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

Model Finiteness and Functionality in a Declarative Language with Oid Invention

Luca Cabibbo*, Giansalvatore Mecca⊗

RT-INF-3-96 **Marzo 1996**

*Università di Roma Tre,
Via della Vasca Navale, 84
00146 Roma, Italy.

⊗ D.I.F.A. – Università della Basilicata
Via della Tecnica 3
85100 Potenza

A preliminary version of this paper appeared under the same title in Secondo Convegno Nazionale su Sistemi Evoluti per Basi di Dati (SEBD-94).
This work was partially supported by MURST, *within the Project "Metodi formali e strumenti per basi di dati evolute", and by* Consiglio Nazionale delle Ricerche, *within "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo, Obiettivo LOGIDATA+".*

ABSTRACT

Two important properties of IsaLog programs are studied: model finiteness and functionality. Finiteness refers to the property of a program of having a finite model over every input instance. Functionality requires a model to contain no contradictory information about object values. These two properties are shown to be undecidable. This is a consequence of the ability of IsaLog programs to simulate computations of arbitrary Turing machines, provided their input is coded as a suitable instance.

Weakly recursive programs, a restricted class of IsaLog programs, is then investigated. It is shown that every weakly recursive program admits a finite model over every input instance. Moreover, models for this class of programs can be computed in polynomial time with respect to the number of objects in the input instance.

1 Introduction

Since the introduction of the Datalog language, the so called "deductive" query languages have had much of attention from the database community. This is essentially due to the high flexibility and to the capacity of expressing recursive queries, whereas the lack of recursive primitives in traditional relational languages (such as relational algebra and calculus) has represented a major limitation [3].

In the last years, the goal of coupling declarative languages and object-oriented data models has been strongly pursued. What an object-oriented data model offers — among others, object identity, object sharing, and inheritance hierarchies — represents a group of very attractive features for next generation database systems. These features, indeed, are even more interesting when embedded in a declarative framework. For example, the presence of oid identity, and thus the need for the creation of new objects, each one required to have a new unique object identifier (oid), has brought to the introduction of the notion of oid invention [2]. Special care has to be devoted to the definition of the semantics of a clause written in a deductive language that performs oid invention. An interesting proposal comes from the ILOG [13] language, in which oid invention is made fully declarative by exploiting a special technique, called *implicit Skolemization*, in which function symbols are used in order to create new objects.

In previous papers [4, 5], the IsaLog data model and language have been presented. Its semantics is much in the spirit of ILOG, since function symbols are exploited as object creators, with some new features:

- *functors* are made *explicit*, where ILOG ones were essentially implicit, thus providing a mechanism for copy control and generation;
- inheritance hierarchies are allowed, and the language is strongly typed. The two things are tightly related, since suitable constraints on the syntax of programs are needed in order to correctly deal with inheritance [7].

In this paper we intend to discuss some interesting aspects of the language due to the interaction among declarative semantics and object-oriented features, namely oid invention, hierarchies, and strong typing. The problems we discuss here are model *finiteness* and *functionality*, two important properties for IsaLog programs.

Finiteness refers to the property of a program of having a finite model over every input instance. This is a strong requirement in a object-oriented database context, since the generation of an infinite number of new objects must be carefully avoided (because it would correspond to a non-terminating computation).

On the other side, a program is said to be *functional* if it preserves the requirement of unique identification associated with object identity (that is, each object — existing or newly created — has a unique, well-defined, associated value). This is a desired property of programs, since the semantics of a non-functional program cannot be properly defined.

In this paper, both properties are shown to be undecidable for IsaLog programs. Sufficient conditions in order to enforce finiteness are introduced, leading to the definition of the class of *weakly recursive programs*, which is heavily based on the strong typing of the language. Besides finiteness, such programs also enjoy the nice property that the problem of finding their model is computationally tractable — a model for a weakly recursive program can always be computed in PTIME (with respect to the number of objects in the input instance).

The technique we use to prove the undecidability results consists in simulating computations of Turing machines, provided their input strings are coded as suitable instances. This sheds further light on the problem of characterizing the expressive power of the language, problem that will be dealed with in a forthcoming paper [6].

The paper is organized as follows. Section 2 briefly reviews the IsaLog data model and language. Section 3 is devoted to the simulation of Turing machines by means of IsaLog programs. Some undecidability results about IsaLog programs descend from this ability. The class of weakly-recursive programs — which enjoy the finiteness property — is introduced in Section 4, where we also discuss some complexity issues. Future research directions are sketched in Section 5.

2 The Data Model and Language

In this section we briefly present the IsaLog data model and language. For a complete presentation of the IsaLog framework we refer the reader to our previous works [4, 5]. In this paper we actually describe a slightly refined version of IsaLog, called IsaLog^{impl}, which exploits an *implicit skolemization mechanism* such as the one elegantly introduced in the ILOG language [13]. We do this in order to simplify the notation throughout the paper, with special regard to the language syntax. Every result obtained in such a way can be easily extended to the original model. Indeed, the two frameworks can be proven to be equivalent; hence, we blur the distiction between them and refer to both just as IsaLog.

2.1 The Data Model

The data model is based on a clear distinction between *scheme* and *instance*. Data are organized by means of two constructs: classes and relations. A *class* is a collection of objects; each object is identified by an *object identifier* (*oid*) and has an associated tuple value. A *relation* is a collection of tuples, used to express relationships among objects and values. Tuples in relations and object values may contain domain values and oid's, used as references to objects. Isa hierarchies are allowed among classes, with multiple inheritance and without any requirement of completeness or disjointness.

We fix a countable set D of *constants*, called the *domain*. An IsaLog scheme is a four-tuple $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \text{TYP}, \text{ISA})$, where:

- $\mathbf{C}$ (the *class names*) and $\mathbf{R}$ (the *relation names*) are finite, pairwise disjoint sets;
- TYP is a total function on $\mathbf{C} \cup \mathbf{R}$ that associates a flat *tuple type* $(A_1 : \tau_1, \ldots, A_k : \tau_k)$ with each class in $\mathbf{C}$ and each relation in $\mathbf{R}$; the A_i's are called *attributes*, and each τ_i (the *type* of A_i) is either a class name in $\mathbf{C}$ or the domain D;
- ISA is a partial order over $\mathbf{C}$, such that if $(C', C'') \in \text{ISA}$ (usually written in infix notation, $C' \text{ ISA } C''$, and read C' is a *subclass* of C''), then $\text{TYP}(C')$ is a *subtype* [1] of $\text{TYP}(C'')$. Multiple inheritance is allowed, with some technical restrictions.

