
Università degli Studi di Roma Tre
Dipartimento di Discipline Scientifiche
Via della Vasca Navale, 84 – 00146 Roma, Italy.

The Expressive Power of Stratified Logic Programs with Value Invention

Luca Cabibbo

RT-INF-11-96 **Maggio 1996**

Part of 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, [10].*
This work was partially supported by Consiglio Nazionale delle Ricerche*, by* MURST*, within the Project "Metodi formali e strumenti per basi di dati evolute", and by* Università di Roma Tre*, within the Project "Sistemi intelligenti."*

ABSTRACT

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 whole expressive power of the family, thus expressing the computable queries. We then show that the language wILOG$^{\neq}$ of programs with non-equality and without negation expresses the monotone queries, and that the language wILOG$^{\frac{1}{2},\neg}$ of semipositive programs expresses the semimonotone queries.

1 Introduction

The study of query languages is a major issue in database theory. Departing from the relational calculus and algebra query languages for the relational model of data [14], 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 by Chandra and Harel [12] of the *computable queries* as a 'reasonable' class of mappings from databases to databases. The computable queries are the class of partial recursive functions between finite relational structures that satisfy a criterion called genericity. The notion of genericity formalizes the *data independence principle* in databases; it intuitively states that the only significant relationships among data are based on (non-)equality of values. Genericity is a generalization of invariant properties of the queries that are expressible by the relational algebra and calculus [6, 9, 35].

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 [25, 38]. In fact, queries in such languages may use only relations of fixed scheme and constants from the input database, hence 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 of variable scheme, or for the use of constants outside the active domain of the input database; several proposals in the literature consider indeed either one or the other way toward completeness. The former approach was pursued in [12] 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 by Abiteboul and Vianu [3, 4] introducing a mechanism of *value invention* as a means to allow for new domain elements in temporary relations during computations. They embedded value invention both in a procedural language [3] and in a rule-based one [4].

A different mechanism to achieve completeness was proposed by Hull and Su [20, 21], 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, essentially corresponding to hereditarely finite sets. The connection between hereditarely finite set construction and value invention was shown by Van den Bussche et al.[37], which reconciled the two approaches.

The idea of value invention originates from a proposal by Kuper and Vardi [31, 32] to choose arbitrary symbolic *object names* to manage new complex object values defined in their logical queries. The concept of object name is a refinement of Codd's notion of *surrogate* [15]; nowadays, we use the term *object identity*. The mechanism of value invention has been recasted into an object-oriented data model within a traditional database framework in the language IQL (Abiteboul and Kanellakis [2]). 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 incorporated also in the rule-based language ILOG (Hull and Yoshikawa [23]); ILOG adopts a different (and more declarative) semantics for object creation, using Skolem functor terms as suggested in previous proposals [7, 33, 13, 27, 28].

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, extend-

ing 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$^\neg$ [23], 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, i.e., by allowing invented values in temporary relations only.) The language so obtained (called wILOG$^\neg$) expresses the computable queries of Chandra and Harel (this fact can be formally proved as a consequence of previous results by Hull and Su [20, 22]). 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, i.e., 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 non-equality, expresses the monotone queries, i.e., all the computable queries that satisfy the monotonicity property. We then study the language wILOG$^{\frac{1}{2},\neg}$ of semipositive programs, in which negation can be applied to input relations only. The language is shown to express the semimonotone queries, i.e., queries that satisfy a weak monotonicity property, also called 'queries preserved under extensions' in the literature [5].

The results shown provide interesting counterpoints to results concerning stratified *datalog*$^\neg$, highlighting 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 study the query languages wILOG$^{1,\neg}$, wILOG$^\neq$, and wILOG$^{\frac{1}{2},\neg}$, that express the class 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. Finally, the Appendix is a brief introduction to domain Turing machines, a technical tool introduced by Hull and Su [21] and widely used in proofs of the main results.

2 Preliminaries

2.1 The data model

We assume the reader to be familiar with the *relational model* [14]. We now briefly review the basic notions and notations.

Assume the existence of disjoint countable sets $\mathcal{L}_r$ of *relation names* and $\mathcal{L}_a$ of *attribute names*.

A *relation scheme* is a relation name $R \in \mathcal{L}_r$ together with a (possibly empty) finite set of attribute names in $\mathcal{L}_a$. We write $R(A_1 \ldots A_n)$ to indicate a relation scheme with name R and set of attributes $\{A_1, \ldots, A_n\}$. 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* $\mathcal{S}$ is a finite set of relation schemes, having distinct names.

Let Δ be a countable set of *constants*, called the *domain*. For a set X of attributes names, a *tuple* t *over* X is a total function $t : X \to \Delta$; we write $(A_1 : d_1, \ldots, A_n : d_n)$ to denote a tuple t over $A_1 \ldots A_n$ such that $t(A_i) = d_i$, for $1 \leq i \leq n$. For a relation scheme R, a *relation instance over* R is a finite set of tuples over $sort(R)$. For a scheme $\mathcal{S}$, a (*database*) *instance* $\mathcal{I}$ *over* $\mathcal{S}$ is a function mapping every relation name R in $\mathcal{S}$ to a relation instance over R. The *active domain* of an instance $\mathcal{I}$, denoted by $adom(\mathcal{I})$, is the set of all domain elements occurring in $\mathcal{I}$. We write $inst(\mathcal{S})$ to denote the set of all instances over a scheme $\mathcal{S}$.

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. For, assume the existence of a total order on $\mathcal{L}_a$, and use the convention of listing sets of attributes according to the total order. This way, if the listing of attributes $A_1 \ldots A_n$ respects the total order, then we write $(d_1, \ldots, d_n)$ to represent a tuple $(A_1 : d_1, \ldots, A_n : d_n)$.

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

For a scheme $\mathcal{S}$, we give to the set of instances over $\mathcal{S}$ the structure of a *complete partially ordered set* [18] with respect to $\subseteq$, by extending $inst(\mathcal{S})$ with a conventional 'infinite' instance $\top_{\mathcal{S}}$, in such a way that for any finite instance $\mathcal{I}$ over $\mathcal{S}$ it is the case that $\mathcal{I} \subseteq \top_{\mathcal{S}}$. This is especially useful in the context of r.e. queries, which can be partial functions, that is, possibly yielding as a result an *undefined instance*, i.e., the one we denoted $\top_{\mathcal{S}}$.

2.2 Queries and query languages

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

Let C be a finite set of constants, out of the domain Δ. A database mapping f is C-*generic* if $f \cdot \rho = \rho \cdot f$ for any permutation ρ over Δ (extended in the natural way to instances) that leaves C fixed (i.e., $\rho(x) = x$ for any $x \in C$). A database mapping is *generic* if it is C-generic for some finite C. A *query from* $\mathcal{S}$ *to* $\mathcal{T}$ is a generic database mapping $f : \mathcal{S} \to \mathcal{T}$.

The class $\mathcal{CQ}$ of *computable queries* [12] 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 $\mathcal{I}$, $adom(q(\mathcal{I})) \subseteq adom(\mathcal{I}) \cup C$. 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. Given query languages L_1, L_2, we say that L_1 *weaker than* L_2, denoted $L_1 \sqsubseteq L_2$, if every 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 C of queries, we say that L *expresses* C, denoted $L \equiv C$, if any query in C is expressible in L.

Finally, we say that 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. A database is *ordered with* min *and* max if it also contains 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.

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 previous works of the authors [23, 24]. We will not consider here the object-based characteristics of the language, which motivated its introduction; indeed, we focus in this paper only on the ability of the language to express queries in the relational setting.

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 $\mathcal{S}$ be fixed. Assume the existence of a countable set **Var** of *variables*, disjoint from the domain Δ.

A *term* is either a domain element $d \in \Delta$ or a variable $X \in$ **Var**. A *relation atom* is an expression of the form $R(A_1 : t_1, \ldots, A_n : t_n)$, where R is a relation with $sort(R) = A_1 \ldots A_n$, and $t_1, \ldots, t_n$ are terms. An *invention atom* is an expression of the form $R(\text{ID} : *, A_1 : t_1, \ldots, A_n : t_n)$, where R is a relation with $sort(R) = \text{ID}A_1 \ldots A_n$, ID is a distinguished attribute name called the *invention attribute*, '$*$' is a special symbol called the *invention symbol*, and $t_1, \ldots, t_n$ are terms. Intuitively, the invention symbol and invention atoms are used to create 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 *non-equality literal* and usually denoted $t_1 \neq t_2$. In the remainder of this work we will not consider equality atoms anymore (because we can resort to multiple occurrences of the same term, instead), whereas non-equality literals will be used.

