IsaLog: A declarative language for complex objects with hierarchies *

P. Atzeni, L. Cabibbo, G. Mecca
Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113 — 00198 Roma, Italy

Abstract

The IsaLog *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 language, given in three different ways, shown to be equivalent: a model-theoretic semantics, a reduction to logic programming with function symbols, and a fixpoint semantics. Each of the semantics presents new aspects with respect to existing proposals, because of the interaction of oid-invention with general isa hierarchies. The solutions are based on a new technique, explicit Skolem functors, which provide a powerful tool for manipulating object-identifiers.*

1 Introduction

There has been a great deal of attention towards object-oriented databases in the last years. The main directions of research are concerned with the development of data models that extend the well known relational model allowing *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* (that is, disjoint taxonomies). *Objects identifiers (oid's)* are associated with objects, to permit duplicates and to allow for object sharing and inheritance.

At the same time, in order to achieve better flexibility and expressiveness, the integration of declarative languages in this framework has been pursued, since they seem to provide a nice way to express data retrieval, updates, and integrity constraints. The main semantic matter connected with the declarative manipulation of objects is the need for *oid-invention* [1]. Oid-invention clauses have a slightly different behavior from ordinary Horn clauses, since universal quantification of the variables is not satisfactory.

In this paper we present IsaLog, a model with objects and isa hierarchies along with a strongly typed language that has a declarative semantics for oid-invention. The language makes use of *explicit Skolem functors* in order to control the generation of duplicates and the well definedness of object values throughout hierarchies.

The main contribution of this paper is the definition of the semantics of the IsaLog language. Specifically, we propose three different semantics and show their equivalence. The first semantics is purely declarative and is based on the notion of a *model*. To the best of our knowledge, this is the first extension of the model-theoretic semantics of Datalog to a framework with classes and hierarchies. The second semantics is based on a reduction to logic programming, following and integrating two independent approaches: ILOG [9] (in the management of functors) and Logres [6] (in dealing with hierarchies). The main step in this approach is the introduction of auxiliary clauses that enforce the containment constraints associated with isa relationships. It can be shown that this technique does not catch the complete meaning of isa. In particular, when negation is allowed, programs with a reasonable model seem to be not stratified in the traditional sense. Finally, we provide a *fixpoint* semantics, based on a trasformation that computes the *closure* of a set of facts *with respect to isa*. In a forthcoming paper [3] we will suggest a new notion of stratification that provides a solution to the problem outlined.

This work originates from the papers about ILOG [9] and the so called alphabet logics [11, 13]. Many of the issues are inherited from the LOGIDATA+ model and language [4] and are therefore similar in spirit to

*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 first author is now with Terza Università di Roma. The second author is partially supported by Systems & Management S.p.A.

those of the IQL and Logres languages [1, 6].

The paper is organized as follows. In Section 2 we informally introduce the data model and the language, and briefly discuss some examples. Section 3 is concerned with the formal definition of the data model. The syntax of the language is presented in Section 4. Section 5 introduces the declarative semantics of IsaLog programs, and gives the definition of *model* of a program over an instance. Finally, in Section 6 several possible reductions to logic programming are discussed and in Section 7 the fixpoint semantics is proposed. Then, the equivalence of the various semantics is shown. For the sake of space, proofs of results are omitted.

2 Overview and motivation

The data model is based on a clear distinction between *scheme* and *instance*. Data is organized by means of three constructs:

- *Classes:* collections of objects; each object is identified by an *object identifier (oid)* and has an associated tuple value.
- *Relations:* collections of tuples.
- *Functors*, mainly used to make oid-invention fully declarative. Each functor has an associated function from tuples to oid's that is stored in the instance. This has been done in order to keep oid's in the instance, while making functors transparent. In this way we can keep track of the generation of each oid through the associated functor term.

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 [1], the presence of a *most specific class* for each object of the database, since this usually leads to an unreasonable increase in the number of classes of the database. For example, given the class containing all the *persons*, and two subclasses containing the *married-persons* and the *students*, respectively, with a non empty intersection, the most specific class requirement would impose a class *married-students*, even when it is not really significant in the application.

The IsaLog language is declarative, a suitable extension of Datalog [8] capable of handling oid-invention and hierarchies. 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. Coherently with the presence of isa, we do not require disjointness between input and output schemes (in contrast with other approaches, from Datalog [8, 15] to IQL [1]).

A fundamental feature of the IsaLog model and language is the original use of explicit Skolem functors. The motivation for this technique arises from a marked difference between value-based and oid-based data models. Let us give an example (where D is a *domain* of atomic constants, including strings and integers): given a relation *fatherhood*, with type *(father:D, child:D)*, and a relation *motherhood*, with type *(mother:D, child:D)*, assume we want to build the class *couple*, with type *(father:D, mother:D)*, that contains all the couples of parents. Intuitively, we could use the following clause:

*couple(*OID*:x, father:f, mother:m)* $\leftarrow$
fatherhood(father:f, child:c),
motherhood(mother:m,child:c)

The clause generates an object for each couple, with the associated oid, which has to be different from those already used in the database. Clearly, every variable occurring in the body is supposed to be universally quantified, whereas the variable x, representing the oid to be created, should rather be existentially quantified. The real problem arises when trying to establish the correct order among quantifiers: the reasonable semantics of the clause suggests the following order:

$(\forall f\ (\forall m\ (\exists x\ (\forall c$ *(fatherhood(father:f, child:c)* $\wedge$
motherhood(mother:m, child:c)
$\rightarrow$ *couple(*OID*:x, father:f, mother:m))))))*

Unfortunately, there is no way to enforce such a semantics without explicit syntactic tools, so that the only feasible solution is to interpret the clause as if the existential quantifier were at the end of the list (this happens, for example, in IQL [1] and in Logres [6]). In this case, the clause would create a duplicate of the same couple for each of their children. To avoid such an undesirable behaviour, in [9] an automatic form of Skolemization was proposed, that is, a sort of pre-processing of clauses in order to substitute each variable representing an oid to be created with an *implicit Skolem term*, containing exactly the same arguments as the value associated with the oid. In the example, the Skolemized version of the clause would be:

*couple (*OID*:F(father:f, mother:m),*

father:f, mother:m) ←
fatherhood(father:f, child:c),
motherhood(mother:m,child:c)

In this way the correct semantics is achieved. Anyway, this approach has important shortcomings as well. The main problem is that implicit functors never allow for duplicates, that is, different objects with the same values. This is a strong drawback in an object-oriented framework, since object identity has been introduced to avoid the value based identification mechanism used in traditional models: since each object can be univocally identified by means of its oid, different objects may have the same values. As a tool for the solution of this problem, we propose explicit functors, typed structures declared in the scheme that generalize implicit functors: an explicit functor term for an oid of a class C has at least the same attributes as the objects 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 attributes, and a class may have different functors. In this way a controlled generation of duplicates is allowed.

Let us give an example: suppose we have two electrical networks, made of resistors and capacitors. Each circuit is described by a relation containing the coordinates of each component along with its type (either *resistor* or *capacitor*), its value and a conventional name that identifies the component. We are interested in deriving an "abstract" representation of the complete network, obtained by merging the partial ones. By "abstract" representation we mean the topological representation of the components regardless of node coordinates. We use a quite informal and self-explanatory syntax in the description of types.

relation circuit1 : (Xfrom:D, Yfrom:D, Xto:D, Yto:D,
name:D, type:D, value:D);
relation circuit2 : (Xfrom:D, Yfrom:D, Xto:D, Yto:D,
name:D, type:D, value:D);
class node : ();
functor F_{node} *: (node, (X:D, Y:D));*
class component : (from:node, to:node);
class resistor ISA component: (resistance:D);
functor F_{R1} *: (resistor, (name:D));*
functor F_{R2} *: (resistor, (name:D));*
class capacitor ISA component : (capacitance:D);
functor F_{C1} *: (capacitor, (name:D));*
functor F_{C2} *: (capacitor, (name:D));*

Note how each functor has a class associated with it. Moreover, a single class may have different functors, intuitively used to generate objects of different origin. The class of the nodes of the first network can be generated in the following way:

*node(*OID*:*F_{node}*(X:x1, Y:y1))* ←
circuit1(Xfrom:x1, Yfrom:y1, Xto:x2, Yto:y2,
name:n, type:t, value:r)

*node(*OID*:*F_{node}*(X:x2, Y:y2))* ←
circuit1(Xfrom:x1, Yfrom:y1, Xto:x2, Yto:y2,
name:n, type:t, value:r)

The same has to be done with respect to the second circuit; note how we make use of only one functor, since in both circuits nodes can be identified by means of their coordinates.

At this point we have obtained an object for each of the nodes of the networks. The generation of the components it is not difficult. For the sake of space, we will consider only the generation of components in the *resistor* class. Since each resistor is not univocally identified by its name in the global circuit, we have to resort to a couple of functors associated with the *resistor* class, one for each partial circuit.

*resistor(*OID*:*F_{R1}*(..., name:n),*
from:F_{node}*(X:x1, Y:y1),*
to:F_{node}*(X:x2, Y:y2), resistance:r)* ←
circuit1(Xfrom:x1, Yfrom:y1, Xto:x2, Yto:y2,
name:n, type:'resistor', value:r),
node(F_{node}*(X:x1,Y:y1)), node(*F_{node}*(X:x2, Y:y2))*

*resistor(*OID*:*F_{R2}*(..., name:n),*
from:F_{node}*(X:x1, Y:y1),*
to:F_{node}*(X:x2, Y:y2), resistance:r)* ←
circuit2(Xfrom:x1, Yfrom:y1, Xto:x2, Yto:y2,
name:n, type:'resistor', value:r),
node(F_{node}*(X:x1,Y:y1)), node(*F_{node}*(X:x2, Y:y2))*

We believe that explicit functors are a very powerful tool when manipulating objects. In fact, not only do they provide a neat way for handling oid inventions, 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.

3 The Data Model

We fix a countable set $\mathcal{L}$ of *labels* or *identifiers*, a countable set D of *constants*, called the *domain*, and a countable set $\mathcal{O}$ of *object identifiers (oid's)*, which are pairwise disjoint.