[1] A tuple type τ' is a *subtype* of another tuple type τ'' if, for each attribute in τ'', the attribute also appears in τ', with the same type or — if the type is a class name — with a type that is a subclass. See [7].

It is convenient to define the *types of a scheme* $\mathbf{S}$, where each type is a *simple type* (that is, either the domain D or a class name) or a *tuple type* (whose attributes have simple types associated).

Given two scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$ and $\mathbf{S}' = (\mathbf{C}', \mathbf{R}', \text{TYP}', \text{ISA}')$, we say that $\mathbf{S}'$ is a *subscheme* of $\mathbf{S}$ (denoted by $\mathbf{S}' \subseteq \mathbf{S}$) if the following conditions are satisfied: (i) $\mathbf{C}' \subseteq \mathbf{C}$ and $\mathbf{R}' \subseteq \mathbf{R}$; (ii) if R is a relation name in $\mathbf{R}'$, then $\text{TYP}'(R) = \text{TYP}(R)$; (iii) if C is a class name in $\mathbf{C}'$, then $\text{TYP}'(C) = \text{TYP}(C)$ and for each C_0 in $\mathbf{C}$ such that C ISA C_0 it is the case that $C_0 \in \mathbf{C}'$ and C ISA$'$ C_0.

A scheme gives the structure of the possible instances of the database. The values that appear in instances are: (i) constants from D; (ii) *object identifiers (oid's)* from a countable set $\mathcal{O}$, disjoint from D; (iii) *tuples* over tuple types, whose components are oid's or constants.

An *instance* [2] $\mathbf{s}$ of a scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \text{TYP}, \text{ISA})$ is a triple $\mathbf{s} = (\mathbf{c}, \mathbf{r}, \mathbf{o})$, where:

- $\mathbf{c}$ is a function that associates with each class name $C \in \mathbf{C}$ a finite set of oid's, preserving the containment constraints (associated with subclass relationships) and disjointness constraints (associated with distinct taxonomies);
- $\mathbf{r}$ is a function that associates with each relation name $R \in \mathbf{R}$ a finite set of tuples over $\text{TYP}(R)$;
- $\mathbf{o}$ is a function that associates tuples with oid's in classes, with the appropriate type;
- if a tuple type has an attribute A whose type is a class $C \in \mathbf{C}$, then the value of the tuple over A is an oid in $\mathbf{c}(C)$ (this condition is required in order to avoid "dangling references").

2.2 IsaLog Syntax

The IsaLog language is declarative and strongly typed, a suitable extension of Datalog [8] capable of handling oid invention and hierarchies. The language semantics is based on the well-known semantics of ordinary logic programming with function symbols, due to the presence of functors in the model. *Functors* are essentially function symbols; they are used in IsaLog programs as a tool to make oid inventions fully declarative. In this paper we adopt the technique consisting in keeping functors hidden, using them at the semantic level to invent new objects.

Let a scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \text{TYP}, \text{ISA})$ be fixed. Also, consider two disjoint countable sets of *variables*: V_D (*value variables*, to denote constants) and $V_{\mathbf{C}}$ (*oid variables*, to denote oid's).

The *terms* of the language are:

- *value terms*, that are: (i) the constants in D and (ii) the variables in V_D;
- *oid terms*: (i) the oid's in $\mathcal{O}$ and (ii) the variables in $V_{\mathbf{C}}$;
- the *oid-invention term* $*$: it is used in order to represent oid's to be invented.

[2] This definition is slightly different from that in [4], where a *pre-instance* is defined as an instance here, whereas an *instance* is defined as an equivalence class of pre-instances. However, the discussion in this paper is not affected from this definition.

The *atoms* of the language may have two forms (where a term t_i in a component is an oid term or a value term depending on the type τ_i associated with the attribute A_i):

- *class atoms*: $C(\text{OID} : t_0, A_1 : t_1, \ldots, A_k : t_k)$, where C is a class name in $\mathbf{C}$, $\text{TYP}(C) = (A_1 : \tau_1, \ldots, A_k : \tau_k)$, and t_0 is an oid term or the oid-invention term;
- *relation atoms*: $R(A_1 : t_1, \ldots, A_k : t_k)$, where R is a relation name in $\mathbf{R}$, with type $\text{TYP}(R) = (A_1 : \tau_1, \ldots, A_k : \tau_k)$.

The notions of *(positive) literal, rule, fact,* and *clause* are as usual. The head and body of a clause γ are denoted with $\text{HEAD}(\gamma)$ and $\text{BODY}(\gamma)$, respectively. There are three relevant forms of clauses. A clause γ is:

- a *relation clause* if $\text{HEAD}(\gamma)$ is a relation atom;
- an *oid-invention clause* if $\text{HEAD}(\gamma)$ is a class atom $C(\text{OID} : *, A_1 : t_1, \ldots, A_k : t_k)$;
- a *specialization clause* if $\text{HEAD}(\gamma)$ is a class atom $C(\text{OID} : X, \ldots)$, where X is an oid variable and $\text{BODY}(\gamma)$ contains (at least) a class atom $C'(\text{OID} : X, \ldots)$ such that C and C' have a common ancestor (that is, a class C_0 such that $C \text{ ISA } C_0$ and $C' \text{ ISA } C_0$).

Hereinafter we consider only clauses of the above three forms.

An IsaLog *program* $\mathbf{P}$ over a scheme $\mathbf{S}$ is a set of clauses that satisfy some technical conditions: *well-typedness* (about typing of oid terms), *safety* (as usual), *visibility* (no explicit oid's are allowed), and $*$-*usage* (the oid-invention term $*$ occurs only in the head of oid-invention clauses).

An *input-output scheme* (or, simply, *i-o scheme*) is a pair $\langle \mathbf{S}_{in}, \mathbf{S}_{out} \rangle$, where $\mathbf{S}_{in}$ and $\mathbf{S}_{out}$ are schemes called the *input scheme* and the *output scheme*, respectively.