Again, it is possible to adopt a positional notation by referring to a total order on $\mathcal{L}_a$ and omitting attribute names in literals. In particular, we assume that ID is the minimum element in $\mathcal{L}_a$ 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

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

where A is either a relation or an invention atom (called the *head of* γ and denoted *head*(γ)), and $L_1, \ldots, L_k$ (with $k \geq 0$) is a (possibly empty) finite set of literals (called the *body of* γ and denoted *body*(γ)). A clause is *range-restricted* if every variable occurring either in its head or in a negative literal in its body, occurs in a positive relation literal in its body as well. Hereinafter we will consider only range-restricted clauses. A *rule* is a clause with a non-empty body. A *fact* is a clause with an empty body (that is, an atom A); a fact is *ground* if no variable occurs in it. A clause is an *invention* (*relation*, resp.) clause if its head is an invention (relation, resp.) atom.

The relation name occurring in the head of an invention clause is called an *invention relation*. We assume the distinguished attribute name ID to be 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. An ILOG$^{\neg}$ program is *stratified* if it satisfies the *stratification condition* [8]. In the remainder of the work, unless explicitly stated, 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.

For a program $\mathcal{P}$, denote *adom*($\mathcal{P}$) the finite set of domain elements which explicitly occurr in $\mathcal{P}$, and *sch*($\mathcal{P}$) the database scheme made of the relation schemes occurring in $\mathcal{P}$. An *input-output scheme* (or, simply, *i-o scheme*) *for* $\mathcal{P}$ is a pair of schemes $(\mathcal{S}, \mathcal{T})$ such that *(i)* $\mathcal{S}$ and $\mathcal{T}$ are disjoint subsets of *sch*($\mathcal{P}$), called the *input* and *output* scheme, respectively; and *(ii)* no relation name in $\mathcal{S}$ occurs in the head of a clause in $\mathcal{P}$. For a program $\mathcal{P}$ over i-o scheme $(\mathcal{S}, \mathcal{T})$, denoted $(\mathcal{P}, \mathcal{S}, \mathcal{T})$, 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*($\mathcal{P}$) but neither in $\mathcal{S}$ nor in $\mathcal{T}$ are viewed as *temporary* relations.

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 behaviour 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. This way, value invention in programs corresponds essentially to a limited use of function symbols in logic programs. As a consequence, as noted by Hull and Yoshikawa [23], positive programs have a monotonic semantics, which is equivalently charaterized 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 [4]), where there exist positive programs defining non-monotone queries (see [23, Example 7.6]).

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

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

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

3. $Skol(\mathcal{P}) \cup \mathcal{I}$ is essentially a logic program with function symbols; a preferred model $\mathcal{M}_{Skol(\mathcal{P})\cup\mathcal{I}}$ of $Skol(\mathcal{P}) \cup \mathcal{I}$ (minimal if $\mathcal{P}$ 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 $\mathcal{M}_{Skol(\mathcal{P})\cup\mathcal{I}}$ is finite, call it the *model of* $\mathcal{P}$ *over* $\mathcal{I}$.

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

We now formalize concepts related to 'Skolem functor terms,' in order to complete the definition of the semantics of ILOG$^{(\neg)}$ programs.

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

Consider an invention relation $R(\text{ID}A_1 \ldots A_n)$, having f_R as the associated Skolem functor; a *Skolem functor term for* R is an expression of the form $f_R(A_1 : t_1, \ldots, A_n : t_n)$, (or simply $f_R(t_1, \ldots, t_n)$, adopting a positional notation), where $t_1 \ldots t_n$ are terms; note how the scheme constraints functor f_R to have arity $n = \alpha(R) - 1$. Then, extend the notion of *term* by considering Skolem functor terms also. The *Skolemization* of a program $\mathcal{P}$, denoted by $Skol(\mathcal{P})$, is obtained by replacing the head of each invention clause in $\mathcal{P}$ of the form $R(*, t_1, \ldots, t_n)$ by $R(f_R(t_1, \ldots, t_n), t_1, \ldots, t_n)$, where $f_R(t_1, \ldots, t_n)$ 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 relative to Skolem functor terms. Given a program $\mathcal{P}$, the *Herbrand universe* $\mathcal{U}_{\mathcal{P}}$ *for* $\mathcal{P}$ is the set of all ground terms built using domain elements from Δ and Skolem functors for invention relations in $sch(\mathcal{P})$. The *Herbrand base* $\mathcal{H}_{\mathcal{P}}$ *for* $\mathcal{P}$ is the set of all ground facts built using relation names in $sch(\mathcal{P})$ and terms in $\mathcal{U}_{\mathcal{P}}$. A *Herbrand interpretation over* $\mathcal{P}$ is a finite subset of $\mathcal{H}_{\mathcal{P}}$. Then, the notions of *Skolemized tuple, Skolemized relation instance,* and *Skolemized* (*database*) *instance over* $\mathcal{S}$ with respect to a program $\mathcal{P}$ are defined in the natural way, referring to the universe $\mathcal{U}_{\mathcal{P}}$ instead of the domain Δ. The set of all Skolemized database instances over a scheme $\mathcal{S}$ wrt a program $\mathcal{P}$ is denoted $S\text{-}inst_{\mathcal{P}}(\mathcal{S})$.

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

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 $\mathcal{P}$ over $\mathcal{I}$, then the semantics of $\mathcal{P}$ might include several possible outcomes (related to the choice of new values), and (by considering *all* possible replacements) it is in general a binary relation rather than a function.

Finally, we briefly 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 $\mathcal{I}$ and a ground literal L, the notion of *satisfaction* is defined in the usual way, and is extended in the natural way to sets of ground literals. For a set of clauses Γ over a scheme $\mathcal{S} = sch(\Gamma)$, the *immediate consequence operator* T_Γ *for* Γ is a mapping $T_\Gamma : S\text{-}inst_\Gamma(\mathcal{S}) \rightarrow S\text{-}inst_\Gamma(\mathcal{S})$ defined as follows:

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

3.3 Safe programs

We now introduce syntactical sublanguages of ILOG$^{(\neg)}$ that limit the use of 'invention' in programs; we follow analogous definitions in [4].

As we have seen, the semantics of an ILOG$^{(\neg)}$ program over an instance may lead to the introduction of new values, not in the active domain of the input database or 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 $(\mathcal{P}, \mathcal{S}, \mathcal{T})$ is *safe* if, for any instance $\mathcal{I}$ of $\mathcal{S}$, the semantics $\varphi_\mathcal{P}(\mathcal{I})$ 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 corresponds to stratified *datalog*$^{(\neg)}$.

Weak safety is defined relative to an i-o scheme, using the auxiliary notion of 'invention-attribute set.' Given a program $\mathcal{P}$ over i-o scheme $(\mathcal{S}, \mathcal{T})$, the *invention attributes for* $(\mathcal{P}, \mathcal{S}, \mathcal{T})$ are the smallest set of attributes such that:

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

A program $\mathcal{P}$ is *weakly safe wrt* $(\mathcal{S}, \mathcal{T})$ if no invention attribute for $(\mathcal{P}, \mathcal{S}, \mathcal{T})$ has the form (R, A), where R is a relation name in $\mathcal{T}$. 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 particular columns of the temporary relations in $sch(\mathcal{P})$, and not in target relations. In particular, 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 $(\mathcal{P}, \mathcal{S}, \mathcal{T})$, new values may appear in tuples in $sch(\mathcal{P}) - \mathcal{T}$ only. Because such 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. Indeed, the semantics of wILOG$^{(\neg)}$ programs coincide with their pre-semantics.

Note that weak safety of a program can be checked in polynomial time in the size of the program.

3.4 Introductory examples

The following 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 a program $\mathcal{P}_{code}$ that produces a representation of all the enumerations of a unary input relation R. Program $\mathcal{P}_{code}$ uses invention relations $list^{nil}(\text{ID})$ and $list^{cons}(\text{ID}, first, tail)$; values invented in these relations correspond to empty and non-empty 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 a variable Nil in programs to highlight terms that are intended to unify with values corresponding to an empty list.)

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

Consider for example the instance $\mathcal{I} = \{R(a), R(b)\}$; the pre-semantics for $\mathcal{P}_{code}$ on $\mathcal{I}$ contains in $list^{out}$ 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 relation R in $\mathcal{I}$.

Note that $\mathcal{P}_{code}$ is stratified, and 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 does not require the use of negation.

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

An ILOG$^{\neq}$ program $\mathcal{P}_{pcode}$ that computes all the partial enumerations of an input unary relation R can be obtained from program $\mathcal{P}_{code}$ of Example 3.1 by replacing its last clause by the following one:

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

□

The total enumerations of a set R can be built applying negation to the relation R only, provided a total order is given to the database. This is shown in the following example.

