg}$ is easily proved by inspection.

To prove that the semantics of Q coincides with q, we show the following statements:

1. For each instance $\mathscr{I}$ over $\mathscr{S}$, Q has finite model over $\mathscr{I}$ if and only if q is defined over $\mathscr{I}$.
2. For each instance $\mathscr{I}$ over $\mathscr{S}$ such that $q(\mathscr{I})$ is defined, $Q(\mathscr{I}) = q(\mathscr{I})$.

To prove Statement 1, consider an instance $\mathscr{I}$. If q is defined over $\mathscr{I}$, then M_q halts on every enumeration of $\mathscr{I}$. Denote by $|\mathscr{I}|$ the number of symbols used to represent $\mathscr{I}$ (that is, the length of any of its enumerations) and, for an enumeration e of $\mathscr{I}$, denote by $\Omega_q(e)$ the finite number of steps in the computation of $M_q(e)$. It can be shown that the number of invented values in the model of Q on input $\mathscr{I}$ is $O(\Sigma_{e \in enum(\mathscr{I})}(\Omega_q(e) + |\mathscr{I}|))$ (we use the standard "big oh" notation), which is finite; hence, the number of facts in this model (which is polynomial in the cardinality of $adom(\mathscr{I})$ and the number of invented values) is finite, that is, the model of Q over $\mathscr{I}$ is finite. Similarly, it can be proven that the model of Q over $\mathscr{I}$ can be computed (according to a fixpoint semantics) in $O(\max_{e \in enum(\mathscr{I})}(\Omega_q(e)) + |\mathscr{I}|)$ stages.

On the other hand, if q is undefined on input $\mathscr{I}$, then M_q does not halt on any enumeration of $\mathscr{I}$. Let e one such enumeration; the computation $M_q(e)$ acrosses an infinite number of instantaneous descriptions; this yields an infinite number of invented values in relation ID^{cons}, hence in an infinite model for Q.

Statement 2 follows from the correctness of the decoding phase, that is, from the ability of program Q_{out} to "parse" enumerations of instances over the target scheme from output tapes into the target relations. ∎

5. EXPRESSIVENESS OF POSITIVE PROGRAMS

In this section we characterize the expressive power of weakly safe ILOG$^{\neq}$; the only negative literals the language allows for are nonequalities. Since it disallows other forms of negation, we do not expect this language to express nonmonotone queries. Interestingly, we show that the language is able to express *all* the monotone computable queries; before stating the result formally, we need to discuss the notion of monotonicity in a framework allowing for r.e. queries—that is, queries that can be partial functions.

For total queries, the notion of monotonicity is defined as follows: A (total) query q from $\mathscr{S}$ to $\mathscr{T}$ is *monotone* if, for each pair of instances $\mathscr{I}$, $\mathscr{J}$ over $\mathscr{S}$, $\mathscr{I} \subseteq \mathscr{J}$ implies $q(\mathscr{I}) \subseteq q(\mathscr{J})$. Intuitively, the result of a monotone query does not decrease by adding new elements to the active domain of the input instance and tuples to the input relations.

The language sILOG$^{\neq}$ coincides with *Datalog*$^{\neg}$. It is well-known that *Datalog*$^{\neq}$ expresses only monotone (PTIME) queries (and thus, total monotone queries). Unfortunately, this monotonicity result cannot be directly generalized to wILOG$^{\neq}$, since the latter expresses partial queries as well.

A query q from $\mathscr{S}$ to $\mathscr{T}$ is $\subseteq$-*defined* if, for each pair of instances $\mathscr{I}$, $\mathscr{J}$ over $\mathscr{S}$, $\mathscr{I} \subseteq \mathscr{J}$ and q defined over $\mathscr{J}$ imply that q is also defined over $\mathscr{I}$. A (partial) query q from $\mathscr{S}$ to $\mathscr{T}$ is *monotone* if it is $\subseteq$-defined and, for each pair of instances $\mathscr{I}$, $\mathscr{J}$ over $\mathscr{S}$, $\mathscr{I} \subseteq \mathscr{J}$ and q defined over $\mathscr{J}$ imply that $q(\mathscr{I}) \subseteq q(\mathscr{J})$.[1]

An example of boolean monotone (partial) query is the one that, given a binary relation representing a directed graph G over a fixed set of nodes, answers *true* (that is, the non-empty 0-ary relation $\{()\}$) if G is planar and contains a Hamiltonian circuit, answers *false* (that is, $\{\}$) if G, being planar, does not contain any Hamiltonian circuit, and is undefined if G is not planar. In other words, this query decides, for planar graphs, if they do contain Hamiltonian circuits, and does not halt on nonplanar graphs.

Reasoning on the fixpoint semantics of ILOG$^{\neq}$ programs, we obtain the following result:

LEMMA 5.1 *Let $\mathscr{P}$ be a* wILOG$^{\neq}$ *program. The semantics of $\mathscr{P}$ is a monotone query.*

To prove the lemma, we need to introduce an operator associated with a set of clauses, which can be used to find the perfect model of a program via a fixpoint computation.

A *substitution* is a total function from variables to ground terms (including Skolem terms). For an instance $\mathscr{I}$ and a ground literal L, the notion of *satisfaction* is defined as usual, and is extended in the natural way to sets of ground literals. For a set Γ of clauses over a scheme $\mathscr{S} = sch(\Gamma)$, the *immediate consequence operator* T_Γ *for* Γ is a mapping T_Γ: *S-inst*$_\Gamma(\mathscr{S}) \to$ *S-inst*$_\Gamma(\mathscr{S})$ defined as follows:

$$T_\Gamma(\mathscr{I}) = \{\theta\, head(\gamma) \mid \gamma \in Skol(\Gamma), \mathscr{I} \text{ satisfies } \theta\, body(\gamma) \text{ for a substitution } \theta\}.$$

Proof of Lemma 5.1. Consider the immediate consequence operator $T_{\mathscr{P}}$ for the wILOG$^{\neq}$ program $\mathscr{P}$, and instances $\mathscr{I} \subseteq \mathscr{J}$.

It is clear that the operator $T_{\mathscr{P}}$ is monotone, that is, $T_{\mathscr{P}}(\mathscr{I}) \subseteq T_{\mathscr{P}}(\mathscr{J})$. This remains true for powers of $T_{\mathscr{P}}$, used in the computation of the least fixpoint of $T_{\mathscr{P}}$ (that is, the minimum model for $\mathscr{P}$): $T^n_{\mathscr{P}}(\mathscr{I}) \subseteq T^n_{\mathscr{P}}(\mathscr{J})$ for every $n \geqslant 0$. Hence, if the sequence of powers $T^n_{\mathscr{P}}$ does not converge over the instance $\mathscr{I}$, it does not converge over $\mathscr{J}$ either, that is, $\mathscr{P}$ is $\subseteq$-defined.