When a program is applied to a database, its semantics is defined identifying some classes and relations as its input (say, the *extensional* part), and others as its output (the *intensional* part). Because of the presence of isa hierarchies, we do not require disjointness among the input and output schemes. In this paper, to simplify the presentation, we assume that $\mathbf{S}_{in}$ is a subscheme of $\mathbf{S}_{out}$, that is, the whole input and temporary classes and relations are considered as part of the output as well. Furthermore, when the input scheme is not important for the discussion, we simply refer to the scheme of a program meaning its output scheme.

2.3 Semantics of IsaLog Programs

In [4] three different semantics for IsaLog programs have been defined and shown to be equivalent. The first semantics is a model-theoretic semantics, the second is a fixpoint semantics, and the third one is based on a reduction to logic programming with function symbols. Here we mainly refer to (a variant of) the latter one.

As in ILOG [13], the semantics of IsaLog programs can be easily reduced to ordinary model-theoretic semantics of logic programs with function symbols once one introduces the notion of *implicit Skolemization* of a program.

First, note that each instance $\mathbf{s} = (\mathbf{c}, \mathbf{r}, \mathbf{o})$ of a scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \text{TYP}, \text{ISA})$ corresponds to a set of atoms $\phi(\mathbf{s})$ of the language; in particular, $\phi(\mathbf{s})$ contains:

- an atom $R(A_1 : t_1, \ldots, A_k : t_k)$ for each tuple $(A_1 : t_1, \ldots, A_k : t_k)$ in $\mathbf{r}(R)$;

- an atom $C(\text{OID} : o, A_1 : t_1, \ldots, A_k : t_k)$ for each oid o in $\mathbf{c}(C)$, where $A_1, \ldots, A_k$ are the attributes of C and $(A_1 : t_1, \ldots, A_k : t_k)$ is the restriction of $\mathbf{o}(o)$ to $A_1, \ldots, A_k$.

Now, let $\mathbf{P}$ be a IsaLog program. The *Skolemization* of $\mathbf{P}$, denoted $\text{SKOL}(\mathbf{P})$, is obtained by the following transformation from $\mathbf{P}$:

1. for each class $C \in \mathbf{C}$ introduce a functor F_C, called the *Skolem functor for* C. Then, if $\text{TYP}(C)$ equals $(A_1 : \tau_1, \ldots, A_k : \tau_k)$, we define $F_C(A_1 : t_1, \ldots, A_k : t_k)$ as a (*functor*) *term* of the language;

2. replace the head of each invention rule in $\mathbf{P}$ having the form $C(\text{OID} : *, A_1 : t_1, \ldots, A_k : t_k)$ by $C(\text{OID} : F_C(A_1 : t_1, \ldots, A_k : t_k), A_1 : t_1, \ldots, A_k : t_k)$.

The *semantics* of an IsaLog program $\mathbf{P}$ over an i-o scheme $\langle \mathbf{S}_{in}, \mathbf{S}_{out} \rangle$ is a function that maps instances of $\mathbf{S}_{in}$ to instances of $\mathbf{S}_{out}$. Given an instance $\mathbf{s}$ of $\mathbf{S}_{in}$, the semantics of $\mathbf{P}$ over $\mathbf{s}$ can be found by means of the following steps, described informally:

1. find the skolemization $\text{SKOL}(\mathbf{P})$ of $\mathbf{P}$;

2. find the set of atoms $\phi(\mathbf{s})$;

3. $\text{SKOL}(\mathbf{P}) \cup \phi(\mathbf{s})$ represents essentially a set of clauses of Datalog with function symbols; its minimum model, $\mathcal{M}_{\text{SKOL}(\mathbf{P}) \cup \phi(\mathbf{s})}$, can be found via a fixpoint computation that uses a special operator, $T^{\text{ISA}}_{\text{SKOL}(\mathbf{P}) \cup \phi(\mathbf{s})}$ [5], in order to deal with inheritance; if $\mathcal{M}_{\text{SKOL}(\mathbf{P}) \cup \phi(\mathbf{s})}$ exists, we call it the *model* of $\mathbf{P}$ over $\mathbf{s}$;

4. if $\mathcal{M}_{\text{SKOL}(\mathbf{P}) \cup \phi(\mathbf{s})}$ exists and is finite, it is something similar to a set of atoms of the language, apart from the presence of Skolem terms. In order to obtain an instance of the scheme, we must coherently replace functor terms by new oid's;

5. finally, if the set of facts that represents the result of the previous step satisfies some suitable conditions (see below), then it is possible to find the corresponding instance $\mathbf{s}'$ by means of a transformation that is the inverse of ϕ; if such $\mathbf{s}'$ exists, we call it the *semantics* of $\mathbf{P}$ over $\mathbf{s}$. Otherwise, the *semantics* is *undefined*.

The main difference with respect to the semantics of a Datalog program is the possibility that the semantics of an IsaLog program over an instance is undefined. There are two main reasons for this fact, corresponding to some of the extensions of the model and language with respect to the traditional Datalog framework, where minimum models always exist [8], and thus the semantics is always defined:

- Recursion through oid invention can lead to the generation of infinite sets of facts, against the hypothesis of finite structures. In this case the model of the program over the input instance would be infinite and the semantics would be undefined. Consider, for example, the following clause, in which the class C_0 has type (), and C_1 is a subclass of C_0 with type $(C_{ref} : C_0)$:

  $$\gamma : C_1(\text{OID} : *, C_{ref} : X) \leftarrow C_1(\text{OID} : X, C_{ref} : Y).$$

  The program made of this single clause has clearly no finite model unless the class C_1 is empty in the input instance. This is due to the fact that, according to γ, for each object in C_1 a new object must be invented, leading to a non finite number of objects in the model of γ.

- The presence of isa hierarchies and specialization clauses allows for multiple and inconsistent specializations of an oid from a superclass to a subclass: this may lead to non functional relationships from oid's to object values. In this case the semantics is undefined as well. In fact, as it is shown in [4], it is always the case that a model $\mathcal{M}_{\mathrm{SKOL(P)}\cup\phi\mathrm{(s)}}$ exists, even though infinite. If the model is finite, in order to find the corresponding output instance, we need only to check functionality of the relationship among oid's and values, that is, the satisfaction of the following condition:

  FUN (*functionality*): there cannot be two different facts $C'(\text{OID} : t'_0, \ldots, A : t', \ldots)$ and $C''(\text{OID} : t''_0, \ldots, A : t'', \ldots)$, with $t'_0 = t''_0$ and $t' \neq t''$. That is, two facts for the same oid term must have respectively identical values for the common attributes.