Example 3.3 Assume given an ordered database, with a unary relation R representing a finite subset of the active domain Δ, and the conventional unary relations min, max and a binary relation $succ$ representing a *total order* on Δ, such that min and max contains just the minimum and maximum element of Δ, respectively, and $succ$ the successor relation on the element of Δ according to the total order. The following program produces a representation of the total enumeration of R that respects the total order. Relation $Represents$ indicates whether a list contains all the elements of R up to a given one.

$$
\begin{array}{lcl}
list^{nil}(*) & \leftarrow & . \\
list^{cons}(*, \text{`]'}, Nil) & \leftarrow & list^{nil}(Nil). \\
Represents(RB, X) & \leftarrow & list^{cons}(RB, \text{`]'}, Nil), min(X), \neg R(X). \\
list^{cons}(*, X, RB) & \leftarrow & list^{cons}(RB, \text{`]'}, Nil), min(X), R(X). \\
Represents(LX, X) & \leftarrow & list^{cons}(RB, \text{`]'}, Nil), min(X), R(X), list^{cons}(LX, X, RB). \\
Represents(LX, Y) & \leftarrow & Represents(LX, X), succ(X, Y), \neg R(Y). \\
list^{cons}(*, Y, L) & \leftarrow & Represents(LX, X), succ(X, Y), R(Y). \\
Represents(LY, Y) & \leftarrow & Represents(LX, X), succ(X, Y), R(Y), list^{cons}(LY, Y, LX). \\
list^{out}(*, \text{`[', } LX) & \leftarrow & Represents(LX, X), max(X).
\end{array}
$$

Note that the foregoing program is semipositive (negation is applied to the input relation R only). The strategy of computing the enumeration consists in iterating on the elements of the domain (using the relations defining the total order) and taking different actions whether the elements belong to the set or not. Semipositive negation allows to continue the iteration in case an element is missing from R. □

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

Example 3.4 Consider relation $list^{out}$ which 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 $\mathcal{P}_{decode}$ computes a unary relation R containing the domain elements occurring in the input lists. It uses relation $toDecode(list)$ to select lists to be decomposed. Relation R will contain the *union* of what we obtain by decomponing each list in $list^{out}$.

$$
\begin{array}{lcl}
toDecode(L') & \leftarrow & list^{out}(L, \text{`['}, L'). \\
R(X) & \leftarrow & toDecode(L), list^{cons}(L, X, L'), X \neq \text{`]'}. \\
toDecode(L') & \leftarrow & toDecode(L), list^{cons}(L, X, L'), X \neq \text{`]'}.
\end{array}
$$

$\mathcal{P}_{decode}$ 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:

$$
\begin{array}{lcl}
toDecode(L') & \leftarrow & list^{out}(L, \text{`['}, L'). \\
R(X) & \leftarrow & toDecode(L), list^{cons}(L, X, L'), list^{cons}(L', X', L''). \\
toDecode(L') & \leftarrow & toDecode(L), list^{cons}(L, X, L'), list^{cons}(L', X', L'').
\end{array}
$$

□

3.5 A complex example

In this section we propose a program to solve instances of the 3SAT problem (an NP-complete problem).

An instance of 3SAT is a propositional formula in conjunctive normal form, made of three-literal clauses; it can be represented by a set $X = \{x_1 \ldots x_n\}$ of n variables and a collection C of m clauses, each containing exactly 3 literals, where a literal is a variable x_i or its negation $\neg x_i$. The problem asks whether there exists a truth assignment for the variables in X that satisfies all the clauses in C. For example, such an input might be the formula

$$(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_4) \wedge (x_3 \vee \neg x_1 \vee \neg x_4)$$

case in which the answer to the problem should be affirmative (e.g., consider the assignment ϕ which assigns *true* to x_1, x_3, and *false* to x_2, x_4).

We assume that the input for the program is encoded by means of a binary relation $var(variable, complement)$ and a ternary relation $clause(l_1, l_2, l_3)$. The former is used to represent the set X: for each variable $x_i \in X$, relation *var* will contain a pair $(x_i, \overline{x}_i)$, stating that x_i and $\overline{x}_i$ are literals (and, in particular, that $\overline{x}_i$ is the negation of x_i). Each clause in C will be represented by a tuple in *clause*; for instance, clause $(x_1 \vee x_2 \vee \neg x_3)$ will be represented by tuple $(x_1, x_2, \overline{x}_3)$.

On one hand, if all the truth assignments were part of the input, it would be easy to write a stratified *datalog*$^\neg$ program able to verify whether the formula is satisfiable. For, let $is\_true(assignment, literal)$ be a binary relation containing tuples of the form (a_i, l) stating that literal l is true in assignment a_i. The following program computes false literals for the various assignments, then assignments that do not satisfy the formula, and finally satisfiability of the formula.

$$\begin{array}{lcl}
is_false(A, L') & \leftarrow & is_true(A, L), var(L, L'). \\
is_false(A, L') & \leftarrow & is_true(A, L), var(L', L). \\
unsat(A) & \leftarrow & clause(X, Y, Z), is_false(A, X), is_false(A, Y), is_false(A, Z). \\
sat & \leftarrow & assignment(A), \neg unsat(A).
\end{array}$$

The above program is correct only on input instances such that relation *is_true* indeed defines all possible assignments. However, it is clear that *datalog*$^\neg$ is not able to compute all such possible truth assignments. Indeed, there are exponentially many such assignments, and *datalog*$^\neg$ can only perform PTIME computations. We now write a WILOG$^\neg$ program that will do such computation; it uses the naive strategy that generates *all* assignments. Note that every assignment must assign value *true* to either literal x_i or literal $\overline{x}_i$, for $1 \leq i \leq n$. The construction of assignments is inspired by Example 3.1.

$$
\begin{array}{lcl}
assign^{nil}(*) & \leftarrow & . \\
misses(Nil, X) & \leftarrow & assign^{nil}(Nil), var(X, X'). \\
assign^{cons}(*, X, A) & \leftarrow & misses(A, X). \\
assign^{cons}(*, X', A) & \leftarrow & misses(A, X), var(X, X'). \\
misses(A, Y) & \leftarrow & assign^{cons}(A, X, A'), misses(A', Y), var(X, X'), X \neq Y. \\
misses(A, Y) & \leftarrow & assign^{cons}(A, X, A'), misses(A', Y), var(X', X), X' \neq Y. \\
misses^{proj}(A) & \leftarrow & misses(A, X). \\
is\_true(A, X) & \leftarrow & assign^{cons}(A, X, A'). \\
is\_true(A, X) & \leftarrow & assign^{cons}(A, Y, A'), is\_true(A', X). \\
assignment(A) & \leftarrow & assign^{cons}(A, X, A'), \neg misses^{proj}(A).
\end{array}
$$

The overall program for 3SAT is stratified and made of two strata (the second one containing clauses defining relations *assignment* and *sat*). It can be verified that this program, on instances satisfying the above hypotheses, answers *sat* if and only if the input instance encodes a satisfiable formula.

3.6 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.

Since strongly safe ILOG$^{(\neg)}$ corresponds syntactically to stratified *datalog*$^{(\neg)}$, it inherits a lot of well-known results. Among others, we recall that sILOG$^\neg$ expresses (total) queries in PTIME. Furthermore, it expresses the *fixpoint queries* if we drop the requirement of stratification and adopt the inflationary semantics for negation (Abiteboul and Vianu [4]). A result by Kolaitis [29] shows that the stratified semantics for sILOG$^\neg$ is weaker than the inflationary one. Finally, the language *datalog*$^{\frac{1}{2},\neg}$ expresses the PTIME queries on ordered databases with *min* and *max* (Papadimitriou [34]).

On the other hand, wILOG$^{(\neg)}$ allows for value invention in temporary relations only, in such a way that the semantics of any weakly safe program is always a query. The following result can be proved as the analogous result stated for weakly safe *datalog*$^\neg_\infty$ programs in [4], even with a different semantics for negation and value invention.

Fact 3.5 *Let $\mathcal{P}$ be a* wILOG$^\neg$ *program over i-o scheme $(\mathcal{S}, \mathcal{T})$. Then, the semantics of $(\mathcal{P}, \mathcal{S}, \mathcal{T})$ is a C-generic database mapping from $\mathcal{S}$ to $\mathcal{T}$, with $C = adom(\mathcal{P})$.*

In presence of value invention or a similar construct, it has been shown [20, 22] 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 proved as the analogous result about COL [22], a deductive language with untyped sets. There, hereditarely finite set construction is used instead of value invention; furthermore, COL programs have to be stratified also wrt set construction.

Fact 3.6 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, relative to a limited use of stratified negation. We then strengthen the result stated as Fact 3.6 by showing that wILOG$^\neg$ programs made of two strata have 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 made of *at most* $i+1$ strata, i.e, programs which use i 'groups' of negations. At the lowest levels of the hierarchy, we find languages with a very limited use of negative literals: ILOG$^\neq$ is the class of programs with no negation, but still allowing for non-equality literals; ILOG is the class of positive programs, i.e., with no negative literals at all; finally, ILOG$^{\frac{1}{2},\neg}$ is the language of *semipositive* programs, i.e., programs in which negation can be applied on input relations only. Analogous hierarchies are defined with respect to languages sILOG$^\neg$ and wILOG$^\neg$.

