IsaLog$^\neg$: a deductive language with negation for complex-object databases with hierarchies[*]

Paolo Atzeni, Luca Cabibbo, and Giansalvatore Mecca

Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza"
Via Salaria 113, 00198 Roma, Italy
{atzeni,cabibbo,mecca}@infokit.ing.uniroma1.it

Abstract. The IsaLog$^\neg$ model and language are presented. The model has complex objects with classes, relations, and isa hierarchies. The language is strongly typed and declarative. The main issue is the definition of the semantics of the IsaLog$^\neg$ language. The novel features are mostly due to the interaction of hierarchies with negation in the body of rules. Two semantics are presented and shown to be equivalent: a stratified semantics based on an original notion of stratification, needed in order to correctly deal with hierarchies, and a reduction to logic programming with function symbols. The solutions are based on a new technique (explicit Skolem functors) that provides a powerful tool for manipulating object identifiers.

1 Introduction

Deductive languages for complex-object databases have received a great deal of attention recently. In this context, the data model includes *classes* of *objects*, that is, sets of real world objects with the same conceptual and structural properties, and *is-a relationships*, used to organize classes in *hierarchies. Objects identifiers (oid's)* are associated with objects, to permit duplicates and to allow for object sharing and inheritance.

One interesting research direction has been the extension of Datalog$^{(\neg)}$ for the management of such a data model. The most relevant proposals in this area are the languages IQL (Abiteboul and Kanellakis [2]) and ILOG (Hull and Yoshikawa [13]), which refer to data models in the traditional database sense. Other interesting ideas have also been proposed as extensions of logic programming languages (Maier [17], Aït-Kaci and Nasr [3], Chen and Warren [12], Kifer et al. [15, 16]).

In this framework, we have recently proposed IsaLog [5], a logic programming language over a model with (flat) classes and (flat) relations, with isa relationships among classes. A distinctive feature of IsaLog is the use of *explicit*

[*] 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+". The second author is partially supported by Systems & Management S.p.A.

Skolem functors (an extension of the *implicit* Skolem functors of ILOG [13]) to deal with oid invention over hierachies of classes. A declarative semantics, a fixpoint semantics, and a reduction to ordinary logic programming with function symbols were defined and proven to be equivalent.

This paper shows how the technique of explicit Skolem functors allows for a clear definition of the semantics of oid invention, with respect to a model with hierarchies and a language with negation. The main contribution is the definition of a stratified semantics for IsaLog$^\neg$ programs, with a notion of stratification based on a partition of clauses that cannot be reduced to a partition of predicate symbols. Then, a reduction to logic programming is shown, yielding an equivalent semantics. Interestingly, this reduction would not be possible without the use of explicit Skolem functors, thus confirming their importance.

The paper is organized as follows. In Section 2 we informally present the framework and the results of the paper; examples are used to illustrate the main issues. Section 3 is devoted to the definition of the model. The language syntax is defined in Section 4. Section 5 formally presents the main results of the paper, with the definition of the semantics of IsaLog$^\neg$ programs. Conclusions and future research directions are sketched in Section 6. Proofs are omitted, due to lack of space; the reader is referred to the full paper [6].

2 Overview and Motivation

2.1 The Framework

The data model is based on a clear distinction between *scheme* and *instance*. Data is organized by means of three constructs: classes, relations, and functors. *Classes* are collections of objects; each object is identified by an *object identifier (oid)* and has an associated tuple value. *Relations* are just collections of tuples. *Functors* are mainly used to make oid invention fully declarative; for each functor the database instance contains a function from tuples to oid's. Tuples in relations, in object values, and in arguments of functions 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. Moreover, we do not require, as in other works [2], the presence of a *most specific class* for each object in the database, since this may lead to an unreasonable increase in the number of classes in the database.

The IsaLog$^\neg$ language is strongly typed and declarative, a suitable extension of Datalog [11] capable of handling oid invention and hierarchies. Three different kinds of clauses are allowed in a program:

– *relation clauses*, that is, ordinary clauses defining relations;
– *oid-invention clauses*, used to create new objects;
– *specialization clauses*, used to "specialize" oid's from superclasses to subclasses; in fact, a specialization clause can be used to specify (on the basis of some conditions) that an object in a class C also belongs to a subclass C' of C.

A program is a set of clauses that specifies a transformation from an instance of the input scheme to an instance of the output scheme.

2.2 Oid Invention

The introduction of *object identifiers (oid's)* in a declarative context gives rise to interesting semantic problems, the main one being the need for *oid invention*, that is, creation of new objects to populate extensions of classes.

The literature proposes two leading approaches to object creation: the *pure object creating* and the *logic-programming* ones. The pure object creating approach — which is adopted by some proposals, including IQL [2], LOGRES [9], and LOGIDATA+ [7] — refers to a "fact for each instance" policy: an oid-invention clause generates a new oid for each satisfiable ground instance of its body. The main drawback of such a semantics consists in the lack of a means to explicitly control the generation of objects with the same value ("duplicates").