  Condition FUN is not always satisfied by a model of an IsaLog program, as the following example from [4] shows. Consider the following scheme:

CLASS person	*(name:String)*
CLASS husband ISA *person*	*(name:String, wife:person)*
RELATION marriage	*(husband:person, wife:person)*

  Suppose we know all the persons and want to fill the class of the husbands, on the basis of the relation *marriage*, using the following rule:

  $$husband(\text{OID}{:}X,\ name{:}H,\ wife{:}Y) \leftarrow marriage(husband{:}X,\ wife{:}Y),\ person(\text{OID}{:}X,\ name{:}H).$$

  The problem of inconsistent multiple specializations for the same object arises if persons with more than one wife are allowed in the input instance. In this case, the program made of this rule has a model, but it contains contradictory facts, so that the semantics is undefined.

We say that a program is *finite* (resp., *functional*) if admits a finite (resp., functional) model over every input instance — possibly allowing for non-functionality (resp., non-finiteness). If a program is both finite and functional we say that is *defined*; only defined programs admit a semantics that is a total function. The problem of determining whether a program is defined (resp., finite, or functional) is called *definedness* (resp. *finiteness,* or *functionality*).

As it is shown in Section 3, the above problems are undecidable in general.

3 Recursive Programs

The IsaLog data model allows for modeling types that are defined inductively, such as lists, trees, and (in some sense) sets. This is an interesting feature, since it allows for managing complex data structures even though the data model, which has been kept as simple as possible, does not explicitly provide complex types. This modeling ability is a consequence of having class names as user-defined types of a scheme, in such a way that

the type of an attribute of a class may be another class name. At the instance level, the value associated with an object may be an oid — an indirect reference to another object. Furthermore, the presence of isa hierarchies allows to define a type as the "union" of different types — a class as the disjunction of its subclasses. And this is all we need to have inductively defined types.

Example 3.1 A type *string*, that is, a list of characters, is inductively defined as (i) the empty string; or (ii) a character followed by a string. We can represent this definition by means of the following scheme $\mathbf{S}_{string}$:

CLASS string	*()*
CLASS $string_{\epsilon}$ ISA *string*	*()*
CLASS $string_{n\epsilon}$ ISA *string*	*(ch:char, s:string)*

Such a scheme allows for having instances representing unbounded structures, that is, structures over a finite alphabet containing an arbitrarily large number of objects. Indeed, we can represent by means of $\mathbf{S}_{string}$ a string of any length (over a fixed alphabet, represented by class *char*). Given a string w in $char^*$, let us define the instance $\mathbf{s}_w$ of $\mathbf{S}_{string}$ representing string w. Instance $\mathbf{s}_w$ contains, in class *string*, all and only the strings that are suffixes for w. That is, if $w = a_1 \ldots a_n$, with $n \geq 0$, then $\mathbf{s}_w$ would contain $n+1$ objects $o_0, o_1, \ldots, o_n$, where $\mathbf{c}(string_{\epsilon}) = \{o_0\}$ (just the empty string), $\mathbf{c}(string_{n\epsilon}) = \{o_1, \ldots, o_n\}$, with $\mathbf{o}(o_{i+1}) = (ch : a_{n-i}, s : o_i)$. Using functors, this can be equivalently stated as $o_{i+1} = F_{n\epsilon}(ch : a_{n-i}, s : o_i)$, where $F_{n\epsilon}$ is the functor for class $string_{n\epsilon}$. Finally, $\mathbf{c}(string) = \{o_0, o_1, \ldots, o_n\}$.

Note that Datalog, which refers to the relational model, does not provide the capability of "inventing" new values, so that a Datalog program may only admit finite minimal models. In contrast, now we show that there exist IsaLog programs that define unbounded structures, that is, that may have no finite model over an input instance. Those programs are said to be recursive through oid invention. Intuitively, a program is *recursive through oid invention* when the invention of an object in a class depends (directly or indirectly) on the presence of other objects in the same class. Seen from another perspective, in a program that involves recursion through oid invention, the invention of an object in a class can give rise to the invention of an unbounded number of other objects in the same class. In the following, we refer to programs that are recursive through oid invention simply as *recursive* programs, since no ambiguity can arise with respect to the notion of recursion defined for Datalog programs.

Example 3.2 Consider program $\mathbf{P}_{Kleene}$, over a scheme obtained by extending $\mathbf{S}_{string}$ of Example 3.1 with a unary relation R, defined as follows.

$$\begin{aligned} string_{\epsilon}(\text{OID} : *) &\leftarrow true. \\ string_{n\epsilon}(\text{OID} : *, ch : C, s : S) &\leftarrow R(ch : C), string(\text{OID} : S). \end{aligned}$$

The first clause invents only a new object (being *true* a predicate which is always satisfied) corresponding to the empty string. The second clause takes characters from a unary relation R, concatenating them in all possible ways. In this way, class *string* would contain all the strings over the alphabet R. Program $\mathbf{P}_{Kleene}$ is recursive (in fact, invention of objects in class $string_{n\epsilon}$, that will belong to class *string* as well, depends on the existence of other objects in class *string*). Note that this program admits a finite model over an instance $\mathbf{s}$ if and only if the relation R is empty in $\mathbf{s}$.

Program $\mathbf{P}_{Kleene}$ in the example above does not enjoy the finiteness property, according to the definition in Section 2.3. Indeed, instances exist over which $\mathbf{P}_{Kleene}$ has no finite model (namely, all those instances having R non-empty).

In the rest of the section we study decidability of the finiteness problem, starting from the problem of determining whether a given program admits a finite model over a given instance. We also study the functionality problem.

The results we obtain are negative. Indeed, both the problems are shown to be undecidable. The proof is by simulating the computation of an arbitrary Turing machine on an input string. The used technique is particularly interesting since it also points out a big potential for the expressive power of the IsaLog language.

3.1 Simulating Turing Machines

In this section we show how to write an IsaLog program that simulates the computation of a Turing machine on an input string, provided the string is coded as a suitable IsaLog instance.

The following definitions are from [18]. A *Turing machine* M is a quadruple (K, Σ, δ, s), where:

- K is a finite set of *states*, not containing the halt state h;
- Σ is a finite alphabet, containing the blank symbol #;
- $s \in K$ is the *initial state*;
- δ is a total function from $K \times \Sigma$ to $(K \cup \{h\}) \times \Sigma \times \{\leftarrow, \rightarrow, -\}$.