With respect to the expressive power, we have the following intuitive hierarchy, based upon syntactical considerations:

$$\text{ILOG} \sqsubseteq \text{ILOG}^\neq \sqsubseteq \text{ILOG}^{\frac{1}{2},\neg} \sqsubseteq \text{ILOG}^{1,\neg} \sqsubseteq \ldots \sqsubseteq \text{ILOG}^\neg$$

As shown by Kolaitis [29], the above hierarchy 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 a following section shows that the simpler language wILOG$^{\frac{1}{2},\neg}$ 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 3.6) we highlight the crucial points.

Consider a query q; since q is computable, there is an effective algorithm for its implementation. We refer here to *domain Turing Machines* (*domTMs*), introduced by Hull and Su [21]; for the sake of completeness, a brief introduction to domTMs is in the Appendix. 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 likes parentheses '(' and ')' and brackets '[' and ']'. Moreover, a domTM is equipped with a *register*, capable of storing a symbol of the alphabet, whose use allows to keep finite the control of the device, even with a countable alphabet.

Given an instance $\mathcal{I}$, an *enumeration of* $\mathcal{I}$ is a sequential representation of $\mathcal{I}$ on a domTM tape (where domain elements are separated by connectives in W, enclosing tuples 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 $enum(\mathcal{I})$ the set of all enumerations of an instance $\mathcal{I}$. For example, if $\mathcal{I}$ is the instance $\{R_1(a), R_2(a,b), R_2(b,c)\}$

over $\{R_1, R_2\}$, then the enumerations of $\mathcal{I}$ (assuming the listing of R_1 precedes that of R_2) are $e_1 =$ `[(a)][(ab)(bc)]` and $e_2 =$ `[(a)][(bc)(ab)]`.

A result by Hull and Su [21] (Fact A.1 in the Appendix) states that, for any computable query, there exists an *order independent* domTM that computes the query; hence, there exists an order independent domTM M_q which computes q. Thus, given an input instance $\mathcal{I}$, either M_q does not halt on any enumeration of $\mathcal{I}$ (meaning that q is undefined on input $\mathcal{I}$), or there exists an instance $\mathcal{J}$ such that, for any enumeration e of $\mathcal{I}$, the computation of M_q on e, denoted by $M_q(e)$, halts resulting in an output that is an enumeration of $\mathcal{J}$ (meaning that $q(\mathcal{I}) = \mathcal{J}$). For example, if $M_q(e_1) =$ `[(c)(b)]` and $M_q(e_2) =$ `[(b)(c)]`, we assume $q(\mathcal{I}) = \{(b), (c)\}$.

Computations of a domTM M_q can be simulated as follows:

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

2. Simulate the computation $M_q(e)$ for any enumeration $e \in enum(\mathcal{I})$; the various simulations are performed simultaneously and eventually result in an output enumeration for every enumeration of $\mathcal{I}$.

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 which computes the following query (on input instance $\mathcal{I}$):

$$\varphi_Q(\mathcal{I}) = \bigcup_{e \in enum(\mathcal{I})} decode(M_q(e)).$$

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

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 because in general we have to deal also 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 (i.e., without negation). Invented values are used to represent *strings* stored on the tape of the domTM. Termination of a computation happens with a finite number of acrossed global configurations (which are represented by a finite number of strings), whereas a non-termination involves an infinite number of acrossed configurations. Therefore, termination and non-termination 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 (i.e., positive) program. The technique used is that of Example 3.4; intuitively, we perform the *union* of the decoding of the various output enumerations.

4. The overall program, obtained by putting together the above three subprograms, is in wILOG$^{1,\neg}$, that is, it is weakly safe and 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 $\mathcal{P}_{code}$ of Example 3.1; there, $\mathcal{P}_{code}$ is made of two strata: the output of the first stratum (which contains only ILOG$^{\neq}$ clauses) contains all partial enumerations of the input relation R; $\mathcal{P}_{code}$ 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 enumerations of the input instance, the 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 $\mathcal{S}$ to $\mathcal{T}$, and M_q an order independent domTM which implements q. We now define three programs Q_{in}, Q_{simu}, and Q_{out} based on M_q, and then we show that program $Q = Q_{in} \cup Q_{simu} \cup Q_{out}$ is in wILOG$^{1,\neg}$ and indeed simulates 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 $\mathcal{S} = \{R_1, \ldots, R_n\}$. To this end, we first encode each input relation R_i in $\mathcal{S}$ independently, and then concatenate them.

For the first part, assume we have to enumerate a binary relation R_i. We use invention relations $ENC_i^{nil}(\text{ID})$ (for the empty string), $ENC_i^{*}(\text{ID}, \mathit{first}, \mathit{tail})$ (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).

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

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 relation R_i. We will use it to concatenate enumerations of the various relations.

$$
\begin{array}{lcl}
ENC_i^{cons}(S, C, T) & \leftarrow & ENC_i^{*}(S, C, T). \\
ENC_i^{cons}(S, C, T) & \leftarrow & ENC_i^{)}(S, C, T). \\
ENC_i^{cons}(S, C, T) & \leftarrow & ENC_i^{2}(S, C, T). \\
ENC_i^{cons}(S, C, T) & \leftarrow & ENC_i^{1}(S, C, T). \\
ENC_i^{cons}(S, C, T) & \leftarrow & ENC_i(S, C, T).
\end{array}
$$

Note how the foregoing set of clauses can be easily adapted to work with any input relation in $\mathcal{S}$, where a different number of invention relations have to 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 clauses are positive.

$$
\begin{array}{lcl}
ENC^{nil}(*) & \leftarrow & . \\
represents(Nil, Nil') & \leftarrow & ENC^{nil}(Nil), ENC_n^{nil}(Nil'). \\
ENC^{cons}(*, C, S) & \leftarrow & ENC_n^{cons}(L, C, S'), represents(S, S'). \\
represents(S, S') & \leftarrow & ENC^{cons}(S, C, T), ENC_n^{cons}(S', C, T'), represents(T, T'). \\
represents(S, Nil) & \leftarrow & ENC_{n-1}^{nil}(Nil), ENC_n(S', C, T), represents(S, S'). \\
ENC^{cons}(*, C, S) & \leftarrow & ENC_{n-1}^{cons}(L, C, S'), represents(S, S'). \\
represents(S, S') & \leftarrow & ENC^{cons}(S, C, T), ENC_{n-1}^{cons}(S', C, T'), represents(T, T'). \\
\ldots & \leftarrow & \ldots \\
ENC(S) & \leftarrow & ENC^{cons}(S, C, T), ENC_1(S', C', T'), represents(S, S').
\end{array}
$$

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

Simulation of 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.

Instantaneous descriptions (a.k.a. global configurations) of the domTM are represented by means of invention relations ID^{nil} and ID^{cons}: the former stores starting configurations and the latter successive ones. The scheme of ID^{nil} is (ID, *state*, *register*, *ltape*, *head*, *rtape*), and that of ID^{cons} includes also 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 configurations, and thus an ordinary relation would contain a finite number of instantaneous descriptions, giving 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 acrossed configuration, 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 values associated with all acrossed instantaneous descriptions. Relations ID^{nil} and ID are defined as follows (q_s is the starting state of the domTM):

$$
\begin{array}{lcl}
ID^{nil}(*, q_s, \text{`\#'}, Nil, \text{`\#'}, X) & \leftarrow & ENC(X), ENC^{nil}(Nil). \\
ID(I, S, N, L, H, R) & \leftarrow & ID^{nil}(I, S, N, L, H, R). \\
ID(I, S, N, L, H, R) & \leftarrow & ID^{cons}(I, S, N, L, H, R, P).
\end{array}
$$

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 would reach an end of the tape.

$$
\begin{array}{lcl}
first(X, F) & \leftarrow & ENC^{cons}(X, F, T). \\
first(Nil, \text{`\#'}) & \leftarrow & ENC^{nil}(Nil). \\
tail(X, T) & \leftarrow & ENC^{cons}(X, F, T). \\
tail(Nil, Nil) & \leftarrow & ENC^{nil}(Nil).
\end{array}
$$

We now describe how to manage the transition function δ.

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

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

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

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

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

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

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

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

Similarly for other types of moves.

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

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

During the computation of the least fixpoint of the foregoing set of clauses, simulations associated with different input enumerations evolve independently. Looking at the simulation relative to a single input enumeration, at each stage of the fixpoint computation all previous IDs 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 simulations reach the halting state.