[1] One could argue another definition of monotonicity (let us call it $*$-monotonicity) for a partial query, by requiring the property to be verified only for the instances on which the query is defined, as follows: A (partial) query q from $\mathscr{S}$ to $\mathscr{T}$ is $*$-*monotone* if, for each pair of instances $\mathscr{I}$, $\mathscr{J}$ over $\mathscr{S}$, we have that $q(\mathscr{I}) \subseteq q(\mathscr{J})$ whenever $\mathscr{I} \subseteq \mathscr{J}$ and q is defined on both $\mathscr{I}$ and $\mathscr{J}$. The two notions are different, but related. Specifically, it turns out that a partial query q is monotone if and only if it is $*$-monotone and $\subseteq$-defined.

To prove monotonicity, assume that the fixpoint computation converges over the instance $\mathscr{J}$, as $T^{\omega}_{\mathscr{P}}(\mathscr{J})$. It then clearly converges over $\mathscr{I}$ as well, with $T^{\omega}_{\mathscr{P}}(\mathscr{I}) \subseteq T^{\omega}_{\mathscr{P}}(\mathscr{J})$. ∎

The above is a typical result for a database programming language: Lemma 5.1. states that all the queries that are expressible in a syntactically defined language satisfy a semantically defined property as well. In such cases, it is interesting to ask whether the language expresses *all* the queries that satisfy the property, or only a part of them. We devote the remainder of the section to show the following result, which strengthens the connection between wILOG$^{\neq}$ and the class of monotone queries.

THEOREM 5.2. wILOG$^{\neq}$ *expresses the monotone queries.*

Proof. Let q be a monotone query. Consider an order independent domTM M_q that computes q. For an input instance $\mathscr{I}$, to evaluate $q(\mathscr{I})$ we would like to simulate a computation of M_q on an enumeration e of $\mathscr{I}$. Because of genericity, we must consider computations of M_q on all enumerations of $\mathscr{I}$ rather than on a single one, as in the proof of Theorem 4.1. However, computing the enumerations of an instance is not a monotone operation; thus we must slightly modify our evaluation strategy.

Let us show what happens if we consider all the "partial" enumerations of $\mathscr{I}$, rather than "total" ones only; note that this operation is monotone. Let *p-enum*$(\mathscr{I})$ be the set of all *partial enumerations of* $\mathscr{I}$, that is, the set:

$$p\text{-}enum(\mathscr{I}) = \bigcup_{\mathscr{J} \subseteq \mathscr{I}} enum(\mathscr{J}).$$

(For example, the set of the partial enumerations for an instance $\{R_1(a), R_2(a, b), R_2(b, c)\}$ includes, among others, `[(a)][]`, `[][(bc)]`, and `[(a)][(bc)(ab)]`, the latter being a total enumeration.)

If we simulate computations of M_q on this set and take the union of the results, we in turn evaluate a query $\tilde{Q}$ defined as follows:

$$\tilde{Q}(\mathscr{I}) = \bigcup_{e \in p\text{-}enum(\mathscr{I})} decode(M_q(e)) = \bigcup_{\mathscr{J} \subseteq \mathscr{I}} \bigcup_{e \in enum(\mathscr{J})} decode(M_q(e)) = \bigcup_{\mathscr{J} \subseteq \mathscr{I}} q(\mathscr{J}). \quad (1)$$

We now prove that $\tilde{Q}(\mathscr{I}) = q(\mathscr{I})$. From $\subseteq$-definedness of q, it follows that $\tilde{Q}$ is defined on input $\mathscr{I}$ if and only if q is; from its monotonicity, we have $q(\mathscr{J}) \subseteq q(\mathscr{I})$ for every $\mathscr{J} \subseteq \mathscr{I}$. In turn, $\tilde{Q}(\mathscr{I}) = q(\mathscr{I})$.

Hence, to prove the theorem, it suffices to show that the evaluation strategy $\tilde{Q}$ for q can be implemented in wILOG$^{\neq}$. So, consider the program $Q = Q_{in} \cup Q_{simu} \cup Q_{out}$, defined with respect to a query q and a corresponding domTM M_q as in the proof of Theorem 4.1. Recall that Q computes q by simulating computations of M_q on all total enumerations of the input instance; recall also that both Q_{simu} and Q_{out} make use of nonequality as the only form of negation. Modify the program Q_{in} by removing its negated relation atoms, thus defining a new program $\tilde{Q}_{in}$; it is apparent that $\tilde{Q}_{in}$ belongs to ILOG$^{\neq}$. Program $\tilde{Q}_{in}$ is obtained from Q_{in}

as program $\mathscr{P}_{pcode}$ of Example 3.2 is obtained from program $\mathscr{P}_{code}$ of Example 3.1. It can be shown that $\tilde{Q}_{in}$, on input $\mathscr{I}$, indeed generates in relation *ENC* the representatives of all partial enumerations of $\mathscr{I}$.

Consider now the program $\tilde{Q} = \tilde{Q}_{in} \cup Q_{simu} \cup Q_{out}$; it belongs to wILOG$^{\neq}$. Furthermore Q_{simu} simulates computations of M_q on enumerations represented in relation *ENC*, and Q_{out} decodes the results and takes their union. Hence, because of Equation (1), the semantics of $\tilde{Q}$ coincides with q.

6. EXPRESSIVENESS OF SEMIPOSITIVE PROGRAMS

In this section we study the expressive power of the language wILOG$^{1/2, \neg}$ of semipositive programs, that is, the class of programs in which negation can be applied to input relations only (nonequality is also allowed). This language is strictly more expressive than wILOG$^{\neq}$; indeed, wILOG$^{1/2, \neg}$ allows to express some nonmonotone queries as well, for example the difference of two input relations. Hence, it is interesting to ask whether this language expresses the computable queries, as wILOG$^{1, \neg}$ does. We first give a negative answer to this question, by proving that the queries defined by wILOG$^{1/2, \neg}$ programs satisfy a weak form of monotonicity; we call "semimonotone" these queries. We then strengthen the result by proving that wILOG$^{1/2, \neg}$ expresses exactly this class of queries.

We need a few preliminary definitions.

Given an instance $\mathscr{I}$ over a scheme $\mathscr{S} = \{R_1, ..., R_n\}$, consider the *enriched scheme* $\bar{\mathscr{S}} = \{R_1, ..., R_n, \overline{R_1}, ..., \overline{R_n}\}$ for $\mathscr{S}$, and the *enriched instance* $\bar{\mathscr{I}}$ (over $\bar{\mathscr{S}}$) for $\mathscr{I}$, defined in such a way that, for $1 \leqslant i \leqslant n$, the relation $\overline{R_i}$ has the same scheme as R_i, $\bar{\mathscr{I}}(R_i) = \mathscr{I}(R_i)$, and $\bar{\mathscr{I}}(\overline{R_i})$ is the complement of $\mathscr{I}(R_i)$ with respect to the active domain $adom(\mathscr{I})$, that is, $\bar{\mathscr{I}}(\overline{R_i}) = adom(\mathscr{I})^{\alpha(R_i)} - \mathscr{I}(R_i)$.