If $q \in K$, $a \in \Sigma$, and $\delta(q, a) = (p, b, d)$, then M, when in state q and scanning symbol a, will enter state p, rewrite a as b, and move its head in the direction shown by d (that is, if d equals $\leftarrow$ ($\rightarrow$) then move the head to the left (right), and do not move the head if d equals $-$).

A *configuration* of a Turing machine $M = (K, \Sigma, \delta, s)$ is a member of $(K \cup \{h\}) \times ((\Sigma - \{\#\})\Sigma^* \cup \epsilon) \times \Sigma \times (\Sigma^*(\Sigma - \{\#\}) \cup \epsilon)$, where ϵ denotes the empty string. Intuitively, the configuration (q, w_L, c, w_R) contains the complete description of a current global state of a computation, in which: q is the current state, c is the character being scanned by the head, and w_L (w_R) is the string representing the content of the tape at the left- (right-) hand side of the head.

The *yields in one step* binary relation on the configurations of M (denoted by $\vdash_M$) is defined in such a way that if M in configuration $C = (q, w_L, c, w_R)$ using $\delta(q, c) = (q', b, d)$ reaches configuration $C' = (q', w'_L, c', w'_R)$, then $C \vdash_M C'$. Moreover, $\vdash^*_M$ denotes the reflexive and transitive closure of $\vdash_M$, which is called *yields*.

Let Σ_0 and Σ_1 be alphabets not containing the blank symbol #. We say that a Turing machine $M = (K, \Sigma, \delta, s)$ *computes* a function f from Σ_0^* to Σ_1^* if $\Sigma_0, \Sigma_1 \subseteq \Sigma$ and for any $w \in \Sigma_0^*$, if $f(w) = u$, then $(s, \epsilon, \#, w\#) \vdash^*_M (h, \epsilon, \#, u\#)$.

Now we turn to the problem of simulating an arbitrary Turing machine by means of an IsaLog program.

Consider the Turing machine $M = (K, \Sigma, \delta, s)$. The simulation is defined over a scheme that contains $\mathbf{S}_{string}$ of Example 3.1 as a subscheme. Moreover, we define class *input* as a subclass of *string*, without further attributes, to contain objects representing strings

on which the simulation must be carried on. In this context we will assume that *input* contains only one object, that is, the object that properly encodes the input string for a computation of the Turing machine; with other words, we can say that we deal basically with computations over a single input string. The input scheme for our simulation is defined as $\mathbf{S}_{string}$ extended with class *input*.

The output scheme contains other classes and relations, as follows.

Class *output*, a subclass of *string* without further attributes, will contain the object representing the string (if any) computed by M on the given input.

Class *state* contains as objects the states of M, that is, it represents the set K, whereas the alphabet Σ is represented by class *char*.

Class $Conf_0$ will contain configurations of the Turing machine, where $\text{TYP}(Conf_0)$ is equal to *(state:state, left:string, ch:char, right:string)*. Each object in $Conf_0$ is a configuration reached by the computation of M on the input string, where: *state* represents the state; *ch* the character scanned by the head; *right* codes the portion of the tape on the right-hand side of the head; and *left* codes the portion on its left, using the convention that it is in *reverse* order, that is, read from the right to the left. Class $Conf_0$ has two subclasses: $Conf_{init}$, of the same type as $Conf_0$, to represent the initial configuration of a computation; and *Conf*, for non-initial configurations, with type *(state:state, left:string, ch:char, right:string, previous:*$Conf_0$*)*; attribute *previous* links subsequent configuratios: if an object o in class *Conf* has for *previous* the value o' — which must be an object in $Conf_0$ — this means that the configuration represented by o is yielded in one step from the one represented by o'.

Relation *toCreate*, of type *(ch:char, tail:string)*, is used to mantain a request pool for the invention of the new strings needed for the computation to evolve. Mainly, it is used to deal with "expansions" of the working area, that occur when the head reaches the left- or right-end of the tape. Finally, relations *first* and *tail*, of type *(string:string, ch:char)* and *(string:string, tail:string)* respectively, are used to access the first character and the tail of a string. We need them in order to correctly deal with the empty string.

Let us go to the description of the IsaLog program [3] $\mathbf{P}_M$ simulating the Turing machine M.

Class *Conf* is initialized by:

$$Conf_{init}(\text{OID} : *, s, E, \#, X) \quad \leftarrow \quad input(\text{OID} : X), string_{\epsilon}(\text{OID} : E).$$

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

$$Conf(\text{OID} : *, q', L, a', R, P) \quad \leftarrow \quad Conf_0(\text{OID} : P, q, L, a, R).$$

For each pair (q, a) such that $\delta(q, a) = (q', a', \rightarrow)$, that are, right-moving transitions, we have clauses:

$$\begin{aligned} toCreate(a', L) \quad &\leftarrow \quad Conf_0(\text{OID} : I, q, L, a, R). \\ Conf(\text{OID} : *, q', L', X, R', P) \quad &\leftarrow \quad Conf_0(\text{OID} : P, q, L, a, R), \\ &\qquad first(R, X), tail(R, R'), first(L', a'), tail(L', L). \end{aligned}$$

Similarly for left-moving transitions $\delta(q, a) = (q', a', \leftarrow)$:

[3] In order to simplify the presentation and without loss of generality, from now on we use a positional syntax for atoms, omitting attribute names. For example, if R is a relation name and $\text{TYP}(R) = (A_1 : \tau_1, \ldots, A_k : \tau_k)$, we write here $R(t_1, \ldots, t_k)$ rather than $R(A_1 : t_1, \ldots, A_k : t_k)$.

$$
\begin{array}{rcl}
toCreate(a', R) & \leftarrow & Conf_0(\text{OID} : I, q, L, a, R). \\
Conf(\text{OID} : *, q', L', X, R', P) & \leftarrow & Conf_0(\text{OID} : P, q, L, a, R), \\
 & & first(L, X), tail(L, L'), first(R', a'), tail(R', R).
\end{array}
$$