On the other side, the so-called logic-programming approach treats new oid's in the output as terms built by means of values and oid's in the input. The ILOG language [13] proposes a semantics of invention based on *Skolem functor terms.* Skolem functors are strongly related to logic-programming function symbols; in this framework, they provide a neat syntactic tool to specify the terms on which oid invention depends. ILOG comes with a transparent skolemization mechanism, in which such terms are chosen to be exactly those occurring in the clause head. This technique allows for a nice reduction to ordinary logic-programming semantics, thus making oid invention truly declarative, but has a major shortcoming: it does not permit the generation of duplicates (when needed). Therefore, in the ILOG framework, equality implies identity, against the main motivation for the use of oid's.

Let us give an example that outlines the importance of duplicates. Assume the joint catalogue of two libraries has to be produced. Each of the libraries has no duplicate volumes and its catalogue is described by a relation R_i, with the book title as a key. If we are interested in defining the class of books, we need to collapse volumes (in different libraries) corresponding to the same book. Instead, if we want the class of all volumes, we have to retain duplicates.

The technique we propose to manage such cases consists in making *Skolem functors* become *explicit.* An *explicit Skolem functor* for an IsaLog$^{(\neg)}$ scheme is a symbol, used to build typed terms in programs. Specifically, each functor is associated with a class, and:

- explicit functors generalize implicit ones: an explicit functor term for an oid of a class C has a set of "arguments" that include the attributes of C. This is necessary in order to avoid ill-definedness of object values (that is, the generation of objects with the same oid and different values). In addition, a functor for a class may contain other arguments;
- different functors may be associated with the same class.

In this way the generation of duplicates is allowed. Consider the previous example; in both cases the catalogue is generated by means of two clauses that create objects: in the first case we have the same functor (to collapse duplicates) and in the second, two functors.

$$\begin{array}{r}
\textit{book}(\text{OID} : f_{book}(\textit{title} : x), \ldots) \leftarrow R_1(\textit{title} : x, \ldots). \\
\textit{book}(\text{OID} : f_{book}(\textit{title} : x), \ldots) \leftarrow R_2(\textit{title} : x, \ldots).
\end{array}$$

$$\begin{array}{r}
\textit{volume}(\text{OID} : f_{volume,1}(\textit{title} : x), \ldots) \leftarrow R_1(\textit{title} : x, \ldots). \\
\textit{volume}(\text{OID} : f_{volume,2}(\textit{title} : x), \ldots) \leftarrow R_2(\textit{title} : x, \ldots).
\end{array}$$

We claim that explicit functors are a very powerful tool for the manipulation of objects. In fact, not only do they provide a neat way for handling oid invention, but they also carry information about oid creation. This permits to distinguish oid's in the same class on the basis of their origin (the class itself or a subclass, for example), and to access the values that "witnessed" the invention of the oid, even if they are transparent with respect to the class. This is very useful for manipulating *imaginary objects* [1], that is, new objects computed on demand, like a relational view concerning objects instead of tuples. It is apparent that these objects exist in some classes of the view, but not in the database. When we update the database and recompute the view, we can assign the same functor (witness of an invention) to the same imaginary object. If we have stored the assignment of oid's to functor terms of previous computations, we can ensure that an object receives the same identifier every time the query is computed, so that imaginary objects maintain their identity as the database evolves [14].

2.3 Isa Hierarchies and Negation

We argued above that functors represent a nice means to manipulate oid's in the general case. Here we claim that they become almost necessary when isa and negation are included in the model. As previously sketched, the treatment of hierarchies in our model has been chosen to be the most general one, allowing for multiple and incomplete inheritance without most specific classes. Such a context reasonably requires to drop the scheme disjointness assumption, to easily deal with programs in which a subclass is newly generated and inherits objects from a superclass defined in the input instance. Moreover, it is interesting to note how hierarchies and negation interact together, requiring an *ad hoc* treatment.

Example 1. Consider a graph represented by means of two classes: *node* and *arc*, with types *()* and *(from:node, to:node)* respectively. Suppose we want to trasform the graph into a strongly connected one, adding an arc for each pair of non-connected nodes, by means of the following program (where *new-arc* ISA *arc*):

γ_1 : *path(from:x, to:y)* $\leftarrow$*node*(OID*:x), node*(OID*:y), arc*(OID*:z, from:x, to:y).*
γ_2 : *path(from:x, to:y)* $\leftarrow$*path(from:x, to:w), arc*(OID*:z, from:w, to:y).*
γ_3 : *new-arc*(OID*:*$f_{new\text{-}arc}$*(from:x, to:y), from:x, to:y)* $\leftarrow$
node(OID*:x), node*(OID*:y),* $\neg$ *path(from:x, to:y).*

The program appears to be stratified: its dependency graph contains no cycle with a negative edge. On the contrary, if we take into account hierarchies and their properties, we can argue that it is not stratified. In fact, since *new-arc* depends on (the negation of) *path* (clause γ_3), we can say that the same also holds for *arc*, since each new object in *new-arc* must also appear in *arc*. Then, since *path* depends on *arc* (clauses γ_1 and γ_2), we have a violation of the intuition behind stratification.