Because M_q implements q, the output tapes of halting computations of M_q correspond to enumerations of the output instance (i.e., 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 (q_h is the unique halting state of the domTM):

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

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

We decode strings corresponding to enumerations of the output instance, that we stored in relation ENC^{out}. For example, assume the target relation T is binary; as in Example 3.4, we use an auxiliary relation $toDecode$:

$$\begin{array}{lll} toDecode(S') & \leftarrow & ENC^{out}(S), ENC^{cons}(S, \text{`[', } S'). \\ T(X_1, X_2) & \leftarrow & toDecode(S), ENC^{cons}(S, \text{`(', } S_1), \\ & & ENC^{cons}(S_1, X_1, S_2), ENC^{cons}(S_2, X_2, S_3). \\ toDecode(S') & \leftarrow & toDecode(S), ENC^{cons}(S, \text{`(', } S_1), \\ & & ENC^{cons}(S_1, X_1, S_2), ENC^{cons}(S_2, X_2, S_3), \\ & & ENC^{cons}(S_3, \text{`)', } S'). \end{array}$$

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 $\mathcal{S}$ to $\mathcal{T}$.

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

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

1. For any instance $\mathcal{I}$ over $\mathcal{S}$, Q has finite model over $\mathcal{I}$ if and only if q is defined over $\mathcal{I}$.

2. For any instance $\mathcal{I}$ over $\mathcal{S}$ such that $q(\mathcal{I})$ is defined, $Q(\mathcal{I}) = q(\mathcal{I})$.

To prove Statement 1, consider an instance $\mathcal{I}$. If q is defined over $\mathcal{I}$, then M_q halts on every enumeration of $\mathcal{I}$. Denote $|\mathcal{I}|$ the number of symbols used to represent $\mathcal{I}$ (i.e., the length of any of its enumerations) and, for an enumeration e of $\mathcal{I}$, denote $\Omega_q(e)$ 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 $\mathcal{I}$ is $O\big(\Sigma_{e \in enum(\mathcal{I})}(\Omega_q(e) + |\mathcal{I}|)\big)$ (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 $adom(\mathcal{I})$ and the number of invented values) is finite, that is, the model of Q over $\mathcal{I}$ is finite. Similarly, it can be proven that the model of Q over $\mathcal{I}$ can be computed (according to a fixpoint semantics) in $O\big(Max_{e \in enum(\mathcal{I})}(\Omega_q(e)) + |\mathcal{I}|\big)$ stages.

On the other hand, if q is undefined on input $\mathcal{I}$, then M_q does not halt on any enumeration of $\mathcal{I}$. 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 non-equalities. Since it disallows other forms of negation, we do not expect this language to express nonmonotonic queries. Interestingly, we show that the language is able to express *all* the monotone queries; before stating formally the result, we need to discuss the notion of monotonicity in a framework allowing for r.e. queries (i.e., partial queries as well).

The notion of monotonicity is as follows (see, among others, [18]). A query $q : \mathcal{S} \to \mathcal{T}$ is *monotone* if, for any pair of instances $\mathcal{I}, \mathcal{J}$ over $\mathcal{S}$, $\mathcal{I} \subseteq \mathcal{J}$ implies $q(\mathcal{I}) \subseteq q(\mathcal{J})$. 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.

It is well-known [5] that *datalog*$^{\neq}$ expresses only monotone PTIME queries (and thus, total monotone queries). Hence, the same holds for sILOG$^{\neq}$.

In the context of r.e. queries, i.e., queries which can be partial functions, we must consider the case in which the result of a query is undefined over an input instance. A (partial) query $q : \mathcal{S} \to \mathcal{T}$ is *downward defined* if, for any pair of instances $\mathcal{I}, \mathcal{J}$ over $\mathcal{S}$, $\mathcal{I} \subseteq \mathcal{J}$ and q defined over $\mathcal{J}$ implies that q is defined over $\mathcal{I}$. It is clear that any monotone query q is downward defined.

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* (i.e., the non-empty 0-ary relation $\{()\}$) if G is planar and contains a Hamiltonian circuit, answers *false* (i.e., $\{\}$) 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 (this problem being NP-complete [17]), and does not halt on non-planar graphs.

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

Lemma 5.1 *Let $\mathcal{P}$ be a* wILOG$^{\neq}$ *program. Then, the semantics of $\mathcal{P}$ is a monotone query.*

Proof: Consider the immediate consequence operator $T_{\mathcal{P}}$ for $\mathcal{P}$, and instances $\mathcal{I} \subseteq \mathcal{J}$.

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

To prove monotonicity, assume that the fixpoint computation converges over instance $\mathcal{J}$, as $T^\omega_{\mathcal{P}}(\mathcal{J})$. It then clearly converges over $\mathcal{I}$ as well, and $T^\omega_{\mathcal{P}}(\mathcal{I}) \subseteq T^\omega_{\mathcal{P}}(\mathcal{J})$. ■

The above is a typical result for database programming languages: Lemma 5.1 states that all the queries which are expressible in a syntactically defined language also satisfy a semantically defined property. 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 among 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 which implements q. For an input instance $\mathcal{I}$, to evaluate $q(\mathcal{I})$ we would like to simulate a computation of M_q on an enumeration e of $\mathcal{I}$. Because of genericity, we are forced to consider computations of M_q on all enumerations of $\mathcal{I}$ rather than on a single one, as in the proof of Theorem 4.1. However, computing the enumerations of an instance is not a monotonic operation; thus we must slightly modify our evaluation strategy.

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

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

(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

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

From downward definition of q, it follows that $\widetilde{Q}$ is defined on input $\mathcal{I}$ if and only if q is; from its monotonicity, we have $q(\mathcal{J}) \subseteq q(\mathcal{I})$ for any $\mathcal{J} \subseteq \mathcal{I}$. In turn, $\widetilde{Q}(\mathcal{I}) = q(\mathcal{I})$.

Hence, to prove the theorem, it suffices to show that the evaluation strategy $\widetilde{Q}$ for q can be implemented in WILOG$^{\neq}$. For, 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 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} do not make use of negation. Modify program Q_{in} by removing its negated literals, thus defining a new program $\widetilde{Q}_{in}$; it is apparent that now $\widetilde{Q}_{in}$ belongs to ILOG$^{\neq}$. Program $\widetilde{Q}_{in}$ is obtained from Q_{in} as program $\mathcal{P}_{pcode}$ of Example 3.2 is obtained from program $\mathcal{P}_{code}$ of Example 3.1. It can be shown that $\widetilde{Q}_{in}$, on input $\mathcal{I}$, indeed generates in relation ENC the representatives of all partial enumerations of $\mathcal{I}$.

Consider now program $\widetilde{Q} = \widetilde{Q}_{in} \cup Q_{simu} \cup Q_{out}$; 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 $\widetilde{Q}$ coincides with q. ■

6 Expressiveness of semipositive programs

In this section we study the expressive power of the language WILOG$^{\frac{1}{2},\neg}$ of semipositive programs, that is, the class of programs in which negation can be applied to input relations only. This language is strictly more expressive than WILOG$^{\neq}$; indeed, WILOG$^{\frac{1}{2},\neg}$ allows to express non-monotone queries as well, e.g., 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 queries defined by WILOG$^{\frac{1}{2},\neg}$ programs satisfy a weak form of monotonicity; we call 'semimonotone' these queries. Then, we strengthen the result by proving that WILOG$^{\frac{1}{2},\neg}$ expresses exactly this class of queries.

We need a few preliminary definitions.

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

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

Given a scheme $\mathcal{S} = \{R_1, \ldots, R_n\}$ and instances $\mathcal{I}, \mathcal{J}$ over $\mathcal{S}$, we say that $\mathcal{J}$ is an *extension of* $\mathcal{I}$ (written $\mathcal{I} \overset{ext}{\sqsubseteq} \mathcal{J}$) if $adom(\mathcal{I}) \subseteq adom(\mathcal{J})$ and, for $1 \leq i \leq n$, $\mathcal{I}(R_i) =$

$\mathcal{J}(R_i)|_{adom(\mathcal{I})}$, that is, the restriction of relation $\mathcal{J}(R_i)$ to the active domain of instance $\mathcal{I}$ coincides with $\mathcal{I}(R_i)$.

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

Proof: Assume that $\mathcal{J}$ is an extension of $\mathcal{I}$, that is, *(i)* $adom(\mathcal{I}) \subseteq adom(\mathcal{J})$ and, for any relation R in $\mathcal{S}$, *(ii)* $\mathcal{I}(R) = \mathcal{J}(R)|_{adom(\mathcal{I})}$. We now prove that, for any relation R in $\mathcal{S}$, *(iii)* $\overline{\mathcal{I}}(R) \subseteq \overline{\mathcal{J}}(R)$ and *(iv)* $\overline{\mathcal{I}}(\overline{R}) \subseteq \overline{\mathcal{J}}(\overline{R})$.