Given a wILOG$^{1/2, \neg}$ program $\mathscr{P}$ over an input scheme $\mathscr{S} = \{R_1, ..., R_n\}$, we can easily eliminate negation from $\mathscr{P}$ by considering the program $\bar{\mathscr{P}}$ over the enriched input scheme $\bar{\mathscr{S}}$ obtained from $\mathscr{P}$ by replacing each negative literal $\neg R_i(...)$ by $\overline{R_i}(...)$; call this program $\bar{\mathscr{P}}$ the *positivization of* $\mathscr{P}$. It turns out that, for every wILOG$^{1/2, \neg}$ program $\mathscr{P}$, its positivization $\bar{\mathscr{P}}$ is a wILOG$^{\neq}$ program. Furthermore, for every input instance $\mathscr{I}$ for $\mathscr{P}$, if $\mathscr{P}$ is defined over $\mathscr{I}$, it is the case that $\mathscr{P}(\mathscr{I}) = \bar{\mathscr{P}}(\bar{\mathscr{I}})$, otherwise $\bar{\mathscr{P}}$ is undefined over $\bar{\mathscr{I}}$. It is this strict relationship between the languages wILOG$^{1/2, \neg}$ and wILOG$^{\neq}$ that induces a limitation on the expressiveness of the former.

Given a scheme $\mathscr{S} = \{R_1, ..., R_n\}$ and instances $\mathscr{I}$, $\mathscr{J}$ over $\mathscr{S}$, we say that $\mathscr{J}$ is an *extension of* $\mathscr{I}$ (written $\mathscr{I} \preccurlyeq \mathscr{J}$) if $adom(\mathscr{I}) \subseteq adom(\mathscr{J})$ and, for $1 \leqslant i \leqslant n$, it is the case that $\mathscr{I}(R_i) = \mathscr{J}(R_i)|_{adom(\mathscr{I})}$, that is, the restriction of the relation $\mathscr{J}(R_i)$ to the active domain of $\mathscr{I}$ coincides with $\mathscr{I}(R_i)$.

LEMMA 6.1. *Let $\mathscr{I}$, $\mathscr{J}$ be instances over a scheme $\mathscr{S}$, and let $\bar{\mathscr{I}}$, $\bar{\mathscr{J}}$ be their enriched instances. Then $\mathscr{J}$ is an extension of $\mathscr{I}$ if and only if $\bar{\mathscr{I}} \subseteq \bar{\mathscr{J}}$.*

Proof. Assume that $\mathscr{J}$ is an extension of $\mathscr{I}$, that is, (i) $adom(\mathscr{I}) \subseteq adom(\mathscr{J})$ and (ii) $\mathscr{I}(R) = \mathscr{J}(R)|_{adom(\mathscr{I})}$, for each relation R in $\mathscr{S}$. We now prove that, for each relation R in $\mathscr{S}$, it holds (iii) $\bar{\mathscr{I}}(R) \subseteq \bar{\mathscr{J}}(R)$ and (iv) $\bar{\mathscr{I}}(\bar{R}) \subseteq \bar{\mathscr{J}}(\bar{R})$.

Inclusion (iii) follows directly from (ii):

$$\bar{\mathscr{I}}(R) = \mathscr{I}(R) \subseteq \mathscr{J}(R) = \bar{\mathscr{J}}(R).$$

For inclusion (iv), we have:

$$\begin{aligned}\bar{\mathscr{I}}(\bar{R}) &= adom(\mathscr{I})^{\alpha(R)} - \mathscr{I}(R) \\ &= adom(\mathscr{I})^{\alpha(R)} - \mathscr{J}(R)|_{adom(\mathscr{I})} \\ &= adom(\mathscr{I})^{\alpha(R)} - \mathscr{J}(R) \\ &\subseteq adom(\mathscr{J})^{\alpha(R)} - \mathscr{J}(R) = \bar{\mathscr{J}}(\bar{R}).\end{aligned}$$

For the converse direction, assume inclusions (iii) and (iv) hold. Inclusion (i) is immediately implied by (iii). We now prove (ii) by showing containment in the two directions,

$$\begin{aligned}\mathscr{J}(R)|_{adom(\mathscr{I})} &= (adom(\mathscr{J})^{\alpha(R)} - \bar{\mathscr{J}}(\bar{R}))\,|_{adom(\mathscr{I})} \\ &= adom(\mathscr{I})^{\alpha(R)} - \bar{\mathscr{J}}(\bar{R})|_{adom(\mathscr{I})} \\ &\subseteq adom(\mathscr{I})^{\alpha(R)} - \bar{\mathscr{I}}(\bar{R}) = \mathscr{I}(R),\end{aligned}$$

whereas $\mathscr{I}(R) \subseteq \mathscr{J}(R)|_{adom(\mathscr{I})}$ follows from (iii). ∎

A (total) query $q: \mathscr{S} \to \mathscr{T}$ is *preserved under extensions* (Afrati *et al.*, 1991) if, for every pair of instances $\mathscr{I}$, $\mathscr{J}$ in $inst(\mathscr{S})$, whenever $\mathscr{J}$ is an extension of $\mathscr{I}$ it is the case that $q(\mathscr{I}) \subseteq q(\mathscr{J})$. Intuitively, the result of a query preserved under extensions does not decrease by adding new elements to the input active domain and tuples containing at least a new element to the input relations.

We now generalize this property to the framework of partial queries. A (partial) query $q: \mathscr{S} \to \mathscr{T}$ is $\preccurlyeq$-*defined* if, for every pair of instances $\mathscr{I}$, $\mathscr{J}$ over $\mathscr{S}$, $\mathscr{I} \preccurlyeq \mathscr{J}$ and q defined over $\mathscr{J}$ imply that q is also defined over $\mathscr{I}$. A (partial) query $q: \mathscr{S} \to \mathscr{T}$ is *preserved under extensions* if, for every pair of instances $\mathscr{I}$, $\mathscr{J}$ over $\mathscr{S}$, $\mathscr{I} \preccurlyeq \mathscr{J}$ and q defined over $\mathscr{J}$ imply that $q(\mathscr{I}) \subseteq q(\mathscr{J})$.[2]

Note that $\mathscr{I} \preccurlyeq \mathscr{J}$ implies $\mathscr{I} \subseteq \mathscr{J}$, but the converse does not hold in general; similarly, $\subseteq$-definedness implies $\preccurlyeq$-definedness, but the converse is not always implied.

Preservation under extensions is a weak form of monotonicity. In what follows, we shall however use a different terminology for this property, by calling *semi-monotone* a query that is preserved under extensions. The next result, in conjunction with Theorem 6.5, motivates our choice for giving this name to the property.