An intuitive proposal [9] for handling hierarchy semantics consists in the introduction of auxiliary clauses that enforce containment constraints associated with isa relationships. This means that for each pair of classes C_0 and C_1 in the scheme such that C_1 ISA C_0, we need to add a clause:

$$C_0(\text{OID} : x, A_1 : x_1, \ldots, A_k : x_k) \leftarrow C_1(\text{OID} : x, A_1 : x_1, \ldots, A_{k+m} : x_{k+m}).$$

(where $A_{k+1}, \ldots, A_{k+m}$ are the additional attributes in C_1) that forces objects in C_1 to belong to C_0 as well. These clauses, called the *isa clauses*, depend only on the scheme and not on the individual programs. The next example shows that this technique, well suited to a positive framework such as IsaLog [5], does not catch the complete meaning of negation.

Example 2. Consider the class *person* with type *(name:D, asset:D, father:person)* (where D is a *domain* of atomic values), and suppose *rich-person* ISA *person, self-made-man* ISA *rich-person.* Suppose we want to specialize people on the basis of their assets, with a special interest in rich people with a non-rich father:

γ_1 : *rich-person(*OID*:x,name:n,asset:a,father:f)* ←
*person(*OID*:x,name:n,asset:a,father:f), a > 100K.*
γ_2 : *self-made-man(*OID*:x,name:n,asset:a,father:f)* ←
*rich-person(*OID*:x,name:n,asset:a,father:f),*
¬ *rich-person(*OID*:f,name:nf,asset:af,father:ff).*

Clause γ_2 specifies the "specialization" of objects in *rich-person* to be objects in *self-made-man* as well, on the basis of some conditions that include a negation on *rich-person.* Intuitively, a natural semantics for this program is obtained by applying first (i) clause γ_1 and then (ii) clause γ_2. Essentially, step (i) computes *rich-person* and step (ii) computes *self-made-man.* Surprisingly, if the isa clauses associated with the scheme are added to the program, the resulting set of clauses is not stratified, that is, the dependency graph contains a cycle with a negative edge.

The shown examples suggest that:

– ordinary stratification [4], defined as a partition of clauses that essentially collapses to a partition of predicate symbols, fails when hierarchies are present;
– isa clauses do not represent a solution to the problem.

In the following sections we introduce an original semantics for IsaLog$^\neg$ programs, based on a notion of *isa-coherent stratification*, that is essentially a partition of clauses that cannot be reduced to a partition of predicate symbols. Then, a reduction to logic programming is shown, yielding an equivalent semantics. Interestingly, this reduction would not be possible without the use of explicit Skolem functors, thus confirming their importance.

3 The Data Model

We present only the essential aspects, omitting standard notions and details not needed in the sequel.

We fix a countable set D of *constants*, called the *domain*. An IsaLog scheme is a five-tuple $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \textsc{typ}, \textsc{isa})$, where

- $\mathbf{C}$ (the *class names*), $\mathbf{R}$ (the *relation names*), and $\mathbf{F}$ (the *functors*) are finite, pairwise disjoint sets;
- $\textsc{typ}$ is a total function on $\mathbf{C} \cup \mathbf{R} \cup \mathbf{F}$ that associates
  - a flat *tuple type* $(A_1 : \tau_1, A_2 : \tau_2, \ldots, A_k : \tau_k)$ with each class in $\mathbf{C}$ and each relation in $\mathbf{R}$; the A_i's are called the *attributes*, and each τ_i (the *type* of A_i) is either a class name in $\mathbf{C}$ or the domain D;
  - a pair (C, τ) with each functor $F \in \mathbf{F}$, where: (i) C is a class name in $\mathbf{C}$ (the *class associated with* F) and (ii) τ is a tuple type whose attributes are disjoint from those of C and of its subclasses;
- $\textsc{isa}$ is a partial order over $\mathbf{C}$, such that if $(C', C'') \in \textsc{isa}$ (usually written in infix notation, $C' \,\textsc{isa}\, C''$, and read C' is a *subclass* of C''), then $\textsc{typ}(C')$ is a *subtype* of $\textsc{typ}(C'')$ [10] (that is, for each attribute in $\textsc{typ}(C'')$, the attribute also appears in $\textsc{typ}(C')$, with the same type or (if the type is a class name) with a type that is a subclass). Multiple inheritance is allowed, with some technical restrictions, omitted for the sake of space.

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).

As in every other data model, the 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.