Moreover we have clauses defining *first, tail*, and new objects in *string*, with respect to creation requests in relation *toCreate* — where we have the first clause repeated for each character $a \in (\Sigma - \{\#\})$:

$$
\begin{array}{rcl}
string_{n\epsilon}(\text{OID} : *, ch : a, s : S) & \leftarrow & toCreate(a, S), string(\text{OID} : S). \\
string_{n\epsilon}(\text{OID} : *, ch : \#, s : S) & \leftarrow & toCreate(\#, S), string_{n\epsilon}(\text{OID} : S). \\
first(X, F) & \leftarrow & string_{n\epsilon}(\text{OID} : X, ch : F, s : R). \\
first(X, \#) & \leftarrow & string_{\epsilon}(\text{OID} : X). \\
tail(X, R) & \leftarrow & string_{n\epsilon}(\text{OID} : X, ch : F, s : R). \\
tail(X, X) & \leftarrow & string_{\epsilon}(\text{OID} : X).
\end{array}
$$

Finally we include the clause that (possibly) detects the termination of the computation:

$$
\begin{array}{rcl}
output(\text{OID} : X) & \leftarrow & Conf_0(\text{OID} : I, h, E, \#, X), \\
 & & string(\text{OID} : X), string_{\epsilon}(\text{OID} : E).
\end{array}
$$

Now, the computation of M on input string $w \in \Sigma^*$ can be simulated by program $\mathbf{P}_M$ over input instance $\mathbf{s}_w$, where $\mathbf{s}_w$ is defined as in Example 3.1, but having also $\mathbf{c}(input) = \{o_n\}$, where o_n represents the whole string w.

The intuition is that, if the computation of M on input w halts, this happens having M reached a finite number of configurations, which can be represented by a finite number of strings. On the other hand, if the computation does not halt, the number of reached configurations is not finite. Indeed, we have the following result:

Lemma 3.3 *Let M be a Turing machine, $\mathbf{P}_M$ the corresponding* IsaLog *program, w a string, and $\mathbf{s}_w$ the corresponding instance. $\mathbf{P}_M$ has a finite model over $\mathbf{s}_w$ if and only if M halts on input w.*

It is worth noting that this finite model contains only an object in class *output*, which represents the string result of the computation.

Now we can state the main results of this section:

Theorem 3.4 *The following problems concerning* model finiteness *of* IsaLog *programs are undecidable:*

1. *Given a program $\mathbf{P}$ and an instance $\mathbf{s}$, has $\mathbf{P}$ a finite model over $\mathbf{s}$?*
2. Finiteness: *Given a program $\mathbf{P}$ over i-o scheme $\langle \mathbf{S}_{in}, \mathbf{S}_{out} \rangle$, has $\mathbf{P}$ a finite model over every instance of $\mathbf{S}_{in}$?*
3. *Given two programs $\mathbf{P}_1$ and $\mathbf{P}_2$ and an instance $\mathbf{s}$, have $\mathbf{P}_1$ and $\mathbf{P}_2$ the same model over $\mathbf{s}$?*

Proof. (Sketch). Part 1 is a consequence of Lemma 3.3 and undecidability of the halting problem for Turing machines.

For part 2, consider again the program $\mathbf{P}_M$ simulating a Turing machine M. A generic instance $\mathbf{s}$ of its input scheme represents a set of input strings for M, and $\mathbf{P}_M$ has a finite model over $\mathbf{s}$ if and only if M halts on all this set of strings. So, $\mathbf{P}_M$ has a finite model over every input instance if and only if M halts on every input string in Σ^*, which is undecidable.

Part 3 descends from the undecidability of the equivalence of two partial recursive functions. □

As a consequence of the previous theorem, the finiteness property has been proved to be undecidable in the general case. In the next section we study sufficient conditions that guarantee model finiteness for particular classes of IsaLog programs.

Using similar arguments, we can prove the following:

Theorem 3.5 *The following problems concerning* functionality *of* IsaLog *programs are undecidable:*

1. *Given a program* $\mathbf{P}$ *and an instance* $\mathbf{s}$*, is* $\mathbf{P}$ *functional over* $\mathbf{s}$*?*
2. Functionality: *Given a program* $\mathbf{P}$ *over i-o scheme* $\langle \mathbf{S}_{in}, \mathbf{S}_{out} \rangle$*, is* $\mathbf{P}$ *functional over every instance of* $\mathbf{S}_{in}$*?*

4 Weakly Recursive Programs

This section is concerned with the finiteness of IsaLog programs, that is, the property of a program to have a finite model over every possible instance.

We have proved the last section that the problem is undecidable in general. What we try to do now is to give some sufficient conditions over the scheme and the program in order to enforce model finiteness. This is a very appealing aim, being infiniteness of models a situation to be carefully avoided, because it would correspond to non-terminating computations. In [13], the notion of *weakly recursive program* is introduced. Here we claim that in a strongly-typed framework, such as IsaLog, easier conditions can be stated. This is basically due to the restrictions on the syntax of programs in order to guarantee type correctness. Some computational aspects are also presented.

Intuitively, a first sufficient condition for finiteness is having no cyclic types in the scheme. This idea leads to the following characterization of schemes.

We say that an IsaLog scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \text{TYP}, \text{ISA})$ is *cyclic* if there exist an $n > 0$, a sequence of class names $C_1, \ldots, C_n \in \mathbf{C}$, and a sequence of attribute names $A_1, \ldots, A_n$ such that:

- for $i \in \{1, \ldots, n\}$ it is the case that A_i is an attribute of C_i, whose type τ_i is a class name;
- $\tau_i \sim C_{i+1}$ for $i \in \{1, \ldots, n-1\}$ and $\tau_n \sim C_1$, where $C' \sim C''$, with $C', C'' \in \mathbf{C}$, if C' and C'' have a common ancestor in $\mathbf{S}$.

Furthermore, we say that $\mathbf{S}$ is *acyclic* if it is not cyclic.

Example 4.1 Consider the IsaLog scheme $\mathbf{S}_{string}$ of Example 3.1. This scheme is cyclic. In the cyclicity conditions, assume $n = 1$, $C_1 = string_{n\epsilon}$ and $A_1 = s$. We have that the type of s in the class $string_{n\epsilon}$ is $string$ and $string_{n\epsilon} \sim string$.

We have already seen in Example 3.1 how to represent a string of length m by means of an IsaLog instance. The latter contains an object o_0 *from the scratch* (the object corresponding to the empty string) plus m objects defined by means of o_0 and a functor. Now, it is possible to define such an instance for every m arbitrarily large.