An IsaLog scheme is a five-tuple $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$, where

- $\mathbf{C}$ (the *class names*), $\mathbf{R}$ (the *relation names*), $\mathbf{F}$ (the *functors*) are finite, pairwise disjoint sets;
- 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;
- ISA is a partial order over $\mathbf{C}$, such that if $(C', C'') \in$ 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* of $\text{TYP}(C'')$, where a type τ' is a *subtype* [7] of a type τ'' (in symbols $\tau' \preceq \tau''$) if one of the following conditions holds:
  1. $\tau' = \tau'' = D$;
  2. $\tau', \tau'' \in \mathbf{C}$ and $\tau' \text{ISA} \tau''$;
  3. τ' and τ'' are both tuple types, $\tau' = (A'_1 : \tau'_1, A'_2 : \tau'_2, \ldots, A'_k : \tau'_k)$ $\tau'' = (A''_1 : \tau''_1, A''_2 : \tau''_2, \ldots, A''_h : \tau''_h)$ and for each $j \in \{1, 2, \ldots, h\}$ there is an $i \in \{1, 2, \ldots, k\}$ such that $A'_i = A''_j$ and $\tau'_i \preceq \tau''_j$.

  Moreover, if C' and C'' have a *common ancestor* (that is, there is a class C_0 such that $C' \text{ISA} C_0$ and $C'' \text{ISA} C_0$) and a common attribute A, then there is a common ancestor C of C' and C'' (that may coincide with C_0) such that A is an attribute of C.

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). If we add two special types $\tau_\top$ (the *top* type) and $\tau_\bot$ (the *bottom* type), such that for every type τ it is the case that $\tau \preceq \tau_\top$ and $\tau_\bot \preceq \tau$, then the notion of subtyping induces a lattice over the types of $\mathbf{S}$.

As in every other data model, the scheme gives the structure of the possible *instances* of the database. As a first step in the definition of instance, let us define for each type τ, the associated *value-set* $\text{VAL}(\tau)$, that is, the set of its possible values: (i) if $\tau = D$, then $\text{VAL}(\tau)$ is the domain D; (ii) if τ is a class name $C \in \mathbf{C}$, then its value-set is the set of the oid's $\mathcal{O}$; (iii) if τ is a tuple type, then $\text{VAL}(\tau)$ is the set of all possible tuples over τ, where a *tuple* (as in other formal frameworks) is a function from the set of attributes to the union of value-sets of the component types, with the restriction that each value belongs to the value-set of the corresponding type.

Now we introduce the notion of *refinement*, defined over values, which is the natural counterpart of subtyping, defined over types. With respect to values of simple types, refinement coincides with equality, so the definition is really significant with respect to tuples: a tuple t_1 is a *refinement* of a tuple t_2, if the type of t_1 is a subtype of the type of t_2 and the restriction of t_1 to the attributes of t_2 (the projection, in relational database terminology) equals t_2.

With respect to classes, it is important to note that their value-sets contain only oid's. In the definition of *instance* below, we will show how actual values are associated with oid's. In this way, it is possible to implement indirect references to objects and other features such as object sharing. Also, for each class, the value-set is the set of *all* possible oid's: essentially, we can say that oid's are not typed, and so they allow the identification of an object regardless of its type (this is an oft-cited requirement for object oriented systems [10, 14]).

Following ILOG [9], we define instances as equivalence classes of pre-instances, where pre-instances depend on actual oid's, whereas instances make oid's transparent.

A *pre-instance* $\mathbf{s}$ of an IsaLog scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$ 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: $\mathbf{c}(C) \subseteq \mathcal{O}$, with the following conditions: (i) if $C' \text{ISA} C''$, then $\mathbf{c}(C') \subseteq \mathbf{c}(C'')$; (ii) if $\mathbf{c}(C') \cap \mathbf{c}(C'') \neq \emptyset$, then C' and C'' have a common ancestor. These conditions have two consequences. First, if ISA is the identity relation (and so there are no non-trivial subset constraints) than the extensions of the classes are pairwise disjoint, as it is usually assumed in other frameworks that do not consider hierarchies [1, 9]. Second, multiple inheritance is allowed only beneath a common ancestor: if there are classes $C, C', C'' \in \mathbf{C}$ such that $C \text{ISA} C'$ and $C \text{ISA} C''$, then C' and C'' have a common ancestor.
- $\mathbf{r}$ 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, as follows. For each $o \in \mathcal{O}$, let us consider the set of classes that contain o: $\text{CLASSES}(o) = \{C \mid C \in \mathbf{C},\ o \in \mathbf{c}(C)\}$. Then for each o, if $\text{CLASSES}(o)$ is empty, then $\mathbf{o}(o)$ is undefined, otherwise it is a value from the value set of the tuple type that is the greatest lower bound (according to the lattice induced by subtyping) of the types of the classes in $\text{CLASSES}(o)$.

- $\mathbf{f}$ is a function that associates with each $F \in \mathbf{F}$ a function $\mathbf{f}(F)$ as follows. Let $\text{TYP}(F) = (C, \tau)$, $\text{TYP}(C) = (A_1 : \tau_1, \ldots, A_k : \tau_k)$ and $\tau = (A'_1 : \tau'_1, \ldots, A'_h : \tau'_h)$. Then $\mathbf{f}(F)$ is a (partial) injective function from the value set of the tuple type $(A_1 : \tau_1, \ldots, A_k : \tau_k, A'_1 : \tau'_1, \ldots, A'_h : \tau'_h)$ to a subset of $\mathbf{c}(C)$. The functions corresponding to the various functors are required to satisfy the following conditions: (i) the ranges are pairwise disjoint, (ii) a partial order $\leq$ is defined among the oid's in such a way that if the oid o_1 is the result of the application of a function to a tuple that involves the oid o_2, then $o_1 < o_2$ (that is, $o_1 \leq o_2$ and $o_1 \neq o_2$);
- 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 *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 Syntax of the Language

Let a scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$ be fixed. Also, let V_D and $V_{\mathbf{C}}$ be disjoint countable sets of *variables*, used to denote constants from the domain D and object identifiers, respectively. The elements of V_D are called *value-variables* and those in $V_{\mathbf{C}}$, *oid-variables*.