[2] Again, a different notion of preservation under extensions can be defined by requiring the property to be verified only for the instances on which a query is defined, as follows: A (partial) query q from $\mathscr{S}$ to $\mathscr{T}$ is $*$-*preserved under extensions* if, for every pair of instances $\mathscr{I}$, $\mathscr{J}$ over $\mathscr{S}$, we have that $q(\mathscr{I}) \subseteq q(\mathscr{J})$ whenever $\mathscr{I} \preccurlyeq \mathscr{J}$ and q is defined on both $\mathscr{I}$ and $\mathscr{J}$. It turns out that a partial query q is preserved under extensions if and only if it is $*$-preserved under extensions and $\preccurlyeq$-defined.

LEMMA 6.2. *Let $\mathscr{P}$ be a semipositive* wILOG$^{1/2,\neg}$ *program. The semantics of $\mathscr{P}$ is a semimonotone query.*

Proof. Consider the positivization $\overline{\mathscr{P}}$ of $\mathscr{P}$. Let $\mathscr{I}$, $\mathscr{J}$ be instances over the input scheme of $\mathscr{P}$ such that $\mathscr{J}$ is an extension of $\mathscr{I}$, and $\overline{\mathscr{I}}$, $\overline{\mathscr{J}}$ the corresponding enriched instances. By Lemma 5.1, the semantics of $\overline{\mathscr{P}}$ is a monotone query. Assume $\mathscr{P}$ defined on input $\mathscr{J}$; then, so is $\overline{\mathscr{P}}$ on input $\overline{\mathscr{J}}$. Now, $\overline{\mathscr{I}} \subseteq \overline{\mathscr{J}}$, and by $\subseteq$-definedness of $\overline{\mathscr{P}}$, the latter is defined on input $\overline{\mathscr{I}}$; moreover, because of its monotonicity, $\overline{\mathscr{P}}(\overline{\mathscr{I}}) \subseteq \overline{\mathscr{P}}(\overline{\mathscr{J}})$. Hence, $\mathscr{P}$ is defined on input $\mathscr{I}$ and $\mathscr{P}(\mathscr{I}) \subseteq \mathscr{P}(\mathscr{J})$. ∎

LEMMA 6.3. *Let CTC be the query that computes the complement of the transitive closure of a binary relation. Then CTC is not semimonotone.*

Proof. Consider the scheme $G = \{N, E\}$ for representing a directed graph, where N is unary (the nodes) and E is binary (the edges). Consider instances $\mathscr{I}$ and $\mathscr{J}$ over G, where $\mathscr{I}(N) = \{(a)\}$ (a single node) and $\mathscr{I}(E) = \varnothing$ (no edges), and $\mathscr{J}(N) = \{(a), (b)\}$ and $\mathscr{J}(E) = \{(a, b), (b, a)\}$; $\mathscr{J}$ is an extension of $\mathscr{I}$ (indeed, $\overline{\mathscr{I}} \subseteq \overline{\mathscr{J}}$). Now, $CTC(\mathscr{I}) = \{(a, a)\}$, whereas $CTC(\mathscr{J}) = \varnothing$; hence, the query is not preserved under extensions. ∎

As a consequence of Lemmas 6.2 and 6.3, *CTC* is not expressible in wILOG$^{1/2,\neg}$, that is, we have a query separating the class of semipositive programs from the computable queries.

COROLLARY 6.4. wILOG$^{\neq} \sqsubset$ wILOG$^{1/2,\neg} \sqsubset$ wILOG$^{1,\neg}$.

Other queries belong to wILOG$^{1,\neg}$ − wILOG$^{1/2,\neg}$. Consider the following queries *min* and *max* defined over a scheme containing a binary relation *succ*, intuitively used to represent a successor relation over an ordered domain. The queries are defined as

$$min(succ) = \{x \mid \nexists w : succ(w, x)\};$$

$$max(succ) = \{x \mid \nexists w : succ(x, w)\}.$$

It can be shown, by means of examples as in the proof of Lemma 6.3, that these queries are not semimonotone.

Again, it raises naturally the question of whether the language wILOG$^{1/2,\neg}$ expresses the semimonotone queries, or only part of them. The remainder of the section is devoted to show that wILOG$^{1/2,\neg}$ indeed expresses this class of queries.

THEOREM 6.5. wILOG$^{1/2,\neg}$ *expresses the semimonotone queries.*

Proof. The proof is similar in spirit to that of Theorem 5.2. However, the enumeration phase requires here a major modification with respect to that used in the proof of Theorem 4.1.

Consider a semimonotone query q. Consider also an order independent domTM M_q that computes q. For an input instance $\mathscr{I}$, our evaluation strategy can neither consider computation of M_q on an enumeration e of $\mathscr{I}$ (because of genericity) nor

computations on all total enumerations of $\mathscr{I}$ (because the operation of computing the set $enum(\mathscr{I})$ is not preserved under extensions). Is there any suitable set of enumerations derivable from $\mathscr{I}$ such that: (i) this set is expressible by means of a semipositive program; and (ii) the union of the results of the computations of M_q on this set yields $q(\mathscr{I})$? Fortunately, the answer is affirmative. To prove formally the result, we need some preliminary considerations and definitions.

Let $\mathscr{I}$ be an instance and $D \subseteq adom(\mathscr{I})$ be a subset of its active domain; denote by $\mathscr{I}|_D$ the restriction of $\mathscr{I}$ to the domain D, that is, the instance obtained from $\mathscr{I}$ by considering only the facts involving constants in D. From the definition of extension, it follows that $\mathscr{I}|_D \preccurlyeq \mathscr{I}$. With respect to the active domain of $\mathscr{I}|_D$, note that in general only the inclusion $adom(\mathscr{I}|_D) \subseteq D$ holds, whereas the equality $adom(\mathscr{I}|_D) = D$ does not necessarily follow, because it is possible that $\mathscr{I}|_D$ does not include some elements from the domain D on which it has been built.

Starting from an instance $\mathscr{I}$, by considering its restrictions to all subsets of its active domain, we obtain all instances for which $\mathscr{I}$ is an extension:

$$\{\mathscr{I}|_D \mid D \subseteq adom(\mathscr{I})\} = \{\mathscr{J} \mid \mathscr{J} \preccurlyeq \mathscr{I}\}.$$

Let $r\text{-}enum(\mathscr{I})$ be the set of enumerations of the instances obtained from $\mathscr{I}$ in such a way that

$$r\text{-}enum(\mathscr{I}) = \bigcup_{D \subseteq adom(\mathscr{I})} enum(\mathscr{I}|_D).$$

If we simulate the computations of M_q on this set, taking the union of the results, we in turn evaluate the query $\hat{Q}$ defined as follows:

$$\begin{aligned}\hat{Q}(\mathscr{I}) &= \bigcup_{e \in r\text{-}enum(\mathscr{I})} decode(M_q(e)) \\ &= \bigcup_{D \subseteq adom(\mathscr{I})} \bigcup_{e \in enum(\mathscr{I}|_D)} decode(M_q(e)) \\ &= \bigcup_{D \subseteq adom(\mathscr{I})} q(\mathscr{I}|_D) = \bigcup_{\mathscr{J} \preccurlyeq \mathscr{I}} q(\mathscr{J}).\end{aligned}$$

Since q is $\preccurlyeq$-defined, $\hat{Q}$ is defined on input $\mathscr{I}$ whenever q is; because of its semimonotonicity, $q(\mathscr{J}) \subseteq q(\mathscr{I})$ for every $\mathscr{J} \preccurlyeq \mathscr{I}$, hence $\hat{Q}(\mathscr{I}) = q(\mathscr{I})$.