Following ILOG, we define instances as equivalence classes of pre-instances: pre-instances depend on actual oid's, whereas instances make oid's transparent.

A *pre-instance* $\mathbf{s}$ of an IsaLog scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \textsc{typ}, \textsc{isa})$ is a four-tuple $\mathbf{s} = (\mathbf{c}, \mathbf{r}, \mathbf{f}, \mathbf{o})$, 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 subclassing) 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 $\textsc{typ}(R)$;
- $\mathbf{o}$ is a function that associates tuples with oid's in classes, with the appropriate type [1].

[1] The definition is not trivial — with respect to other models, such as IQL [2] — because we do not require for each object a *most specific class*.

- $\mathbf{f}$ is a function that associates with each functor $F \in \mathbf{F}$ (associated with a class C) a partial injective function $\mathbf{f}(F)$ from the set of tuples over the tuple type of the functor to a subset of the oid's in $\mathbf{c}(C)$ [2].
- 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 avoids "dangling references").

Two pre-instances $\mathbf{s}_1$ and $\mathbf{s}_2$ over a scheme $\mathbf{S}$ are said to be *oid-equivalent* if there is a permutation σ of the oid's in $\mathcal{O}$ such that (extending σ to objects, tuples, and pre-instances in the natural way) it is the case that $\mathbf{s}_1 = \sigma(\mathbf{s}_2)$. An *instance* is an equivalence class of pre-instances under oid-equivalence. When needed, $[\mathbf{s}]$ will denote the instance whose representative is the pre-instance $\mathbf{s}$.

4 IsaLog$^\neg$ Syntax

Let a scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$ 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 $\mathcal{O}$, (ii) the variables in $V_{\mathbf{C}}$, and (iii) *functor terms* $F(A_1 : t_1, \ldots, A_k : t_k, A'_1 : t'_1, \ldots, A'_h : t'_h)$, where $F \in \mathbf{F}$ and $\text{TYP}(F) = (C, \tau)$, $\text{TYP}(C) = (A_1 : \tau_1, \ldots, A_k : \tau_k)$ and $\tau = (A'_1 : \tau'_1, \ldots, A'_h : \tau'_h)$.

The *atoms* of the language may have two forms (where terms in components are oid terms or value terms depending on the type associated with the attribute):

- *class atoms*: $C(\text{OID} : t_0, A_1 : t_1, \ldots, A_k : t_k)$, where C is a class name in $\mathbf{C}$, with $\text{TYP}(C) = (A_1 : \tau_1, \ldots, A_k : \tau_k)$ and t_0 is an oid 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* and *negative) 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 $C(\text{OID} : t_0, A_1 : t_1, \ldots, A_k : t_k)$, where t_0 is a functor term $F(A_1 : t_1, \ldots, A_k : t_k, \ldots)$ not occurring in $\text{BODY}(\gamma)$ and C is the class associated with F;
- a *specialization clause* if $\text{HEAD}(\gamma)$ is a class atom $C(\text{OID} : t, \ldots)$, where t is an oid term and $\text{BODY}(\gamma)$ contains (at least) a class atom $C'(\text{OID} : t, \ldots)$ such that C and C' have a common ancestor (that is, a class C_0 such that $C \text{ ISA } C_0$ and $C' \text{ ISA } C_0$).

[2] In order to guarantee well-definedness in the generation of oid's, by avoiding circularity, a partial order is defined over oid's. Moreover, ranges of functors are required to be pairwise disjoint.

Hereinafter we consider only clauses of the above three forms. A *positive clause* is a clause whose body contains only positive literals.

An IsaLog$^\neg$ *program* **P** over a scheme **S** is a set of clauses that satisfy some technical conditions: *well-typedness* (about typing of oid terms), *safety* (as usual), and *visibility* (no explicit oid's are allowed). An IsaLog *program* is an IsaLog$^\neg$ program made of a set of positive clauses.

5 Semantics of IsaLog$^\neg$ Programs

We have studied in a previous paper [5] the semantics of positive IsaLog programs, by means of three independent techniques (declarative, fixpoint, and reduction to logic programming), and shown their equivalence. The declarative semantics is based on a notion of *unique minimal model.* As opposed to what happens for the standard Datalog framework, it may be the case that the semantics for a program over an instance is undefined. There are two reasons for this: infinite generation of new objects and multiple inconsistent specialization of existing objects.

In this section we study the semantics of IsaLog$^\neg$ programs. As usual, the introduction of negation may lead to the existence of multiple incomparable minimal models, thus requiring a criterion for the selection of one of the minimal models, as the "right" semantics of a program over an instance. We follow the approach based on the notion of stratification, which howerever requires a number of variations to be used in our framework, because of the presence of isa hierarchies.

5.1 Instances as Herbrand Interpretations

In this section we briefly explain how an IsaLog instance can be represented by means of a set of facts, a preliminary tool for the description of the semantics of IsaLog$^\neg$ programs.