The *terms* of the language are

1. *value-terms*, which are of two forms: (i) the constants in D and (ii) the variables in V_D;
2. *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 $\tau = (A'_1 : \tau'_1, \ldots, A'_h : \tau'_h)$, and each t_i (t'_j) is a constant or an oid-term depending on whether τ_i (τ'_j) is the domain D or a class $C' \in \mathbf{C}$.

The *atoms* of the language may have two forms:

1. *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 $\text{TYP}(C) = (A_1 : \tau_1, \ldots, A_k : \tau_k)$, t_0 is an oid-term, and for each $i \in \{1, 2, \ldots, k\}$, t_i is a value term (if $\tau_i = D$) or an oid-term (if $\tau_i = C'$, with $C' \in \mathbf{C}$).
2. *relation-atoms*: $R(A_1 : t_1, \ldots, A_k : t_k)$, where R is a relation name in $\mathbf{R}$, with type $\text{TYP}(R) = (A_1 : \tau_1, \ldots, A_k : \tau_k)$, and for each $i \in \{1, 2, \ldots, k\}$, t_i is a value term (if $\tau_i = D$) or an oid-term (if $\tau_i = C'$, with $C' \in \mathbf{C}$).

The class name or relation name in an atom is called the *predicate symbol* of the atom.

Given an atom L, we say that an oid-term t *ranges over a class* C in L if one of the following conditions holds:

- L is a relation-atom $R(\ldots, A : t, \ldots)$ and the type of the attribute A in R is the class C;
- L is a class-atom $C'(\text{OID} : t_0, A_1 : t_1, \ldots, A_k : t_k)$ and
  - $t_0 = t$ and $C' = C$, or
  - $t_0 = F(\ldots, A : t, \ldots)$ and the type of A in F is the class C, or
  - $t_i = t$ and the type of A_i in C' is the class C for some $i \in \{1, \ldots, k\}$.

A *rule* has the form: $r : L_0 \leftarrow L_1, L_2, \ldots, L_p$, where r is the *name* of the rule (often omitted), $L_0, L_1, \ldots, L_p$ (with $p > 0$) are atoms. A *fact* is a ground atom (that is, without variables). A *clause* is a rule or a fact. Given a clause γ, it is convenient to define its head and body, denoted with $\text{HEAD}(\gamma)$ and $\text{BODY}(\gamma)$, respectively. If γ is a rule $L_0 \leftarrow L_1, L_2, \ldots, L_p$, then $\text{HEAD}(\gamma) = L_0$ and $\text{BODY}(\gamma) = \{L_1, \ldots, L_p\}$. If γ is a fact L_0, then $\text{HEAD}(\gamma) = L_0$ and $\text{BODY}(\gamma)$ is the empty set. Let us introduce 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 $C(\text{OID} : t_0, \ldots)$, where t_0 is a functor term $F(\ldots)$ 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.

Moreover, we say that a clause γ is:

- *well-typed* if whenever an oid-term t ranges in $\text{HEAD}(\gamma)$ over a class C it is the case that
  - γ is a specialization-clause or an oid-invention clause and $\text{HEAD}(\gamma) = C(\text{OID} : t, \ldots)$ or
  - there is an atom in $\text{BODY}(\gamma)$ in which t ranges over a class C' such that $C' \text{ ISA } C$;
- *safe* if each variable in $\text{HEAD}(\gamma)$ occurs in $\text{BODY}(\gamma)$ as well;

- *visible* if it does not contain oid's.

An IsaLog *program* $\mathbf{P}$ over a scheme $\mathbf{S}$ is a set of oid-invention-clauses, specialization-clauses, and relation-clauses that are well-typed, safe and visible.

5 Declarative Semantics

Let $\mathbf{P}$ be a program over an IsaLog scheme $\mathbf{S}$. A *substitution* θ is a total function from variables to terms that maps variables in $V_{\mathbf{C}}$ to oid-terms and variables in V_D to constants. A substitution θ is *simple* if $\theta(x)$ is an oid or a constant for every variable x. Given a simple substitution θ, a preinstance $\mathbf{s}$ over $\mathbf{S}$, and a term t, the *instantiation* $\text{INST}_{\theta,\mathbf{S}}$ induced by θ is a partial function from terms to ground terms that applies θ and the functions corresponding to functors, thus recursively replacing functor terms with oid's, defined as follows:

- if t is a constant or an oid, then $\text{INST}_{\theta,\mathbf{S}}(t) = t$;
- if t is a variable x, then $\text{INST}_{\theta,\mathbf{S}}(x) = \theta(x)$;
- if t is a functor term $F(B_1 : t_1, \ldots, B_p : t_p)$, then consider the tuple

  $$t' = (B_1 : \text{INST}_{\theta,\mathbf{S}}(t_1), \ldots, B_p : \text{INST}_{\theta,\mathbf{S}}(t_p))$$

  obtained by recursively applying $\text{INST}_{\theta,\mathbf{S}}$ to the terms $t_1, \ldots, t_p$, and the function $\mathbf{f}(F)$. If $\mathbf{f}(F)$ is defined over t', then $\text{INST}_{\theta,\mathbf{S}}(t)$ equals the value of $\mathbf{f}(F)$ over t', otherwise it is not defined.