We now define an $ILOG^{1/2,\neg}$ program $\hat{Q}_{in}$ as follows. First, we define a unary relation *a_dom* to store the active domain of the input database. Then, we build all the partial enumerations of this set *a_dom*. Any partial enumeration d of *a_dom* is a total enumeration of a subset D of $adom(\mathscr{I})$; besides, it naturally induces a total order on its elements (any enumeration being a list without repeats): while we build the enumerations, at the same time we define relations *min*, *max*, and *succ* to make apparent the total orders associated with them. Starting from enumerations and using total orders, we iterate on their elements to build encodings of enumerations of the input relations; we use semipositive negation in this phase. Then, we

concatenate encodings of the input relations—without resorting to negation anymore, as we did in the proof of Theorem 4.1.

Finally, the wILOG$^{1/2,\neg}$ program $\hat{Q}$ is defined by putting $\hat{Q}_{in}$ together with programs Q_{simu} and Q_{out} as in the proof of Theorem 4.1.

The remainder of the proof is devoted to the definition of $\hat{Q}_{in}$.

Relation $a\_dom(element)$ is defined using a clause for each relation R_i of the input scheme $\mathscr{S}$ and for each attribute (R_i, A) of R_i:

$$a\_dom(X_A) \leftarrow R_i(..., X_A, ...).$$

We use the invention relations $enc^{nil}(\text{ID})$, $enc^{cons}(\text{ID}, first, tail)$ and the relation $enc(string\_id)$ to represent partial enumerations of a_dom. At the same time, we define total orders using relations $min(enum, first)$ (to store and propagate the first element inserted into an enumeration), $max(enum, last)$ (to store the last element inserted into), and $succ(enum, element, successor)$ (to store and propagate a successor relation):

$$\begin{aligned}
enc^{nil}(*) &\leftarrow .\\
misses(Nil, X) &\leftarrow enc^{nil}(Nil), a_dom(X).\\
enc^{cons}(*, X, L) &\leftarrow misses(L, X).\\
misses(L, X) &\leftarrow enc^{cons}(L, Y, L'), misses(L', X), X \neq Y.\\
enc(L) &\leftarrow enc^{nil}(L).\\
enc(L) &\leftarrow enc^{cons}(L, X, L').\\
min(L, X) &\leftarrow enc^{cons}(L, X, Nil), enc^{nil}(Nil).\\
min(L, X) &\leftarrow enc^{cons}(L, Y, L'), min(L', X).\\
max(L, X) &\leftarrow enc^{cons}(L, X, L').\\
succ(L, X, Y) &\leftarrow enc^{cons}(L, Y, L'), enc^{cons}(L', X, L'').\\
succ(L, X, Y) &\leftarrow enc^{cons}(L, W, L'), succ(L', X, Y).
\end{aligned}$$

Then, we build encodings of input relations starting from the partial enumerations of a_dom and their associated total orders, as follows. Note how we keep track of the originating enumeration of a_dom: for instance, invention relation ENC_i^{nil} has scheme (ID, *enum*) instead of simply (ID) as in program Q_{in} in the proof of Theorem 4.1.

Consider an input relation R_i; assume it is binary. We iterate on the possible tuples over R_i, and test membership in the input instance. If the tuple belongs to the input, we encode it and continue the iteration; if the tuple does not belong to the input (we use semipositive negation here) we simply skip it and continue the iteration.

$ENC_i^{nil}(*, P)$ $\leftarrow enc(P).$

$ENC_i^*(*, \text{“]”}, Nil, P)$ $\leftarrow ENC_i^{nil}(Nil, P).$

$ENC_i^)(*, \text{“)”}, S, P)$ $\leftarrow toAppend_i(S, P, X_1, X_2.$

$ENC_i^2(*, X_2, S_1, P)$ $\leftarrow ENC_i^)(S_1, \text{“)”}, S, P), toAppend_i(S, P, X_1, X_2).$

$ENC_i^1(*, X_1, S_2, P)$ $\leftarrow ENC_i^2(S_2, X_2, S_1, P), ENC_i^)(S_1, \text{“)”}, S, P),$
$toAppend_i(S, P, X_1, X_2).$

$ENC_i^*(*, \text{“(”}, S_3, P)$ $\leftarrow ENC_i^1(S_3, X_1, S_2, P), ENC_i^2(S_2, X_2, S_1, P),$
$ENC_i^)(S_1, \text{“)”}, S, P), toAppend_i(S, P, X_1, X_2).$

$Represents_i(S', P, X_1, X_2)$ $\leftarrow ENC_i^*(S', \text{“(”}, S_3, P), ENC_i^1(S_3, X_1, S_2, P),$
$ENC_i^2(S_2, X_2, S_1, P), ENC_i^)(S_1, \text{“)”}, S, P),$
$toAppend_i(S, P, X_1, X_2).$

$toAppend_i(S, P, X, X)$ $\leftarrow ENC_i^*(S, \text{“]”}, Nil, P), ENC_i^{nil}(Nil, P),$
$min(P, X), R_i(X, X).$

$Represents_i(S, P, X, X)$ $\leftarrow ENC_i^*(S, \text{“]”}, Nil, P), ENC_i^{nil}(Nil, P),$
$min(P, X), \neg R_i(X, X).$

$toAppend_i(S, P, X_1, X_2')$ $\leftarrow Represents_i(S, P, X_1, X_2), succ(P, X_2, X_2'),$
$R_i(X_1, X_2').$

$toAppend_i(S, P, X_1', X_2')$ $\leftarrow Represents_i(S, P, X_1, X_2), max(P, X_2), min(P, X_2'),$
$succ(P, X_1, X_1'), R_i(X_1', X_2').$

$Represents_i(S, P, X_1, X_2')$ $\leftarrow Represents_i(S, P, X_1, X_2), succ(P, X_2, X_2'),$
$\neg R_i(X_1, X_2').$

$Represents_i(S, P, X_1', X_2')$ $\leftarrow Represents_i(S, P, X_1, X_2), max(P, X_2), min(P, X_2'),$
$succ(P, X_1, X_1'), \neg R_i(X_1', X_2').$

$ENC_i(*, \text{“[”}, S, P)$ $\leftarrow Represents_i(S, P, X, X), max(P, X).$

$ENC_i(*, \text{“[”}, S, Nil')$ $\leftarrow ENC_i^*(S, \text{“]”}, Nil, Nil'), ENC_i^{nil}(Nil, Nil'), enc^{nil}(Nil').$

The foregoing set of clauses, written to encode a binary relation, can be modified so as to build encodings of each input relation in $\mathscr{S}$, using a different number of invention relations depending on its arity.