Given a scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \textsc{typ}, \textsc{isa})$, the *Herbrand base* $\mathrm{H}_{\mathbf{S}}$ for **S** is the set of all ground facts with predicate symbols from **R** and **C** and terms with function symbols from **F** and values from $\mathcal{O}$ and D. A *Herbrand interpretation* is a finite subset of the Herbrand base. We define a function ϕ that associates a Herbrand interpretation with each pre-instance $\mathbf{s} = (\mathbf{c}, \mathbf{r}, \mathbf{f}, \mathbf{o})$ of **S**. We proceed in two steps:

1. $\phi^0(\mathbf{s})$ is the set of facts that contains one fact for each tuple in each relation and as many facts for an object o as the number of different classes to which the object belongs. Each of these facts involves only the attributes that are relevant for the corresponding class.
2. $\phi(\mathbf{s})$ is obtained from $\phi^0(\mathbf{s})$ by recursively replacing each oid o such that o equals $\mathbf{f}(F)$ applied to $(A_1 : v_1, \ldots, A_k : v_k)$, with the term $F(A_1 : v_1, \ldots, A_k : v_k)$. Note that this replacement is univocally defined (since the

functions are injective and have disjoint ranges) and terminates (because of the partial order among oid's) [3].

The function ϕ is defined for every pre-instance but it can be shown that is not surjective: there are Herbrand interpretations that are not in the image of ϕ. This happens if one of the following conditions (here informally defined) is violated:

WT (*well-typedness*): for each fact, all terms have the appropriate type.

CON (*containment*): for each fact $C_1(\text{OID} : t_0, \ldots)$, there is a fact $C_2(\text{OID} : t_0, \ldots)$ for each class C_2 such that $C_1 \,\text{ISA}\, C_2$. This condition requires the satisfaction of the containment constraints corresponding to isa hierarchies.

DIS (*disjointness*): if two facts $C_1(\text{OID} : t_0, \ldots)$ and $C_2(\text{OID} : t_0, \ldots)$ appear, then classes C_1 and C_2 have a common ancestor in $\mathbf{S}$.

COH (*oid-coherence*): if an oid term t_0 occurs as a value for an attribute whose type is a class C, then there is a fact $C(\text{OID} : t_0, \ldots)$. This condition rules out dangling references.

FUN (*functionality*): there cannot be two different facts for the same oid term with different values for some common attributes.

Conditions FUN, CON, and DIS guarantee that oid's object values are well defined throughout hierarchies.

It can be shown that a Herbrand interpretation over a scheme $\mathbf{S}$ satisfies conditions WT, CON, DIS, COH, and FUN with respect to $\mathbf{S}$ if and only if it belongs to the image of ϕ over the pre-instances of $\mathbf{S}$.

Furthermore, ϕ preserves oid-equivalence, that is, $\phi(\mathbf{s}_1)$ and $\phi(\mathbf{s}_2)$ are oid-equivalent if and only if $\mathbf{s}_1$ and $\mathbf{s}_2$ are oid-equivalent. Therefore, we can define a function Φ that maps instances to equivalence classes of Herbrand interpretations: $\Phi : [\mathbf{s}] \mapsto [\phi(\mathbf{s})]$. Since $\phi(\mathbf{s}_1)$ is equivalent to $\phi(\mathbf{s}_2)$ only if $\mathbf{s}_1$ is equivalent to $\mathbf{s}_2$, we have that Φ is injective. So, Φ is a bijection from the set of instances to the set of equivalence classes of Herbrand interpretations that satisfy the five conditions above. The inverse of Φ is therefore defined over equivalence classes of Herbrand interpretations that satisfy conditions WT, CON, DIS, COH, and FUN.

5.2 Isa-Coherent Stratification of IsaLog$^\neg$ Programs

We need some preliminary definitions. Assume that a scheme $\mathbf{S}$ is fixed. We call *predicate symbol* of the scheme $\mathbf{S}$ every class and relation name. Given a clause γ, we say that γ *defines* a predicate symbol Q if one of the following conditions holds:

- γ is a relation clause with head $Q(\ldots)$;
- γ is an oid-invention clause with head $C(\ldots)$, with $C \in \mathbf{C}$ and $C \,\text{ISA}\, Q$;
- γ is a specialization clause with head $C(\text{OID} : t, \ldots)$, with $C \in \mathbf{C}$, $C \,\text{ISA}\, Q$, and there is no positive literal $C'(\text{OID} : t, \ldots)$ in $\text{BODY}(\gamma)$, with $C' \in \mathbf{C}$, such that $C' \,\text{ISA}\, Q$.

[3] This procedure leaves some oid's in the facts — those that, as allowed by the definition of instance, do not appear in the range of any function.