The notion of instantiation is extended in the natural way to atoms and sets of atoms (and so to bodies of rules).

Given a simple substitution θ and an atom L, we say that a pre-instance $\mathbf{s} = (\mathbf{c}, \mathbf{r}, \mathbf{f}, \mathbf{o})$ *satisfies* $\text{INST}_{\theta,\mathbf{S}}(L)$ if

- L is a relation-atom $R(A_1 : t_1, \ldots, A_k : t_k)$ and $(A_1 : \text{INST}_{\theta,\mathbf{S}}(t_1), \ldots, A_k : \text{INST}_{\theta,\mathbf{S}}(t_k))$ is a tuple in the relation $\mathbf{r}(R)$;
- L is a class-atom $C(\text{OID} : t_0, A_1 : t_1, \ldots, A_k : t_k)$, $\text{INST}_{\theta,\mathbf{S}}(t_0)$ is an oid o in $\mathbf{c}(C)$, and $\mathbf{o}(o)$ is a refinement of $(A_1 : \text{INST}_{\theta,\mathbf{S}}(t_1), \ldots, A_k : \text{INST}_{\theta,\mathbf{S}}(t_k))$.

This definition differs from the usual notion of satisfaction in two aspects (the first due to classes and functors and the second to hierarchies): (i) the use of instantiation instead of substitution; and (ii) the weaker requirement on values of objects, refinement rather than equality.

A pre-instance $\mathbf{s}$ *satisfies* a clause γ if, for each simple substitution θ such that $\mathbf{s}$ satisfies $\text{INST}_{\theta,\mathbf{S}}(\text{BODY}(\gamma))$, it is the case that $\mathbf{s}$ also satisfies $\text{INST}_{\theta,\mathbf{S}}(\text{HEAD}(\gamma))$.

Given a program $\mathbf{P}$ over a scheme $\mathbf{S}$ and a pre-instance $\mathbf{s}_0 = (\mathbf{c}_0, \mathbf{r}_0, \mathbf{f}_0, \mathbf{o}_0)$ of $\mathbf{S}$, we say that a pre-instance $\mathbf{s} = (\mathbf{c}, \mathbf{r}, \mathbf{f}, \mathbf{o})$ of $\mathbf{S}$ is a *pre-model* for $\mathbf{P}$ over $\mathbf{s}_0$ if

1. $\mathbf{s}$ is an *extension* of $\mathbf{s}_0$, defined as follows: (i) for each relation name $R \in \mathbf{R}$, the relation $\mathbf{r}(R)$ is a superset of the relation $\mathbf{r}_0(R)$; (ii) for each class name $C \in \mathbf{C}$, the set of oid's $\mathbf{c}(C)$ is a superset of $\mathbf{c}_0(C)$, and, for each oid $o \in \mathbf{c}_0(C)$, $\mathbf{o}(o)$ is a refinement of $\mathbf{o}_0(o)$; (iii) for each functor $F \in \mathbf{F}$, if $\mathbf{f}_0(F)$ is defined over a tuple t, then $\mathbf{f}(F)$ is also defined over t and has the same value.
2. $\mathbf{s}$ satisfies each clause in $\mathbf{P}$.

We have two results.

Lemma 1 *Let $\mathbf{P}$ be a program over a scheme $\mathbf{S}$. If the pre-instance $\mathbf{s}$ is a pre-model for $\mathbf{P}$ over the pre-instance $\mathbf{s}_0$, then (i) for each pre-instance $\mathbf{s}'_0$ oid-equivalent to $\mathbf{s}_0$, there is a pre-model $\mathbf{s}'$ for $\mathbf{P}$ over $\mathbf{s}'_0$, such that $\mathbf{s}'$ is oid-equivalent to $\mathbf{s}$; and (ii) for each pre-instance $\mathbf{s}'$ oid-equivalent to $\mathbf{s}$, there is a pre-instance $\mathbf{s}'_0$ oid-equivalent to $\mathbf{s}_0$ such that $\mathbf{s}'$ is pre-model for $\mathbf{P}$ over $\mathbf{s}'_0$.*

Because of Lemma 1 we can give a definition of *model* with reference to instances based on the definition of pre-model. An instance $[\mathbf{s}]$ is a *model* for a program $\mathbf{P}$ over an instance $[\mathbf{s}_0]$ if $\mathbf{s}$ is a pre-model for $\mathbf{P}$ over $\mathbf{s}_0$.

Lemma 2 *Let $\mathbf{S}$ be a scheme. If a pre-instance $\mathbf{s}$ is an extension of a pre-instance $\mathbf{s}_0$, then (i) for each pre-instance $\mathbf{s}'_0$ oid-equivalent to $\mathbf{s}_0$, there is a pre-instance $\mathbf{s}'$ oid-equivalent to $\mathbf{s}$ such that $\mathbf{s}'$ is an extension of $\mathbf{s}'_0$; and (ii) for each pre-instance $\mathbf{s}'$ oid-equivalent to $\mathbf{s}$, there is a pre-instance $\mathbf{s}'_0$ oid-equivalent to $\mathbf{s}_0$ such that $\mathbf{s}'$ is an extension of $\mathbf{s}'_0$.*

As a consequence, the notion of extension, originally defined for pre-instances, becomes meaningful also for instances. That is, given two pre-instances $\mathbf{s}$ and $\mathbf{s}_0$, if $\mathbf{s}$ is an extension of $\mathbf{s}_0$, then we can say that the instance $[\mathbf{s}]$ is an *extension* of the instance $[\mathbf{s}_0]$.