The following clauses are meant to build a uniform representation of the non-empty strings occurring in the enumerations.

$$ENC_i^{cons}(S, C, T, P) \leftarrow ENC_i^{*}(S, C, T, P).$$
$$ENC_i^{cons}(S, C, T, P) \leftarrow ENC_i^{)}(S, C, T, P).$$
$$ENC_i^{cons}(S, C, T, P) \leftarrow ENC_i^{2}(S, C, T, P).$$
$$ENC_i^{cons}(S, C, T, P) \leftarrow ENC_i^{1}(S, C, T, P).$$
$$ENC_i^{cons}(S, C, T, P) \leftarrow ENC_i(S, C, T, P).$$

We now concatenate encodings of enumerations of the various input relations. To obtain enumerations of instances for which $\mathscr{I}$ is an extension, we must only concatenate those encodings originating from the same partial enumeration of *a_dom*. We use clauses similar to those used in the proof of Theorem 4.1; again, to keep track of the partial enumerations that defined the encodings, we use an additional attribute (as in the above clauses).

$$ENC^{nil}(*, P) \leftarrow enc(P).$$
$$represents(Nil, Nil', P) \leftarrow ENC^{nil}(Nil, P), ENC_n^{nil}(Nil', P).$$
$$ENC^{cons}(*, C, S) \leftarrow ENC_n^{cons}(L, C, S', P), represents(S, S', P).$$
$$represents(S, S', P) \leftarrow ENC^{cons}(S, C, T), ENC_n^{cons}(S', C, T', P), represents(T, T', P).$$
$$represents(S, Nil, P) \leftarrow ENC_{n-1}^{nil}(Nil, P), ENC_n(S', C, T, P), represents(S, S', P).$$
$$ENC^{cons}(*, C, S) \leftarrow ENC_{n-1}^{cons}(L, C, S', P), represents(S, S', P).$$
$$represents(S, S', P) \leftarrow ENC^{cons}(S, C, T), ENC_{n-1}^{cons}(S', C, T', P), represents(T, T', P).$$
$$\ldots \leftarrow \ldots$$
$$ENC(S) \leftarrow ENC^{cons}(S, C, T), ENC_1(S', C', T', P), represents(S, S', P). \blacksquare$$

A comment is useful here. We used two different approaches in proving completeness of the various languages: on the one hand, we preferred to compute (partial) enumerations of the *input active domain* in the case of wILOG$^{1/2, \neg}$; on the other hand, we computed (partial) enumerations of the *input relations* in the cases of wILOG$^{1, \neg}$ and wILOG$^{\neq}$.

Actually, we can devise a different completeness proof for wILOG$^{1, \neg}$ (Theorem 4.1) by using the "active domain" approach, as follows. We first compute the total enumerations of the input active domain (using one stratified negation) and then the total enumerations of the input relations (using semipositive negation only). Note that, in some cases, this way of proceeding does not lead to the generation of

all the enumerations of the input relations; this fact, however, does not invalidate the proof.

The above approach does not seem to be useful in proving expressiveness of monotone queries for wILOG$^{\neq}$ (Theorem 5.2); intuitively, having a total order at disposal is indeed useful to build enumerations of the input instance only if we can apply negation (semipositive negation, at least) to the input relations.

The proof of Theorem 6.5 suggests that it would be possible for a wILOG$^{1/2,\neg}$ program to compute a total enumeration of an input instance if a total order on the input domain were given. Indeed, the following result shows that wILOG$^{1/2,\neg}$ expresses the computable queries provided a total order is given. Formally, a database is *ordered* if it includes a binary relation (conventionally denoted *succ*) containing a successor relation on all the constants occurring in the active domain of the database, and two unary relations (denoted *min* and *max*) containing the minimum and maximum element according to the successor relation. A *query on an ordered database* is a query whose input scheme is an ordered database scheme and that ranges only over ordered instances.

COROLLARY 6.6 wILOG$^{1/2,\neg}$ *expresses the computable queries on ordered databases.*

7. TOWARD THE STRONGLY MONOTONE QUERIES?

All the results proved in this paper refer essentially to simulations of computations of domain Turing machines done by suitable programs in subclasses of wILOG$^{(\neg)}$. In all the cases, the difficult part of the proof was concerned with the ability of the language to enumerate the input instance as a list of domain constants and connectives. We proved that wILOG$^{1,\neg}$ is able to build exactly the enumerations of an input instance, that wILOG$^{\neq}$ can build enumerations of the instances contained ($\subseteq$) in an input instance, and that wILOG$^{1/2,\neg}$ can build enumerations of instances for which the input instance is an extension ($\preccurlyeq$). Then, simulations of domTMs can be carried over these enumerations without resorting to negation anymore (nonequality is required, however); finally, the results of the computations can be decoded with no negation and no nonequality.

The expressive power of the language wILOG, in which the use of negative literals is totally disallowed, remains to be characterized.

It seems natural to compare this language with the class of queries satisfying a stronger form of monotonicity, called *strong monotonicity* in (Kolaitis and Vardi, 1990; Afrati *et al.*, 1991) and defined there with respect to total queries. Given instances $\mathcal{I}$, $\mathcal{J}$ over a same scheme $\mathcal{S}$, a *homomorphism from* $\mathcal{I}$ *to* $\mathcal{J}$ is a function h: $adom(\mathcal{I}) \to adom(\mathcal{J})$ such that (extending h to facts and instances in the natural way) $h(\mathcal{I}) \subseteq \mathcal{J}$; we denote with $\mathcal{I} \xrightarrow{h} \mathcal{J}$ the existence of such a homomorphism. A query q: $\mathcal{S} \to \mathcal{T}$ is *strongly monotone* if it is preserved under homomorphisms, that is, for every pair of instances $\mathcal{I}$, $\mathcal{J}$ over $\mathcal{S}$, $\mathcal{I} \xrightarrow{h} \mathcal{J}$ implies $h(q(\mathcal{I})) \subseteq q(\mathcal{J})$. Intuitively, the result of a strongly monotone query does not decrease by adding new elements to the input active domain, adding tuples to the input relations, and identifying elements of the active domain.

We can generalize the property to partial queries, as follows. A (partial) query $q: \mathscr{S} \to \mathscr{T}$ is $\xrightarrow{h}$*-defined* if, for every pair of instances $\mathscr{I}$, $\mathscr{J}$ over $\mathscr{S}$, q defined over $\mathscr{J}$ and $\mathscr{I} \xrightarrow{h} \mathscr{J}$ imply that Q is defined over $\mathscr{I}$. Then, a (partial) query $q: \mathscr{S} \to \mathscr{T}$ is said to be *strongly monotone* if, for every pair of instances $\mathscr{I}$, $\mathscr{J}$ over $\mathscr{S}$, q defined over $\mathscr{J}$ and $\mathscr{I} \xrightarrow{h} \mathscr{J}$ imply that $h(q(\mathscr{I})) \subseteq q(\mathscr{J})$.