Inclusion *(iii)* follows directly from *(ii)*:

$$\overline{\mathcal{I}}(R) = \mathcal{I}(R) \subseteq \mathcal{J}(R) = \overline{\mathcal{J}}(R).$$

For inclusion *(iv)*, we have:

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

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:

$$\begin{aligned}
\mathcal{J}(R)|_{adom(\mathcal{I})} &= \left(adom(\mathcal{J})^{\alpha(R)} - \overline{\mathcal{J}}(\overline{R})\right)|_{adom(\mathcal{I})} \\
&= adom(\mathcal{I})^{\alpha(R)} - \overline{\mathcal{J}}(\overline{R})|_{adom(\mathcal{I})} \\
&\subseteq adom(\mathcal{I})^{\alpha(R)} - \overline{\mathcal{I}}(\overline{R}) = \mathcal{I}(R)
\end{aligned}$$

whereas $\mathcal{I}(R) \subseteq \mathcal{J}(R)|_{adom(\mathcal{I})}$ follows from *(iii)*. ■

A query $q : \mathcal{S} \to \mathcal{T}$ is *preserved under extensions* [5] if, for any pair of instances $\mathcal{I}, \mathcal{J}$ over $\mathcal{S}$, whenever $\mathcal{J}$ is an extension of $\mathcal{I}$, it is the case that $q(\mathcal{I}) \subseteq q(\mathcal{J})$. 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.

A (partial) query $q : \mathcal{S} \to \mathcal{T}$ is $\overset{ext}{\sqsubseteq}$*-defined* if, for any pair of instances $\mathcal{I}, \mathcal{J}$ over $\mathcal{S}$, $\mathcal{I} \overset{ext}{\sqsubseteq} \mathcal{J}$ and q defined over $\mathcal{J}$ imply that q is defined over $\mathcal{I}$. Note that $\mathcal{I} \overset{ext}{\sqsubseteq} \mathcal{J}$ implies $\mathcal{I} \subseteq \mathcal{J}$, but the converse does not hold in general; similarly, downward-definedness implies $\overset{ext}{\sqsubseteq}$-definedness, but the converse is not always implied.

We observe that any query preserved under extensions is $\overset{ext}{\sqsubseteq}$-defined as well.

Preservation under extensions is a weak form of monotonicity. In what follows, we shall however use a different terminology for this property, by calling *semimonotone* any 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.

Lemma 6.2 *Let $\mathcal{P}$ be a semipositive* WILOG$^{\frac{1}{2},\neg}$ *program. Then, the semantics of $\mathcal{P}$ is a semimonotone query.*

Proof: Consider the positivization $\overline{\mathcal{P}}$ of $\mathcal{P}$ obtained by replacing negative literals. Let $\mathcal{I}, \mathcal{J}$ be instances over the input scheme of $\mathcal{P}$ such that $\mathcal{J}$ is an extension of $\mathcal{I}$, and $\overline{\mathcal{I}}, \overline{\mathcal{J}}$ the corresponding enriched instances. By Lemma 5.1, the semantics of $\overline{\mathcal{P}}$ is a monotone query. Assume $\mathcal{P}$ defined on input $\mathcal{J}$; then, so it is $\overline{\mathcal{P}}$ on input $\overline{\mathcal{J}}$. Now, $\overline{\mathcal{I}} \subseteq \overline{\mathcal{J}}$, and by downward definition of $\overline{\mathcal{P}}$, the latter is defined on input $\overline{\mathcal{I}}$; moreover, because of its monotonicity, $\overline{\mathcal{P}}(\overline{\mathcal{I}}) \subseteq \overline{\mathcal{P}}(\overline{\mathcal{J}})$. Hence, $\mathcal{P}$ is defined on input $\mathcal{I}$ and $\mathcal{P}(\mathcal{I}) \subseteq \mathcal{P}(\mathcal{J})$. ■

Lemma 6.3 *The query that computes the complement of the transitive closure (CTC) of a binary relation is not semimonotone.*

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

As a consequence of the above lemmata, CTC is not expressible in $\text{wILOG}^{\frac{1}{2},\neg}$, that is, we have a query separating the class of semipositive programs from the computable queries.

Corollary 6.4 $\text{wILOG}^{\neq} \sqsubset \text{wILOG}^{\frac{1}{2},\neg} \sqsubset \text{wILOG}^{1,\neg}$.

Other queries belong to $\text{wILOG}^{1,\neg} - \text{wILOG}^{\frac{1}{2},\neg}$. 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

$$\begin{aligned} min(succ) &= \{x \mid \nexists w : succ(w, x)\} \\ max(succ) &= \{x \mid \nexists w : succ(x, w)\} \end{aligned}$$

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 $\text{wILOG}^{\frac{1}{2},\neg}$ expresses the semimonotone queries, or only part of them. The remainder of the section is devoted to show that $\text{wILOG}^{\frac{1}{2},\neg}$ indeed expresses this class of queries.

Theorem 6.5 $\text{wILOG}^{\frac{1}{2},\neg}$ *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 which implements q. For an input instance $\mathcal{I}$, our evaluation strategy can neither consider computation of M_q on an enumeration e of $\mathcal{I}$ (because of genericity) nor computations on all total enumerations of $\mathcal{I}$ (because the operation of computing the set $enum(\mathcal{I})$ is not preserved under extensions). Is there any suitable set of enumerations derivable from $\mathcal{I}$ such that: *(i)* this set is expressible by means of a semipositive program; and *(ii)* the union of results of computations of M_q on this set yields $q(\mathcal{I})$? Fortunately, the answer

is affirmative. To prove formally the result, we need some preliminary considerations and definitions.

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

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

$$\{\mathcal{I}|_D \mid D \subseteq adom(\mathcal{I})\} = \{\mathcal{J} \mid \mathcal{J} \overset{ext}{\sqsubseteq} \mathcal{I}\}.$$

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

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

If we simulate computations of M_q on this set, taking the union of the results, we in turn evaluate the following query:

$$\begin{aligned}
\widehat{Q}(\mathcal{I}) &= \bigcup_{e \in r\text{-}enum(\mathcal{I})} decode(M_q(e)) \\
&= \bigcup_{D \subseteq adom(\mathcal{I})} \bigcup_{e \in enum(\mathcal{I}|_D)} decode(M_q(e)) \\
&= \bigcup_{D \subseteq adom(\mathcal{I})} q(\mathcal{I}|_D) = \bigcup_{\mathcal{J} \overset{ext}{\sqsubseteq} \mathcal{I}} q(\mathcal{J}).
\end{aligned}$$

Because q is $\overset{ext}{\sqsubseteq}$-defined, $\widehat{Q}$ is defined on input $\mathcal{I}$ whenever q is; because of its semimonotonicity, $q(\mathcal{J}) \subseteq q(\mathcal{I})$ for any $\mathcal{J} \overset{ext}{\sqsubseteq} \mathcal{I}$, hence $\widehat{Q}(\mathcal{I}) = q(\mathcal{I})$.

We now define an ILOG$^{\frac{1}{2},\neg}$ program $\widehat{Q}_{in}$ doing the following. 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 $adom(\mathcal{I})$; 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 encoding of enumerations of the input relations; we do so as in Example 3.3, using semipositive negation in this phase only. 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$^{\frac{1}{2},\neg}$ program $\widehat{Q}$ is defined by putting $\widehat{Q}_{in}$ 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 $\widehat{Q}_{in}$.

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

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

We use invention relations $enc^{nil}(\text{ID})$, $enc^{cons}(\text{ID}, first, tail)$ and 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):

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

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 $(\text{ID}, enum)$ 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.

$$\begin{array}{lcl}
ENC_i^{nil}(*, P) & \leftarrow & enc(P). \\
ENC_i^{*}(*, \text{`]'}, Nil, P) & \leftarrow & ENC_i^{nil}(Nil, P). \\
ENC_i^{)}(*, \text{`)'}, S, P) & \leftarrow & toAppend_i(S, P, X_1, X_2). \\
ENC_i^{2}(*, X_2, S_1, P) & \leftarrow & ENC_i^{)}(S_1, \text{`)'}, S, P), toAppend_i(S, P, X_1, X_2). \\
ENC_i^{1}(*, X_1, S_2, P) & \leftarrow & ENC_i^{2}(S_2, X_2, S_1, P), ENC_i^{)}(S_1, \text{`)'}, S, P), \\
 & & toAppend_i(S, P, X_1, X_2). \\
ENC_i^{*}(*, \text{`('}, S_3, P) & \leftarrow & ENC_i^{1}(S_3, X_1, S_2, P), ENC_i^{2}(S_2, X_2, S_1, P), \\
 & & ENC_i^{)}(S_1, \text{`)'}, S, P), toAppend_i(S, P, X_1, X_2).
\end{array}$$