A model $[\mathbf{s}]$ for $\mathbf{P}$ over $[\mathbf{s}_0]$ is *minimal* if there is no other model $[\mathbf{s}']$ for $\mathbf{P}$ over $[\mathbf{s}_0]$ such that $[\mathbf{s}]$ is an extension of $[\mathbf{s}']$. If there is only one minimal model, then we call it the *minimum model.*

Theorem 1 *Let $\mathbf{P}$ be a program over a scheme $\mathbf{S}$, and let $[\mathbf{s}]$ be an instance of $\mathbf{S}$. Then either there is no model for $\mathbf{P}$ over $[\mathbf{s}]$ or there exists a minimum model.*

Let $\mathbf{P}$ be a program over a scheme $\mathbf{S}$. The *declarative semantics* of $\mathbf{P}$ is a partial function $\text{D-SEM}_\mathbf{P}$ from instances of $\mathbf{S}$ to instances of $\mathbf{S}$: $\text{D-SEM}_\mathbf{P}([\mathbf{s}])$ equals the minimum model of $\mathbf{P}$ over $[\mathbf{s}]$ if it exists, and is undefined otherwise.

There can be various reasons for which there is no model for a program $\mathbf{P}$ over an instance $[\mathbf{s}]$ (and therefore the declarative semantics is not defined). They correspond to various extensions of the model and language with respect to the traditional Datalog framework, where minimum models always exist.

- Recursion through oid invention can lead to the generation of infinite sets of facts. For example, given a class C, whose tuple type is $(C_{ref} : C)$, and the rule $\gamma : C(\text{OID} : F(C_{ref} : x), C_{ref} : x) \leftarrow C(\text{OID} : x, C_{ref} : y)$, the program made of this rule has clearly no model unless the class C is empty in the input instance.
- 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. Consider the following scheme:

  class person : (name:D);
  class husband ISA person : (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(*OID*:x, name:h, spouse:y)* $\leftarrow$
  marriage(husband:x, wife:y),
  *person(*OID*:x, name:h), person(*OID*:y, name:w)*

  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 rule has clearly no model.

6 Reduction to logic programming

We give an alternative semantics based on a reduction to logic programming with function symbols [12].

As a preliminary, we briefly explain how an IsaLog instance can be represented by means of a set of facts. Let $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$ be an IsaLog scheme. The *Herbrand universe* $\text{U}_\mathbf{S}$ for $\mathbf{S}$ is the set of all ground terms of $\mathbf{S}$. The *Herbrand base* $\text{H}_\mathbf{S}$ for $\mathbf{S}$ is the set of all facts of the language. A *Herbrand interpretation* $I_\mathbf{S}$ is a finite subset of $\text{H}_\mathbf{S}$.

Given a scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$, 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 $\mathbf{S}$. We proceed in two steps:

1. Let $\phi^0(\mathbf{s})$ be the set of facts that contains: (i) a fact $R(A_1 : v_1, \ldots, A_k : v_k)$, for each $R \in \mathbf{R}$ and each tuple $(A_1 : v_1, \ldots, A_k : v_k)$ in the relation $\mathbf{r}(R)$; (ii) a fact $C(\text{OID} : o, A_1 : v_1, \ldots, A_k : v_k)$, for each $o \in \mathcal{O}$ and for each class $C \in \text{CLASSES}(o)$, where $A_1, \ldots, A_k$ are the attributes of C and $(A_1 : v_1, \ldots, A_k : v_k)$ is the restriction of $\mathbf{o}(o)$ to $A_1, \ldots, A_k$. In plain words, $\phi^0(\mathbf{s})$ contains one fact for each tuple in each relation and as many facts for an object o as the number of different classes in $\text{CLASSES}(o)$, that is, the classes the object belongs to. 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).

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 is violated (for the sake of brevity we define them rather informally):

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 object values are well defined throughout hierarchies.

Lemma 3 *If a Herbrand interpretation satisfies conditions* WT, CON, DIS, COH, *and* FUN *with respect to a scheme* $\mathbf{S}$*, then it belongs to the image of* ϕ *over the pre-instances of* $\mathbf{S}$*.*

Another property of the function ϕ is that if $\phi(\mathbf{s}_1) = \phi(\mathbf{s}_2)$, then $\mathbf{s}_1$ and $\mathbf{s}_2$ are oid-equivalent pre-instances. Moreover, the notion of oid-equivalence can be easily extended to interpretations: $\mathcal{F}_1$ is *oid-equivalent* to $\mathcal{F}_2$ if there is a permutation of $\mathcal{O}$ that transforms $\mathcal{F}_1$ into $\mathcal{F}_2$. Then, we have that 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 interpretations: $\Phi : [\mathbf{s}] \mapsto [\phi(\mathbf{s})]$. Since $\phi(\mathbf{s}_1)$ is equivalent to $\phi(\mathbf{s}_2)$ only if 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 interpretations that satisfy the five conditions above. The inverse of Φ is therefore defined over equivalence classes of interpretations that satisfy conditions WT, CON, DIS, COH, and FUN.