Essentially, each clause defines a predicate symbol Q if it (possibly) generates new facts that involve Q: an oid-invention clause generates a new fact for each superclass of the predicate symbol in its head; a specialization clause generates a new fact only for some superclasses (because the corresponding facts for other superclasses already exist). Clearly, this distinction is relevant only for a language with class hierarchies: in languages without hierarchies, each clause defines exactly one predicate symbol.

Given an IsaLog$^\neg$ program $\mathbf{P}$ and a predicate symbol Q, the *definition of* Q (within $\mathbf{P}$) is the set of clauses in $\mathbf{P}$ whose set of defined symbols contains Q.

A partition $P_1 \cup \ldots \cup P_n$ of the clauses of $\mathbf{P}$ is an *isa-coherent stratification* of $\mathbf{P}$ (and each P_i is a *stratum*) if the following two conditions hold for $i = 1, \ldots, n$:

1. if a predicate symbol Q occurs in a positive literal in the body of a clause $\gamma \in P_i$, then the definition of Q is contained within $\cup_{j \leq i} P_j$;
2. if a predicate symbol Q occurs in a negative literal in the body of a clause $\gamma \in P_i$, then the definition of Q is contained within $\cup_{j < i} P_j$.

An IsaLog$^\neg$ program $\mathbf{P}$ is *isa-coherently stratified* if it has an isa-coherent stratification. It should be noted that this notion is apparently the same as the usual one: the difference is inside the notion of "definition" of a symbol [4].

Isa-coherently stratified programs can be characterized by means of properties of clauses (rather than predicate symbols, as it happens in the Datalog framework). We need a few definitions. We say that a clause γ_1 *refers to* a clause γ_2 if there is a predicate symbol Q that is defined by γ_2 and occurs in a literal in the body of γ_1; if such a literal is negative, then γ_1 *negatively refers to* γ_2. Given a program $\mathbf{P}$ we define its *clause dependency graph* $\mathrm{CDG}_{\mathbf{P}}$ as a directed graph representing the relation *refers to* between the clauses of $\mathbf{P}$. An edge (γ_1, γ_2) is *negative* if γ_1 negatively refers to γ_2.

Lemma 1. *A program* $\mathbf{P}$ *is isa-coherently stratified iff its clause dependency graph* $\mathrm{CDG}_{\mathbf{P}}$ *does not contain a cycle with a negative edge.*

5.3 Fixpoint Semantics of IsaLog$^\neg$ Isa-Coherently Stratified Programs

In this section we present the fixpoint semantics for IsaLog$^\neg$ programs. The presence of isa requires a modification of the traditional approach, as follows. Given a scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$ we define the *closure* T_{ISA} *with respect to* ISA as a mapping from the powerset $2^{\mathrm{H}_{\mathbf{S}}}$ of $\mathrm{H}_{\mathbf{S}}$ to itself, as follows:

$$\begin{aligned} T_{\text{ISA}}(I_{\mathbf{S}}) = I_{\mathbf{S}} \cup \{ & C_2(\text{OID} : t_0, A_1 : t_1, \ldots, A_k : t_k) \mid \\ & C_1(\text{OID} : t_0, A_1 : t_1, \ldots, A_{k+h} : t_{k+h}) \in I_{\mathbf{S}},\ \ C_1 \text{ISA} C_2, \\ & \text{and } \text{TYP}(C_2) = (A_1 : \tau_1, \ldots, A_k : \tau_k)\} \end{aligned}$$

[4] In ordinary theory of stratification [4], the *definition* of a predicate symbol Q (within a program $\mathbf{P}$) is the set of clauses in $\mathbf{P}$ whose head's predicate symbol is Q.

The closure with respect to isa enforces containment constraints on facts associated with hierarchies, as required by condition CON defined in Section 5.1.

Then, given a set of clauses Γ over a scheme $\mathbf{S}$, we define the trasformation $T_{\Gamma,0}$ associated with Γ as a mapping from $2^{\mathbf{H}_{\mathbf{S}}}$ to itself, as follows (where the notions of substitution and satisfaction are standard):

$$T_{\Gamma,0}(I_{\mathbf{S}}) = \{\theta(\text{HEAD}(\gamma)) \mid \gamma \in \Gamma, I_{\mathbf{S}} \text{ satisfies } \theta(\text{BODY}(\gamma)), \text{ for a substitution } \theta\}$$

Then, the *immediate consequence operator* T_Γ associated with a set of clauses Γ over a scheme $\mathbf{S}$ is a mapping from $2^{\mathbf{H}_{\mathbf{S}}}$ to itself:

$$T_\Gamma(I_{\mathbf{S}}) = T_{\text{ISA}}(T_{\Gamma,0}(I_{\mathbf{S}}))$$

Let us note that in Datalog frameworks, the immediate consequence operator essentially coincides with our operator $T_{\Gamma,0}$. The application of T_{ISA} is needed here to enforce condition CON, that is, isa relationships.