$$
\begin{array}{lcl}
Represents_i(S', P, X_1, X_2) & \leftarrow & ENC_i^*(S, \text{`(`}, S_3, P), ENC_i^1(S_3, X_1, S_2, P), \\
 & & ENC_i^2(S_2, X_2, S_1, P), ENC_i^{)}(S_1, \text{`)`}, S, P), \\
 & & toAppend_i(S, P, X_1, X_2). \\
toAppend_i(S, P, X, X) & \leftarrow & ENC_i^*(S, \text{`]`}, Nil, P), ENC_i^{nil}(Nil, P), \\
 & & min(P, X), R_i(X, X). \\
Represents_i(S, P, X, X) & \leftarrow & ENC_i^*(S, \text{`]`}, Nil, P), ENC_i^{nil}(Nil, P), \\
 & & min(P, X), \neg R_i(X, X). \\
toAppend_i(S, P, X_1, X_2') & \leftarrow & Represents_i(S, P, X_1, X_2), succ(P, X_2, X_2'), \\
 & & R_i(X_1, X_2'). \\
toAppend_i(S, P, X_1', X_2') & \leftarrow & Represents_i(S, P, X_1, X_2), max(P, X_2), min(P, X_2'), \\
 & & succ(P, X_1, X_1'), R_i(X_1', X_2'). \\
Represents_i(S, P, X_1, X_2') & \leftarrow & Represents_i(S, P, X_1, X_2), succ(P, X_2, X_2'), \\
 & & \neg R_i(X_1, X_2'). \\
Represents_i(S, P, X_1', X_2') & \leftarrow & Represents_i(S, P, X_1, X_2), max(P, X_2), min(P, X_2'), \\
 & & succ(P, X_1, X_1'), \neg R_i(X_1', X_2'). \\
ENC_i(*, \text{`[`}, S, P) & \leftarrow & Represents_i(S, P, X, X), max(P, X). \\
ENC_i(*, \text{`[`}, S, Nil') & \leftarrow & ENC_i^*(S, \text{`]`}, Nil, Nil'), ENC_i^{nil}(Nil, Nil'), enc^{nil}(Nil').
\end{array}
$$

The foregoing set of clauses, written to encode a binary relation, can be modified so as to build encodings of any input relation in $\mathcal{S}$, 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 enumerations:

$$
\begin{array}{lcl}
ENC_i^{cons}(S, C, T, P) & \leftarrow & ENC_i^*(S, C, T, P). \\
ENC_i^{cons}(S, C, T, P) & \leftarrow & ENC_i^{)}(S, C, T, P). \\
ENC_i^{cons}(S, C, T, P) & \leftarrow & ENC_i^2(S, C, T, P). \\
ENC_i^{cons}(S, C, T, P) & \leftarrow & ENC_i^1(S, C, T, P). \\
ENC_i^{cons}(S, C, T, P) & \leftarrow & ENC_i(S, C, T, P).
\end{array}
$$

We now concatenate encodings of enumerations of the various input relations. To obtain enumerations of instances for which $\mathcal{I}$ 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).

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

■

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

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 only semipositive negation). 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$^{\frac{1}{2},\neg}$ 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$^{\frac{1}{2},\neg}$ expresses the computable queries provided a total order is given.

Corollary 6.6 WILOG$^{\frac{1}{2},\neg}$ *expresses the computable queries on ordered databases with* min *and* max.

7 Towards strongly monotone queries?

All the results proved in this paper refer essentially to simulations of computations of domain Turing machines made by suitable programs in subclasses of WILOG$^{(\neg)}$. In all the cases, the difficult part of the proof concerned the ability of the language to enumerate the input instance as a (sort of a) 'string' 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$^{\frac{1}{2},\neg}$ can build enumerations of instances for which the input instance is an extension ($\overset{ext}{\sqsubseteq}$). Then, simulations of domTMs can be carried over these enumerations without resorting to negation anymore (non-equality is required, however); finally, results of computations can be decoded with no negation and no non-equality.

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 [30, 5] 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 $\mathcal{I} \overset{h}{\to} \mathcal{J}$ such a homomorphism. A query $q : \mathcal{S} \to \mathcal{T}$ is *strongly monotone* if it is preserved under homomorphisms, that is, for any pair of instances $\mathcal{I}, \mathcal{J}$ over $\mathcal{S}$, $\mathcal{I} \overset{h}{\to} \mathcal{J}$ 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.

Strongly monotone partial queries satisfy a definedness property, as follows. A (partial) query $q : \mathcal{S} \to \mathcal{T}$ is *$\overset{h}{\to}$-defined* if, for any pair of instances $\mathcal{I}, \mathcal{J}$ over $\mathcal{S}$, q defined over $\mathcal{J}$ and $\mathcal{I} \overset{h}{\to} \mathcal{J}$ imply that Q is defined over $\mathcal{I}$. It turns out that any strongly monotone query is $\overset{h}{\to}$-defined.

Now, it is easy to show that the semantics of any wILOG program is a strongly monotone query (again, by reasoning on its fixpoint semantics). However, in this case 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 of all, the approach of resorting to domain Turing machines as an effective way to implement a query can not 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 expressing the strongly monotone queries); to simulate these transition values, non-equality literals must be used, and we do not have them in wILOG.

Second, observe that for any (finite) input instance $\mathcal{I}$, both the set of instances contained in $\mathcal{I}$ and the set of instances for which $\mathcal{I}$ is an extension are finite; this is in contrast with the fact that the set of instances $\{\mathcal{J} \mid \mathcal{J} \overset{h}{\to} \mathcal{I}\}$ from which a (non-empty) input instance $\mathcal{I}$ 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 can not directly built because the corresponding operation is not strongly monotone).

This 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 hierarchy of rule-based query languages with value invention and stratified negation. The hierarchy is defined relative to the number of strata allowed in stratified programs. The main result 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 non-equality comparisons; and WILOG$^{\frac{1}{2},\neg}$, the class of semipositive programs, allowing for negation on input relations; more precisely, we have shown that these languages express the computable queries of Chandra and Harel (Theorem 4.1), the monotone queries (Theorem 5.2), and the semimonotone 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 lets us 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 $\tilde{p}$ the program expressing the same query in this language ($\tilde{p}$ is certainly different from p, because p does not belong syntactically to WILOG$^{1,\neg}$). Because the computation strategy of $\tilde{p}$ is in general different from that of p (mainly because $\tilde{p}$ can not use the mechanism of strata to alternate existential and universal quantifications, so it must probably use the mechanism of constructing enumerations, which has in general exponential cost), we can not ensure that the complexity of finding a model for $\tilde{p}$ 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. Expressiveness of the two families of languages are very different: the former ranges over the computable queries, whereas the latter does not go beyond the PTIME queries. The hierarchy of WILOG$^{(\neg)}$ relative to the number of strata allowed collapses level '1' (WILOG$^{1,\neg}$, Theorem 4.1); the same hierarchy referred to *datalog*$^{(\neg)}$ does not collapse [29]. Moreover, comparing the result in [29] with that in [4], 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 [20].

Referring to languages with limited use of negation, it is known that the queries expressible in *datalog*$^{\neq}$ and *datalog*$^{\frac{1}{2},\neg}$ 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 [30], Afrati et al. [5]). In contrast, WILOG$^{\neq}$ and WILOG$^{\frac{1}{2},\neg}$ express *exactly* the classes of monotone and semimonotone computable queries, respectively.

The language *datalog*$^{\frac{1}{2},\neg}$ expresses the PTIME queries on ordered databases with *min* and *max* [34]. We obtained a similar result for the language WILOG$^{\frac{1}{2},\neg}$ with respect to the computable queries.

This work is clearly related to the original paper introducing ILOG [23]. However, there the focus is on query issues in the context of an object-based data model, whereas the main concern of this work is on the ability of expressing relational queries, especially with respect to a limited use of stratified negation.

The ability of WILOG$^{\neg}$ (with unbounded stratified negation) to express the computable queries can be inferred from [22]. There, the results refer to *COL*, a rule-based language with stratified negation and untyped set construction. The two approaches are comparable, as it is suggested by Van den Bussche et al. [37], where value invention is related to

hereditarely 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 [1]; 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 [22].

The first completeness result for a *datalog* extension with value invention was shown by Abiteboul and Vianu [4]; the proof technique of building all enumerations of an input instance was also proposed there. Nevertheless, the connection between the family wILOG$^{(\neg)}$ and the *datalog* extensions proposed in [4] 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 (i.e., semipositive, or no negation at all) can be different [23, Example 7.6].

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* [2] and *constructivism* [37]. The subject of querying object-oriented databases has been investigated also by other authors (e.g., [16, 19, 26]). Expressiveness of ILOG$^{\neg}$ as a database transformation languages has been formalized in [11] as the class of *list-constructive transformations*, (introduced by Van den Bussche [36]), 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. The results holding for ILOG$^{\neg}$ are the analogous of those proven for wILOG$^{\neg}$; more precisely, the class of two-strata programs expresses the list-constructive transformations, and ILOG$^{\neq}$ and ILOG$^{\frac{1}{2},\neg}$ express the class of monotone and semimonotone list-constructive transformations, respectively.