Such a situation, in which an object in a class can cause the invention of another object in the same class, is likely to produce infiniteness. Clearly, programs over acyclic schemes cannot produce such an effect. This observation is formalized in the following lemma (proof omitted).

Lemma 4.2 *Let* $\mathbf{S}$ *be an acyclic scheme and* $\mathbf{s}$ *an instance of* $\mathbf{S}$*; suppose* n *denotes the cardinality of the active domain of* $\mathbf{s}$*. Given a program* $\mathbf{P}$ *over* $\mathbf{S}$*, if* $\mathbf{P}$ *is functional over* $\mathbf{s}$*, then the semantics* $\mathbf{s}'$ *of* $\mathbf{P}$ *over* $\mathbf{s}$ *is finite and the cardinality of its active domain is at most* $p_{\mathbf{S}}(n)$*, where* $p_{\mathbf{S}}$ *is a polynomial depending only on* $\mathbf{S}$*.*

As a consequence, referring to the fixpoint semantics of IsaLog [4], the problem of finding a model for a program over an acyclic scheme is computationally tractable.

Corollary 4.3 *Let* $\mathbf{S}$ *be an acyclic scheme, and* $\mathbf{s}$ *an instance of* $\mathbf{S}$*, For every program* $\mathbf{P}$ *over* $\mathbf{S}$*, the following answers can be computed in polynomial time with respect to the active domain of* $\mathbf{s}$*:*

- *the model of* $\mathbf{P}$ *over* $\mathbf{s}$*;*
- *if* $\mathbf{P}$ *is functional over* $\mathbf{s}$*, the semantics of* $\mathbf{P}$ *over* $\mathbf{s}$*.*

Now we give a graph-based characterization for (a)cyclic schemes. On one hand, the following characterization is important because several object-oriented data models define scheme as (labelled) graphs [10]. On the other hand, definitions concerning weakly recursive programs (see below) are more natural if expressed in a graph-based fashion.

Let us define the *dependency graph* $\mathcal{G}^{dep}_{\mathbf{S}}$ for a scheme $\mathbf{S}$ as follows:

- $\mathcal{G}^{dep}_{\mathbf{S}}$ contains a node n_C for each class $C \in \mathbf{C}$;
- for each pair of classes C, C', such that C ISA C', $\mathcal{G}^{dep}_{\mathbf{S}}$ contains: a *blue edge* directed from n_C to $n_{C'}$ and a *red edge* directed from $n_{C'}$ to n_C;
- for each class $C \in \mathbf{C}$ and for each of its attributes A_i such that the type of A_i is a class $C' \in \mathbf{C}$, $\mathcal{G}^{dep}_{\mathbf{S}}$ contains a *black edge* directed from n_C to $n_{C'}$.

With respect to the dependency graph of a scheme, we have the following result.

Theorem 4.4 *A scheme* $\mathbf{S}$ *is cyclic if and only if its scheme dependency graph* $\mathcal{G}^{dep}_{\mathbf{S}}$ *contains a cycle with a black edge.*

The above discussion highlights that every program over an acyclic scheme satisfies the finiteness property. It essentially states that, if there is no cyclic structure in the scheme, that is, no type defined in terms of itself, it is not possible for a program to generate an infinite number of objects. However, requiring scheme acyclicity to ensure finiteness is a very strict condition. In fact, sometimes programs over cyclic schemes admit finite models (for example, if the cyclic part of the scheme is "read-only"). So, now we turn to the problem of finding sufficient conditions for programs over cyclic schemes to have the finiteness property.

The *dependency graph* $\mathcal{G}^{dep}_{\mathbf{P},\mathbf{S}}$ for a program $\mathbf{P}$ over a scheme $\mathbf{S}$ is defined as follows:

- $\mathcal{G}^{dep}_{\mathbf{P},\mathbf{S}}$ contains a node n_C for each class $C \in \mathbf{C}$;
- for each pair of classes C, C', such that C ISA C', and $\mathbf{P}$ contains an oid invention clause for C, $\mathcal{G}^{dep}_{\mathbf{P},\mathbf{S}}$ contains a *red edge* directed from $n_{C'}$ to n_C;
- for each pair of classes C, C', such that C ISA C', and $\mathbf{P}$ contains a specialization clause from C' to C, $\mathcal{G}^{dep}_{\mathbf{P},\mathbf{S}}$ contains a *blue edge* directed from n_C to $n_{C'}$;
- for each class $C \in \mathbf{C}$ and for each of its attributes A_i such that the type of A_i is a class $C' \in \mathbf{C}$, and $\mathbf{P}$ contains an oid-invention clause for C, $\mathcal{G}^{dep}_{\mathbf{P},\mathbf{S}}$ contains a *black edge* directed from C to C'.

We can extend the notion of cyclicity to programs by defining a *weakly recursive program* as a program such that its dependency graph $\mathcal{G}^{dep}_{\mathbf{P},\mathbf{S}}$ does not contains a cycle with a black edge.

In a weakly recursive program, no recursion through oid invention occurs, and this justifies the definition. Thus, even if the program refers to a cyclic scheme, non finiteness problems cannot arise, since the creation of recursive objects does not happen.

Clearly, every program over an acyclic scheme is weakly recursive, so that we can state the following results, that generalize the previous ones.

Lemma 4.5 *Let* $\mathbf{S}$ *be a scheme and* $\mathbf{s}$ *an instance of* $\mathbf{S}$*; suppose* n *denotes the cardinality of the active domain of* $\mathbf{s}$*. Given a program* $\mathbf{P}$ *over* $\mathbf{S}$*, if* $\mathbf{P}$ *is weakly recursive and functional over* $\mathbf{s}$*, then the semantics* $\mathbf{s}'$ *of* $\mathbf{P}$ *over* $\mathbf{s}$ *is finite and the cardinality of its active domain is at most* $p_{\mathbf{P},\mathbf{S}}(n)$*, where* $p_{\mathbf{P},\mathbf{S}}$ *is a polynomial depending only on* $\mathbf{P}$ *and* $\mathbf{S}$*.*

For weakly recursive programs, the problem of finding a model is still computationally tractable.