Let us note that transformation T_Γ preserves all the conditions satisfied by Herbrand interpretations that correspond to pre-instances, but condition FUN, which is not in general preserved [5].

Now, the semantics of an isa-coherently stratified program can be defined following the same steps as in the traditional framework [4]. However, most properties have a significantly different proof, because of the differences in the immediate consequence operator T_Γ due to hierarchies and in the definition of stratification.

Let us consider an isa-coherently stratified program $\mathbf{P}$ over a scheme $\mathbf{S}$. Let $P_1 \cup \ldots \cup P_n$ be an isa-coherent stratification of $\mathbf{P}$, and $I_{\mathbf{S}}$ be a Herbrand interpretation. Now we define a sequence of Herbrand interpretations by putting

$$\begin{aligned} M_{0,I_{\mathbf{S}}} &= I_{\mathbf{S}} \\ M_{i,I_{\mathbf{S}}} &= T_{P_i}\Uparrow\omega(M_{i-1,I_{\mathbf{S}}}), \quad \text{for } 1 \le i \le n \end{aligned}$$

where each T_{P_i} is the operator T_Γ, for a set of clauses $\Gamma = P_i$, and $T\Uparrow\omega$ is the union of all the cumulative powers of an operator T: $T\Uparrow\omega(I) = \cup_{n\geq 0} T\Uparrow n(I)$, with $T\Uparrow 0(I) = I$ and $T\Uparrow(n+1)(I) = T(T\Uparrow n(I)) \cup T\Uparrow n(I)$. It can be shown that the last Herbrand interpretation $M_{n,I_{\mathbf{S}}}$ does not depend on the chosen stratification of $\mathbf{P}$, thus we can refer to it as $M_{\mathbf{P},I_{\mathbf{S}}}$.

We can now define the *isa-coherently stratified semantics* ST-SEM of ISALOG$^\neg$ programs, as a partial function from instances to instances, as follows. Given an isa-coherently stratified program $\mathbf{P}$ over a scheme $\mathbf{S}$, and an instance $[\mathbf{s}]$:

$$\text{ST-SEM}_{\mathbf{P}}([\mathbf{s}]) = \begin{cases} \Phi^{-1}([M_{\mathbf{P},\phi(\mathbf{s})}]) \text{ if } M_{\mathbf{P},\phi(\mathbf{s})} & \text{is finite and} \\ & \text{satisfies condition FUN} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

5.4 Reduction to Logic Programming for IsaLog$^\neg$ Programs

We have shown in a previous paper [5] that an elegant semantics for (positive) IsaLog programs can be defined as a reduction to logic programming. Given an IsaLog program $\mathbf{P}$ over a scheme $\mathbf{S}$ and an instance $[\mathbf{s}]$, this semantics is defined by means of three steps:

1. compute the Herbrand interpretation $\phi(\mathbf{s})$ associated with $\mathbf{s}$;
2. compute the minimum model $\mathcal{M}$ of the logic program composed of: (i) (a syntactic variation of) the IsaLog program $\mathbf{P}$, (ii) the set of facts $\phi(\mathbf{s})$, and (iii) the isa clauses associated with the scheme;
3. if $\mathcal{M}$ is in the image of ϕ, then let the LP-semantics of $\mathbf{P}$ over $[\mathbf{s}]$ be $\Phi^{-1}([\mathcal{M}])$ (or, equivalently, $[\phi^{-1}(\mathcal{M})]$), otherwise let it be undefined.

With respect to stratified programs, this approach is not satisfactory. Consider the program in Example 2: it has an isa-coherent stratification and thus it is isa-coherently stratified; on the other hand, the logic program obtained by adding the isa clause γ : *rich-person(*OID*:x,...)* $\leftarrow$*self-made-man(*OID*:x,...)* is not stratified in the ordinary sense. Essentially, the problem is caused by isa clauses that specify the propagation of objects from subclasses to superclasses more strongly than needed. In the example, the isa clause γ is actually needed only to support the creation of new objects, whereas it does nothing with respect to applications of clause γ_2, since γ_2 specializes in *self-made-man* objects that already belong to *rich-person*.

A solution to the problem is based on a finer specification of the propagation of objects: rather than adding isa clauses associated with a scheme, it uses additional clauses only with reference to the clauses of the program that require oid propagation. Specifically, for each clause γ that defines more than one predicate symbol, we add the following set of clauses (the *defined-symbol clauses*):

$$\begin{gathered}\{C_2(\text{OID} : t_0, A_1 : x_1, \ldots, A_k : x_k) \leftarrow \text{BODY}(\gamma) \mid \\ \text{HEAD}(\gamma) = C_1(\text{OID} : t_0, A_1 : x_1, \ldots, A_{k+m} : x_{k+m}) \\ C_1 \text{ ISA } C_2,\ C_2(\neq C_1) \text{ is a defined symbol of } \gamma\}\end{gathered}$$