References

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

[2] S. Abiteboul and P. Kanellakis. Object identity as a query language primitive. In *ACM SIGMOD International Conf. on Management of Data*, pages 159–173, 1989.

[3] S. Abiteboul and V. Vianu. Procedural languages for database queries and updates. *Journal of Computer and System Science*, 41(2):181–229, August 1990.

[4] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Science*, 43(1):62–124, August 1991.

[5] F. Afrati, S. Cosmadakis, and M. Yannakakis. On Datalog vs. polynomial time. In *Tenth ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems*, pages 13–25, 1991.

[6] A.V. Aho and J.D. Ullman. Universality of data retrieval languages. In *Sixth ACM Symp. on Principles of Programming Languages*, pages 110–117, 1979.

[7] H. Aït-Kaci and R. Nasr. LOGIN a logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.

[8] K. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 493–574. Elsevier Science Publishers (North-Holland), Amsterdam, 1990.

[9] F. Bancilhon. On the completeness of query languages for relational databases. In *Mathematical Foundations of Computer Science, LNCS 64*, pages 112–124. Springer-Verlag, 1978.

[10] L. Cabibbo. On the power of stratified logic programs with value invention for expressing database transformations. In *ICDT'95 (Fifth International Conference on Data Base Theory), Prague, Lecture Notes in Computer Science 893*, pages 208–221, 1995.

[11] L. Cabibbo. *Querying and Updating Complex-Object Databases*. PhD thesis, Università degli Studi di Roma "La Sapienza", 1996.

[12] A.K. Chandra and D. Harel. Computable queries for relational databases. *Journal of Computer and System Science*, 21:333–347, 1980.

[13] W. Chen and D.S. Warren. C-logic for complex objects. In *Eigth ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems*, pages 369–378, 1989.

[14] E.F. Codd. A relational model for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[15] E.F. Codd. Extending the database relational model to capture more meaning. *ACM Trans. on Database Syst.*, 4(4):397–434, December 1979.

[16] K. Denninghoff and V. Vianu. Database method schemas and object creation. In *Twelfth ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems*, pages 265–275, 1993.

[17] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, San Francisco, 1979.

[18] C.A. Gunter and D.S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 633–674. Elsevier Science Publishers (North-Holland), Amsterdam, 1990.

[19] M. Gyssens, J. Paredaens, and D. Van Gucht. A graph-oriented object database model. In *Ninth ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems*, pages 417–424, 1990.

[20] R. Hull and J. Su. Untyped sets, invention, and computable queries. In *Eigth ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems*, pages 347–359, 1989.

[21] R. Hull and J. Su. Algebraic and calculus query languages for recursively typed complex objects. *Journal of Computer and System Science*, 47(1):121–156, August 1993.

[22] R. Hull and J. Su. Deductive query languages for recursively typed complex objects. Technical report, University of Southern California, 1993.

[23] R. Hull and M. Yoshikawa. ILOG: Declarative creation and manipulation of object identifiers. In *Sixteenth International Conference on Very Large Data Bases, Brisbane*, pages 455–468, 1990.

[24] R. Hull and M. Yoshikawa. On the equivalence of database restructurings involving object identifiers. In *Tenth ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems*, pages 328–340, 1991.

[25] N. Immerman. Relational queries computable in polynomial time. *Information and Control*, 68:86–104, 1986.

[26] M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *ACM SIGMOD International Conf. on Management of Data*, pages 393–402, 1992.

[27] M. Kifer and G. Lausen. F-logic: A higher order language for reasoning about objects, inheritance and scheme. In *ACM SIGMOD International Conf. on Management of Data*, pages 134–146, 1989.

[28] M. Kifer and J. Wu. A logic for object-oriented logic programming (Maier's O-logic revisited). In *Eigth ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems*, pages 379–393, 1989.

[29] P.G. Kolaitis. The expressive power of stratified logic programs. *Information and Computation*, 90(1):50–66, January 1991.

[30] P.G. Kolaitis and M. Vardi. On the expressive power of Datalog: Tools and a case study. In *Ninth ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems*, pages 61–71, 1990.

[31] G.M. Kuper and M.Y. Vardi. A new approach to database logic. In *Third ACM SIGACT SIGMOD Symp. on Principles of Database Systems*, 1984.

[32] G.M. Kuper and M.Y. Vardi. The logical data model. *ACM Trans. on Database Syst.*, 18(3):379–413, September 1993.

[33] D. Maier. A logic for objects. In *Workshop on Foundations of Deductive Database and Logic Programming (Washington, D.C. 1986)*, pages 6–26, 1986.

[34] C. Papadimitriou. A note on the expressive power of prolog. *Bulletin of the EATCS*, 26:21–23, 1985.

[35] J. Paredaens. On the expressive power of the relational algebra. *Information Processing Letters*, 7(2):107–111, 1978.

[36] J. Van den Bussche. *Formal Aspects of Object Identity in Database Manipulation.* PhD thesis, University of Antwerp, 1993.

[37] J. Van den Bussche, D. Van Gucht, M. Andries, and M. Gyssens. On the completeness of object-creating query languages. In *33rd Annual Symp. on Foundations of Computer Science*, pages 372–379, 1992.

[38] M. Vardi. The complexity of relational query languages. In *Fourteenth ACM SIGACT Symp. on Theory of Computing*, pages 137–146, 1988.

A Domain Turing Machines

We now describe formally domain Turing Machines (domTMs). They were introduced by Hull and Su [21] as a variant of Turing machines, focused on database manipulation.

In general, Turing Machines (TMs): *(i)* have a finite alphabet, *(ii)* may compute non-generic functions, and *(iii)* use ordered inputs (tape cells are ordered); this implies that an encoding for instances into a finite alphabet must be established and that not all Turing machines can be used as specifications of queries.

Unlike conventional TMs, the alphabet to be used on a domTM tape includes a domain Δ of symbols, which is in general a countable set. It also contains a finite set of 'working symbols,' corresponding to connectives likes parentheses and brackets. A domTM has a two-way infinite tape, and is equipped with a 'register,' which can be used to store a single letter of the alphabet. This register is used to express transitions that are (essentially) 'generic' and to keep finite the control of the machine. For, 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.

The correspondence between domTMs and queries is not yet complete. A domTM tape is used for input and processing; hence, mappings computed by domTMs may depend on the input order. Fortunately, it is possible to restrict the attention to domTMs whose computations, in a sense made precise as follows, do not depend on the input order. A domTM M is *input-order independent* with respect to input database scheme $\mathcal{S}$ if for any input instance $\mathcal{I}$ of $\mathcal{S}$, either (i) for every enumeration e of $\mathcal{I}$, M does not halt; or (ii) there is an instance $\mathcal{J}$ of the output scheme such that, for every enumeration e of $\mathcal{I}$, M halts and the output of M, denoted $M(e)$, is some enumeration of $\mathcal{J}$.

Formally, a *(deterministic) domain Turing machine* (*domTM*) (relative to a domain Δ) is a 6-tuple $M = (K, W, C, \delta, q_s, q_h)$, where

- 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 '`,`','`(`','`)`','`[`','`]`', which are 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 total 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 domTM state, a the register content, and b the content of the tape cell under the head. Here, apart from constants in $W \cup C$, two distinguished symbols η and κ can be used to model templates for infinite sets on transitions. 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. 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 $b = \kappa$.

The domTM M is viewed as having a two-way infinite tape, and a *register*. An *instantaneous description* (ID) of M is a 5-tuple (q, a, α, b, β), where q is a state; $a \in W \cup \Delta \cup \{\#\}$ 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. (Is is assumed the usual restriction 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, intended to range over distinct elements of $\Delta - C$, are used to model templates for infinite sets of transition values: a transition value $\delta(q, \eta, \eta) = \ldots$ 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 $\delta(q, \eta, \kappa) = \ldots$ is applicable in the complementary situation in which they differ. At the beginning of the computation, the register holds $\#$. Under these previsions, a *computation* of M is defined in the usual fashion.

The main point in the introduction of domain Turing machines is that they provide an effective way to implement queries. This is shown in the following important result.

Fact A.1 ([21]) *For any computable query q there is an order independent domTM M_q which computes q. That is, given an input instance $\mathcal{I}$, either M_q does not halt on any enumeration of $\mathcal{I}$ (i.e., q is undefined on input $\mathcal{I}$), or there exists an instance $\mathcal{J}$ such that, for any enumeration e of $\mathcal{I}$, the computation of M_q on e, denoted by $M_q(e)$, halts resulting in an output that is an enumeration of $\mathcal{J}$ (i.e., $q(\mathcal{I}) = \mathcal{J}$).*