Given a program $\mathbf{P}$ over a scheme $\mathbf{S}$ and a pre-instance $\mathbf{s}$ it is therefore possible to build the IsaLog set of clauses $\mathbf{P} \cup \phi(\mathbf{s})$ that is essentially a set of clauses of ordinary logic programming with function symbols. In the next subsection we show that, if the scheme does not contain isa relationships, this set of clauses provides an equivalent way of defining the semantics of programs. Then, we extend this result to the general case, considering also isa.

6.1 Reduction to Logic Programming if there are no isa

Let us consider an IsaLog scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$ where ISA is the identity relation and therefore the classes are disjoint in every pre-instance $\mathbf{s}$ of $\mathbf{S}$. Also, consider a program $\mathbf{P}$ over $\mathbf{S}$. Since there are no significant isa, there can be no specialization-clauses, so $\mathbf{P}$ contains only relation-clauses and oid-invention-clauses.

Given a pre-instance $\mathbf{s}$, let us consider the IsaLog set of clauses $\mathbf{P} \cup \phi(\mathbf{s})$. By known results [12], this set has a unique minimal model $\mathcal{M}_{\mathbf{P} \cup \phi(\mathbf{s})}$, which can be either finite or infinite. It can be shown that $\mathcal{M}_{\mathbf{P} \cup \phi(\mathbf{s})}$ satisfies conditions WT, DIS, COH, FUN, and, trivially, CON. Furthermore, if it is finite, then there exists an instance $[\mathbf{s}] = \Phi^{-1}([\mathcal{M}_{\mathbf{P} \cup \phi(\mathbf{s})}])$. We can define the *LP-semantics* of an IsaLog program $\mathbf{P}$ over a scheme $\mathbf{S}$ as a partial function $\text{LP-SEM}_{\mathbf{P}}$ that maps instances to instances corresponding to minimum models (when they are finite): $\text{LP-SEM}_{\mathbf{P}}([\mathbf{s}])$ equals $\Phi^{-1}([\mathcal{M}_{\mathbf{P} \cup \phi(\mathbf{s})}])$ if $\mathcal{M}_{\mathbf{P} \cup \phi(\mathbf{s})}$ is finite, and undefined otherwise.

Theorem 2 *For every* IsaLog *program* $\mathbf{P}$ *over a scheme* $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$ *where* ISA *is the identity relation the declarative semantics and the LP-semantics coincide.*

It should be noted that Theorem 2 guarantees the equivalence of various semantics, since it is known that three equivalent semantics exist for ordinary logic programming (model-theoretic, fixpoint, and proof-theoretic).

A comment is useful here. The declarative semantics and the LP-semantics coincide also when undefined: in fact, recursion through oid invention, which, as we saw above, can give rise to model undefinedness (in declarative semantics) corresponds to an infinite interpretation and unbounded structures for functor terms (in LP-semantics).

6.2 Reduction to Logic Programming with nontrivial isa

In contrast with what we saw in the previous subsection, there is no direct reduction to traditional logic programming in the general case: isa relationships require generation of facts for the satisfaction of containment constraints — intuitively, facts that correspond to the propagation of oid's through the class hierarchy.

A possible reduction to logic programming can be obtained by adding, to each program, clauses that enforce the isa relationships defined over the corresponding scheme (as it is done in the Logres language [6]). More precisely, given a scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$, we define the *isa-clauses* , $_{\mathbf{S}}$ for $\mathbf{S}$ as follows:

$$\{C_2(\text{OID} : x_0, A_1 : x_1, \ldots, A_k : x_k) \leftarrow C_1(\text{OID} : x_0, A_1 : x_1, \ldots, A_{k+h} : x_{k+h}) \mid C_1 \text{ISA} C_2, \text{TYP}(C_2) = (A_1 : \tau_1, \ldots, A_k : \tau_k) \text{ and } \text{TYP}(C_1) = (A_1 : \tau_1, \ldots, A_{k+h} : \tau_{k+h})\}$$

Note that these are neither specialization nor oid-invention clauses. However, this is not contradictory with our approach, as here we refer to logic programs, where clauses of this form are allowed and can be handled in a standard fashion.

Given a program $\mathbf{P}$ over a scheme $\mathbf{S}$ and a pre-instance $\mathbf{s}$, it is therefore possible to build the IsaLog set of clauses $\mathbf{P} \cup$, $_{\mathbf{S}} \cup \phi(\mathbf{s})$, which is essentially a set of clauses of ordinary logic programming with function symbols. Again, this set has a unique minimal model

$\mathcal{M}_{\mathbf{P}\cup\Gamma_{\mathbf{S}}\cup\phi(\mathbf{s})}$ that can be either finite or infinite. In general, $\mathcal{M}_{\mathbf{P}\cup\Gamma_{\mathbf{S}}\cup\phi(\mathbf{s})}$ satisfies conditions WT, CON, DIS, and COH, whereas it need not satisfy condition FUN, as shown in section 5. Therefore the existence of an instance $[\mathbf{s}'] = \Phi^{-1}([\mathcal{M}_{\mathbf{P}\cup\Gamma_{\mathbf{S}}\cup\phi(\mathbf{s})}])$ cannot be guaranteed. We define the *LP-semantics* of an IsaLog program $\mathbf{P}$ over a scheme $\mathbf{S}$ as a partial function LP-SEM$_{\mathbf{P}}$ that maps instances to instances corresponding to minimum models (when they are finite and satisfy the required conditions): LP-SEM$_{\mathbf{P}}([\mathbf{s}])$ equals $\Phi^{-1}([\mathcal{M}_{\mathbf{P}\cup\phi(\mathbf{s})}])$ if $\mathcal{M}_{\mathbf{P}\cup\phi(\mathbf{s})}$ is finite and satisfies condition FUN, undefined otherwise.