Corollary 4.6 *Let* $\mathbf{S}$ *be a scheme, and* $\mathbf{s}$ *an instance of* $\mathbf{S}$*, For every weakly recursive program* $\mathbf{P}$ *over* $\mathbf{S}$*, the following answers can be computed in polynomial time with respect to the active domain of* $\mathbf{s}$*:*

- *the model of* $\mathbf{P}$ *over* $\mathbf{s}$*;*
- *if* $\mathbf{P}$ *is functional over* $\mathbf{s}$*, the semantics of* $\mathbf{P}$ *over* $\mathbf{s}$*.*

5 Discussion

5.1 Towards the Language Expressive Power

We have shown that the IsaLog language is powerful enough to simulate arbitrary Turing machines. This can be done because of the ability of building unbounded recursive structures, made of objects. Hence, we can expect a great potential of expressive power. Nevertheless, it turns out that IsaLog is not computationally complete for relational transformations, in the sense that it cannot express all *computable queries* of [9]. Consequently, with respect to the ability of expressing *object-oriented* queries, IsaLog is complete neither in the sense of [2] nor in that of [20]. This inability is not in contradiction with our simulation, where we have assumed some restrictions on the nature of the computations, as follows:

- we assume the input instance to code an input string for the Turing machine in a suitable way. In the most general and usual case, the input is supposed to be a set of relations, that is, an arbitrary instance of a given scheme, without further limitations;
- we assume the alphabet to be fixed and finite; in general we must consider a domain which is countable.

So, a more complicated simulation is needed in order to prove completeness results for IsaLog.

In this paper we basically referred to a positive framework. However, it turns out that extending the language with some form of negation yields greater expressive power. It can be shown that IsaLog$^{\neq}$, that is, IsaLog allowing programs to use safe inequality atoms in bodies of rules, is able to simulate the *domain Turing machines* of [12] (which express all computable queries) still provided the input is coded in a suitable way. This simulation at least guarantees independence of the domain, because domain Turing machines have this property. It is worth noting that a rule-based language that uses positive literals and inequality atoms can only express *monotonic* queries. This is the reason why we cannot expect for IsaLog and IsaLog$^{\neq}$ the ability of expressing all computable queries. However, it is clear that IsaLog$^{\neq}$ properly subsumes IsaLog. We conjecture that IsaLog$^{\neq}$ is powerful enough to express all *monotonic computable queries*.

Extending the language with negative literals allows for *non-monotonic* queries as well. In [5] we have proposed IsaLog$^{\neg}$, that is, IsaLog extended with *isa-coherent stratified negation*, a notion of stratification that takes into account the presence of isa hierarchies. Note that, in contrast to results related to Datalog, where stratified negation is strictly weaker than negation with inflationary semantics [16, 17], it turns out that, in presence of oid invention (or an equivalent construct) stratified and inflationary negation have the same expressive power [11]. We claim that IsaLog$^{\neg}$ expresses all computable queries and all the *list-constructive queries* of [20]. A deeper analysis of the expressive power of the language will be done in a forthcoming paper [6].

5.2 Further Issues about Model Finiteness

Two important properties for IsaLog programs have been studied, namely, functionality and model finiteness. The corresponding problems of deciding functionality and finiteness

of a given IsaLog program are shown to be unsolvable in the general case. The problem of functionality has been previously studied in [1] for Datalog programs, where is also shown to be undecidable, even in presence of functional dependencies.

With respect to model finiteness, an interesting class of programs, the weakly recursive programs, has been introduced. These programs, which basically do not involve recursion through oid invention, were first defined in [13]; they always have a finite model and their model can be computed in time polynomial with respect to the number of objects in the input instance. An interesting problem consists in finding larger classes of programs, which enjoy the property of model finiteness.

Consider for example the following query, which refers to an input scheme with two classes, C_0 with type () and C_1, which is a subclass of C_0 with type $(ref : C_0)$, and a relation R of type $(ref : C_0)$:

$$\gamma_1 : C_1(\text{OID} : *, ref : X) \leftarrow C_1(\text{OID} : X, ref : Y), R(ref : X).$$

The clause involves recursion through oid invention, but it has clearly finite model whatever the input instance is, since the recursion is "bounded" to the active domain of the input instance by the presence of the atom $R(ref : X)$.

Such programs, that involve *domain bounded recursion*, and thus could be called *domain bounded recursive*, represent a larger class than weakly recursive programs. They enjoy the property of finiteness and their semantics is still tractable. Even larger classes, which correspond to less restrictive sufficient conditions could be defined [6]. An interesting problem is concerned with extending such definitions to a language where inequality atoms or negative literals are allowed. In this context, reasonable finite queries that go beyond PTIME are expressible.

Acknowledgements

The authors would like to thank Paolo Atzeni and Jan Van den Bussche for the fruitful discussions on the subject of this paper.

References

[1] S. Abiteboul and R. Hull. Data functions, Datalog and negation. In *ACM SIGMOD International Conf. on Management of Data*, pages 143–153, 1988.

[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] 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.

[4] P. Atzeni, L. Cabibbo, and G. Mecca. IsaLog: A declarative language for complex objects with hierarchies. In *Ninth IEEE International Conference on Data Engineering, Vienna*, pages 219–228, 1993.

[5] P. Atzeni, L. Cabibbo, and G. Mecca. IsaLog$^{\neg}$: a deductive language with negation for complex-objects databases with hierarchies. In *Third Int. Conf. on Deductive and Object-Oriented Databases*, pages 222–235, 1993.

[6] L. Cabibbo and G. Mecca. In preparation.

[7] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2):138–164, 1988.

[8] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Data Bases.* Springer-Verlag, 1989.

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

[10] 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.

[11] R. Hull and J. Su. Deductive query languages for recursively typed complex objects. Technical report, University of Southern California, 1989.

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

[13] 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.

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

[15] M. Kifer, R. Ramakrishnan, and A. Silbershatz. An axiomatic approach to deciding query safety in deductive databases. In *Seventh ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems*, pages 52–60, 1988.

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

[17] P.G. Kolaitis and C.H. Papadimitriou. Why not negation by fixpoint? *Journal of Comp. and System Sc.*, 43(1):125–144, August 1991.

[18] H. Lewis and C. Papadimitriou. *Elements of the Theory of Computation.* Prentice-Hall, 1981.

[19] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Sixth ACM SIGACT SIGMOD SIGART Symp. on Principles of Database Systems*, pages 237–249, 1987.

[20] 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.

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