Therefore, we have two different reductions to logic programming. We call them the *isa-clause (IC) reduction* and the *defined-symbol (DS) reduction*. We can therefore define two logic programming semantics for IsaLog$^\neg$ programs (possibly with negation), the *IC-semantics* and the *DS-semantics*, respectively. It is convenient to define them in three steps again (where the first and third coincide with the analogous for positive programs):

1. compute $\phi(\mathbf{s})$;
2. compute the perfect model $\mathcal{M}$ (in the standard logic programming sense) of the logic program composed of: (i) the IsaLog$^\neg$ program $\mathbf{P}$, (ii) $\phi(\mathbf{s})$, and (iii) the isa clauses associated with the scheme (for the IC-semantics) or the defined-symbol clauses (for the DS-semantics);
3. if $\mathcal{M}$ is in the image of ϕ, then let the (IC- or DS-) semantics of $\mathbf{P}$ over $[\mathbf{s}]$ be $\Phi^{-1}([\mathcal{M}])$, otherwise let it be undefined.

We have the following results, which relate the two logic programming semantics with each other and with the fixpoint semantics. The first theorem shows that the DS-reduction can always be used instead of IC-reduction — the converse does not hold, as we argued with respect to the example above. As a consequence, the two semantics are equivalent with respect to positive programs. The second (and more important) theorem states the equivalence of the DS-semantics and the stratified semantics.

Theorem 2. *For every* ISALOG$^\neg$ *program, if the IC-reduction is stratified, then the DS-reduction is also stratified, and the IC-semantics and the DS-semantics coincide.*

Theorem 3. *For every* ISALOG$^\neg$ *program* **P**

- *the DS-reduction of* **P** *is stratified if and only if* **P** *is isa-coherently stratified;*
- *the isa-coherently stratified semantics* $\text{ST-SEM}_{\mathbf{P}}$ *and the DS-semantics of* **P** *coincide.*

It is worth noting that the DS-reduction is heavily based on explicit Skolem functors. Let us argue by means of an example. Assume we have a scheme with the isa relationship between the classes *person* and *student*, whose respective type is *(name:D)* and *(name:D,id-no:D)*, and a program with an oid-invention clause:

$$\gamma : \textit{student}(\text{OID} : f_{student} : \textit{(name:n,id-no:id), name:n,id-no:id}) \leftarrow \text{BODY}(\gamma).$$

The DS-reduction introduces a clause

$$\gamma' : \textit{person}(\text{OID} : f_{student} : \textit{(name:n,id-no:id), name:n}) \leftarrow \text{BODY}(\gamma).$$

with the same body and the same functor term in the head. The use of the same functor guarantees that in each pair of facts generated by these clauses the oid is the same, and so they refer to the same object. This behavior could hardly be introduced without explicit functors. As a matter of fact, if implicit functors were to be used, the clause would produce two different functor terms for the two classes, thus generating different oid's.

6 Conclusions and Future Work

Several issues need to be further investigated, as follows.

- The characterization of the definedness of the semantics of programs over instances; more specifically, since functors possibly induce model infiniteness, we need to find (necessary and) sufficient conditions for a set of clauses to have a finite model.
- The management of integrity constraints, especially with regard to their preservation in derived data.
- The introduction of recursive methods attached to classes of the scheme.

References

1. S. Abiteboul and A. Bonner. Objects and views. In *ACM SIGMOD International Conf. on Management of Data*, pages 238–247, 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. H. Aït-Kaci and R. Nasr. LOGIN a logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:185–215, 1986.
4. K. Apt, H. Blair, and A. Walker. Toward a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kauffman, Los Altos, 1988.
5. 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*, 1993.
6. P. Atzeni, L. Cabibbo, and G. Mecca. IsaLog$^{\neg}$: a deductive language with negation for complex-objects databases with hierarchies. Technical report, Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza", 1993.
7. P. Atzeni and L. Tanca. The LOGIDATA+ model and language. In *Next Generation Information Systems Technology, Lecture Notes in Computer Science 504*. Springer-Verlag, 1991.
8. C. Beeri. A formal approach to object-oriented databases. *Data and Knowledge Engineering*, 5:353–382, 1990.
9. F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object oriented data modelling with a rule-based programming paradigm. In *ACM SIGMOD International Conf. on Management of Data*, pages 225–236, 1990.
10. L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76(2):138–164, 1988.
11. S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Data Bases*. Springer-Verlag, 1989.
12. 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.
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. M. Kifer, W. Kim, and Y. Sagiv. Querying object-oriented databases. In *ACM SIGMOD International Conf. on Management of Data*, pages 393–402, 1992.
15. 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.
16. 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.
17. D. Maier. A logic for objects. In *Workshop on Foundations of Deductive Database and Logic Programming (Washington, D.C. 1986)*, pages 6–26, 1986.

This article was processed using the LaTeX macro package with LLNCS style