Now, it turns out that the semantics of every wILOG program is a strongly monotone query (again, by reasoning on its fixpoint semantics as in the proof of Lemma 5.1). In this case, however, there is no evidence of the ability of the language to express all queries in that class. In particular, we have two arguments suggesting that the proof schemes used in this paper are unuseful to eventually characterize expressiveness of wILOG.

First, the approach of resorting to domain Turing machines as an effective way to implement a query cannot be pursued. In fact, transition values of the form $\delta(q, \eta, \kappa) = \ldots$ are inherently required in domTMs (that is, domTMs without this kind of transition values are not a formalism that expresses the strongly monotone queries); to simulate these transition values, nonequality literals must be used, and we do not have them in wILOG.

Second, observe that for any (finite) input instance $\mathscr{I}$, both the set of instances contained in $\mathscr{I}$ and the set of instances for which $\mathscr{I}$ is an extension are finite; this is in contrast with the fact that the set of instances $\{\mathscr{J} \mid \mathscr{J} \xrightarrow{h} \mathscr{I}\}$ from which a (nonempty) input instance $\mathscr{I}$ can be obtained by means of a homomorphism is in general infinite. This fact must prevent us to use any naive wILOG program to build enumerations of this set of instances obtained from the input instance (rather than the total enumerations only, which cannot directly built because the corresponding operation is not strongly monotone).

These difficulties leave us with the open problem of characterizing the expressive power of wILOG, and for the quest of defining a formalism to express the class of strongly monotone queries.

8. DISCUSSION

In this paper we have introduced a family of rule-based query languages with value invention and stratified negation. We have defined a hierarchy of languages based on limitations in the use of stratified negation in wILOG$^{\neg}$ programs. The main contribution is the characterization of the expressiveness of the following languages: wILOG$^{1,\neg}$, the class of programs made of a positive program followed by a semipositive one; wILOG$^{\neq}$, the class of programs allowing for nonequality comparisons; and wILOG$^{1/2,\neg}$, the class of semipositive programs, allowing for negation on input relations and nonequality; more precisely, we have shown that these languages express the computable queries of Chandra and Harel (Theorem 4.1), the monotone computable queries (Theorem 5.2), and the semimonotone computable queries (Theorem 6.5), respectively. To the best of our knowledge, this is the first proposal of languages expressing exactly the latter two classes of queries.

Corollary 6.4 allows us to argue that wILOG$^{1,\neg}$ is a "minimal" formalism among those expressing the computable queries. It is important to note that the

minimality of wILOG$^{1,\neg}$ potentially implies some "inefficiency" in expressing queries. Consider, for instance, some stratified *Datalog*$^{\neg}$ query belonging to *Datalog*$^{i,\neg}$ for some $i>1$ (but not with less strata); call P the program expressing this query. It is clear that the data complexity of evaluating P is in PTIME. The query is clearly expressible in wILOG$^{1,\neg}$ as well; call P' the program expressing the same query in this language. Note that P' is certainly different from P, because P does not belong syntactically to wILOG$^{1,\neg}$. The computation strategy of P' has to be different from that of P, since P' cannot use the mechanism of strata to alternate existential and universal quantifications. Thus, P' will then probably use the mechanism of constructing enumerations, which has in general an exponential cost, and therefore we cannot ensure that the complexity of finding a model for P' is in PTIME.

A comparison between the expressiveness of the family wILOG$^{(\neg)}$ and that of stratified *Datalog*$^{(\neg)}$ allows us to highlight the impact that value invention has in querying relational databases. The two families of languages differ a lot in expressiveness: the former ranges over computable queries, whereas the latter does not go beyond PTIME queries. The hierarchy of wILOG$^{(\neg)}$ with respect to the allowed number of strata collapses at level '1' (wILOG$^{1,\neg}$, Theorem 4.1); the same hierarchy referred to *Datalog*$^{(\neg)}$ does not collapse (Kolaitis, 1991). Moreover, comparing the result in (Kolaitis, 1991) with that in (Abiteboul and Vianu, 1991), it turns out that the stratified semantics for negation in *Datalog*$^{(\neg)}$ is weaker than the inflationary one; in contrast, the two semantics for negation (though different) have been shown equally expressive in rule-based languages having a mechanism comparable to that of value invention (Hull and Su, 1989).

Referring to languages with a limited use of negation, it is known that the queries expressible in *Datalog*$^{\neq}$ and *Datalog*$^{1/2,\neg}$ are monotone and semimonotone (preserved under extensions) PTIME queries, respectively. However, these two languages fail to express exactly the two classes of queries (Kolaitis and Vardi, 1990; Afrati *et al.*, 1991). In contrast, wILOG$^{\neq}$ and wILOG$^{1/2,\neg}$ express *exactly* the classes of monotone and semimonotone computable queries, respectively.

The language *Datalog*$^{1/2,\neg}$ expresses the PTIME queries on ordered databases (Papadimitriou, 1985). We have shown a similar result for the language wILOG$^{1/2,\neg}$ with respect to the computable queries.

This work is clearly related to the original paper introducing ILOG (Hull and Yoshikawa, 1990). However, there the focus is on query issues in the context of an object-based data model, whereas our main concern is on the ability of expressing relational queries, especially with respect to a limited use of stratified negation.

The first completeness result for a *Datalog* extension with value invention was shown in (Abiteboul and Vianu, 1991); the proof technique of building all enumerations of an input instance was also proposed there. However, the connection between the family wILOG$^{(\neg)}$ and the *Datalog* extensions proposed in (Abiteboul and Vianu, 1991) is looser than it might appear. Indeed, *Datalog*$^{\neg}_{\infty}$ adopts the inflationary semantics for negation and a different semantics for value invention, making the language "operational." As a consequence, even the semantics of "similar" wILOG$^{(\neg)}$ and *Datalog*$^{\neg}_{\infty}$ programs with limited use of negation (that is, semipositive, or no negation at all) can be different.

The ability of wILOG$^\neg$ (with unbounded stratified negation) to express the computable queries can be inferred from (Hull and Su, 1997). There, the results refer to an extension of the rule-based language COL with recursive types (untyped set construction). The two approaches are comparable, as it is suggested in (Van den Bussche *et al.*, 1997), where value invention is related to hereditarily finite set construction. However, COL programs have to be stratified with respect to set construction as well; furthermore, negation in COL can be simulated using set construction (Abiteboul and Grumbach, 1991); because of this, it is not clear whether the results concerning wILOG$^\neg$ with limited use of negation can be generalized to corresponding languages in (Hull and Su, 1997).

Languages with value invention (or object creation) specify mappings such that new values (outside the input active domain) may appear in their result; this fact, in turn, implies a potential violation of the criterion of genericity. Because of the nondeterministic choice of new values, the semantics of such mappings define binary relations between databases, rather than functions. These mappings are called *database transformations*. Criterions that extend genericity in the framework of transformations are (among others) *determinacy* (Abiteboul and Kanellakis, 1989) and *constructivism* (Van den Bussche *et al.*, 1997). The subject of querying object-oriented databases has been also investigated by other authors, e.g., (Gyssens *et al.*, 1994; Denninghoff and Vianu, 1993; Kifer *et al.*, 1992). Expressiveness of ILOG$^\neg$ as a database transformation languages has been formalized in (Cabibbo, 1996) as the class of *list-constructive transformations*, that is, "generic" transformations in which new values in the result can be put in correspondence with nested lists constructed by means of input values. (List-constructive transformations have been introduced in (Van den Bussche *et al.*, 1997) as a subclass of the *constructive queries*, where the latter refer to "hereditarily finite sets" rather than lists.) The results holding for ILOG$^\neg$ are the analogues of those proven for wILOG$^\neg$; more precisely, the class of two-strata programs expresses the list-constructive transformations, and ILOG$^{\neq}$ and ILOG$^{1/2}$ express the class of monotone and semimonotone list-constructive transformations, respectively.

Received April 24, 1997; final manuscript received February 19, 1998

REFERENCES

Abiteboul, S., and Grumbach, S. (1991), A rule-based language with functions and sets., *ACM Trans. Database Syst.* **16(1)**, 1–30.

Abiteboul, S., Hull, R., and Vianu, V. (1995), "Foundations of Databases," Addison-Wesley, Reading, MA.

Abiteboul, S., and Kanellakis, P. (1989), Object identity as a query language primitive, *in* "ACM SIGMOD International Conference on Management of Data" pp. 159–173.

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

Abiteboul, S., and Vianu, V. (1991), Datalog extensions for database queries and updates, *J. Comput. Syst. Sci.* **43**(1), 62–124.

Afrati, F., Cosmadakis, S., and Yannakakis, M. (1991), On Datalog vs. polynomial time, *in* "Tenth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems," pp. 13–25.

Aho, A., and Ullman, J. (1979), Universality of data retrieval languages, *in* "Sixth ACM Symposium on Principles of Programming Languages," pp. 110–117.

Apt, K. (1990), Logic programming, *in* "Handbook of Theoretical Computer Science" (J. Van Leeuwen, Ed.), , Vol. B, pp. 493–574, Elsevier Science Publishers, Amsterdam.

Bancilhon, F. (1978), On the completeness of query languages for relational databases, *in* "Mathematical Foundations of Computer Science, LNCS 64," pp. 112–124, Springer-Verlag, Berlin.

Cabibbo, L. (1996), "Querying and Updating Complex-Object Databases," Ph.D. thesis, Università degli Studi di Roma "La Sapienza".

Chandra, A., and Harel, D. (1980), Computable queries for relational databases, *J. Comput. Syst. Sci.* **21**, 333–347.

Chen, W., and Warren, D. (1989), C-logic for complex objects, *in* "Eigth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems," pp. 369–378.

Codd, E. (1970), A relational model for large shared data banks, *Commun. ACM* **13**(6), 377–387.

Codd, E. (1970), Extending the database relational model to capture more meaning, *ACM Trans. Database Syst.* **4**(4), 397–434.

Denninghof, K., and Vianu, V. (1993), Database method schemas and object creation, *in* "Twelfth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems," pp. 265–275.

Gyssens, M., Paredaens, J., Van den Bussche, J., and Van Gucht, D. (1994), A graph-oriented object database model, *IEEE Trans. Knowledge Data Eng.* **6**(4), 572–586.

Hull, R., and Su, J. (1989), Untyped sets, invention, and computable queries, *in* "Eigth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems," pp. 347–359.

Hull, R., and Su, J. (1993), Algebraic and calculus query languages for recursively typed complex objects, *J. Comput. Syst. Sci.* **47**(1), 121–156.

Hull, R., and Su, J. (1994), Domain independence and the relational calculus, *Acta Informatica* **31**, 513–524.

Hull, R., and Su, J. (1997), Deductive query languages for recursively typed complex objects, *J. Logic Programming*, to appear.

Hull, R., and Yoshikawa, M. (1990), ILOG: Declarative creation and manipulation of object identifiers, *in* "Sixteenth International Conference on Very Large Data Bases, Brisbane" pp. 455–468,.

Hull, R., and Yoshikawa, M. (1991), On the equivalence of database restructurings involving object identifiers, *in* "Tenth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems," pp. 328–340.

Immerman, N. (1986), Relational queries computable in polynomial time, *Inform. Control* **68**, 86–104.

Kanellakis, P. (1990), Elements of relational database theory, *in* "Handbook of Theoretical Computer Science" (J. van Leeuwen, Ed.), Vol. B, pp. 1073–1156, Elsevier Science Publishers, Amsterdam.

Kifer, M., Kim, W., and Sagiv, Y. (1992), Querying object-oriented databases, *in* "ACM SIGMOD International Conference on Management of Data," pp. 393–402.

Kifer, M., Lausen, G., and Wu, J. (1995), Logical foundations of object-oriented and frame-based languages, *J. ACM* **42**(4), 741–843.

Kifer, M., and Wu, J. (1993), A logic for programming with complex objects, *J. Comput. Syst. Sci.* **47**(1), 77–120.

Kolaitis, P. (1991), The expressive power of stratified logic programs, *Inform. Computation* **90**(1), 50–66.

Kolaitis, P., and Vardi, M. (1990), On the expressive power of Datalog: Tools and a case study, *in* "Ninth ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems," pp. 61–71.

Kuper, G., and Vardi, M. (1984), A new approach to database logic, *in* "Third ACM SIGACT SIGMOD Symposium on Principles of Database Systems," pp. 86–96.

Kuper, G., and Vardi, M. (1993), The logical data model, *ACM Trans. on Database Systems* **18**(3), 379–413.

Maier, D. (1986), A logic for objects, *in* "Workshop on Foundations of Deductive Database and Logic Programming (Washington, D.C. 1986)," pp. 6–26.

Papadimitriou, C. (1985), A note on the expressive power of prolog, *Bull. EATCS* **26**, 21–23.

Paredaens, J. (1978), On the expressive power of the relational algebra, *Information Processing Lett.* **7**(2), 107–111.

Van den Bussche, J., Van Gucht, D., Andries, M., and Gyssens, M. (1997), On the completeness of object-creating database transformation languages, *J. ACM* **44**(2), 272–319.

Vardi, M. (1982), The complexity of relational query languages, *in* "Fourteenth ACM SIGACT Symposium on Theory of Computing," pp. 137–146.