Theorem 3 *For every* IsaLog *program* $\mathbf{P}$ *the declarative semantics and the LP-semantics coincide.*

This approach is apparently interesting, but not completely satisfactory, because of two reasons. First, it uses clauses with a different "philosophy" than the clauses allowed in IsaLog programs, which have a natural counterpart in the actual operations specified. Second, and more important, it turns out that, when extending the language by introducing negation, it presents undesirable limitations with respect to stratification: there are programs that, if extended with isa-clauses in this way, are not stratified in the ordinary sense, but have a reasonable model. [3]

7 Fixpoint semantics

In this section we present the fixpoint semantics for IsaLog programs.

Let a program $\mathbf{P}$ over a scheme $\mathbf{S}$ be fixed. We say that an interpretation $I_{\mathbf{S}}$ *satisfies* a ground atom L if $L \in I_{\mathbf{S}}$. Similarly for a set of ground atoms. Given a clause γ and an interpretation $I_{\mathbf{S}}$, $I_{\mathbf{S}}$ *satisfies* γ if for each substitution θ ground over γ such that $I_{\mathbf{S}}$ satisfies $\theta(\text{BODY}(\gamma))$ it is the case that $I_{\mathbf{S}}$ satisfies $\theta(\text{HEAD}(\gamma))$. An interpretation $I_{\mathbf{S}}$ is a *model* for a program $\mathbf{P}$ if it satisfies all the clauses in $\mathbf{P}$.

The main step in the definition of a fixpoint semantics is the introduction of a continuous transformation associated with a program. For the sake of space, we omit the technical definition of *continuous transformation* [12]. 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 $2^{\text{H}}\mathbf{S}$ of $\text{H}_{\mathbf{S}}$ to itself, defined as follows:

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

The closure with respect to isa enforces the satisfaction of containment constraints associated with hierarchies, as required by Condition CON defined in the previous section.

Then, given a set of clauses , over a scheme $\mathbf{S}$ we define the trasformation $T_{\Gamma,0}$ associated with , as a mapping from the powerset $2^{\text{H}}\mathbf{S}$ to itself, as follows:

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

Finally, we introduce the *immediate consequence operator* T_Γ associated with , as a mapping from Herbrand interpretations to Herbrand interpretations. Given a set of clauses , over a scheme $\mathbf{S}$, and an interpretation $I_{\mathbf{S}}$, we define

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

We now define *powers* of a monotonic operator T, putting $T{\uparrow}0(I) = I$, $T{\uparrow}(n+1)(I) = T(T{\uparrow}n(I))$ (for every $n \geq 0$), and $T{\uparrow}\omega(I) = \cup_{n=0}^{\infty} T{\uparrow}n(I)$.

Lemma 4 *For every scheme* $\mathbf{S}$ *and for every set of clauses , over* $\mathbf{S}$*, the transformation* T_Γ *is both finitary and monotonic, and thus continuous.*

As a consequence, by Knaster-Tarski theorem [12], we have that, for every scheme $\mathbf{S}$ and for every set of clauses , over $\mathbf{S}$, (i) the transformation T_Γ has at least one fixed point, (ii) the set of the fixed points of T_Γ is a complete lattice, (iii) the least fixed point of T_Γ can be computed as $T_\Gamma{\uparrow}\omega(\emptyset)$.

Given an IsaLog program $\mathbf{P}$ over a scheme $\mathbf{S}$, we can therefore define the *fixpoint semantics* of $\mathbf{P}$ as a partial function FP-SEM$_{\mathbf{P}}$ that maps instances to instances, as follows. Given an instance $\mathbf{s}$, we can consider the corresponding interpretation $\phi(\mathbf{s})$, that along with $\mathbf{P}$ forms a set of clauses $\mathbf{P} \cup \phi(\mathbf{s})$ on which we compute a fixpoint: FP-SEM$_{\mathbf{P}}([\mathbf{s}])$ equals $\Phi^{-1}([T_{\mathbf{P}\cup\phi(\mathbf{s})}{\uparrow}\omega(\emptyset)])$ if $T_{\mathbf{P}\cup\phi(\mathbf{s})}{\uparrow}\omega(\emptyset)$ is finite and satisfies condition FUN, undefined otherwise.

It turns out that the three semantics proposed for the IsaLog language coincide. This is stated in the next theorem. Therefore, we have a robust concept, thus confirming the validity of the approach.

Theorem 4 *For every* IsaLog *program* $\mathbf{P}$*, the declarative semantics* D-SEM$_{\mathbf{P}}$*, the logic programming semantics* LP-SEM$_{\mathbf{P}}$*, and the fixed point semantics* FP-SEM$_{\mathbf{P}}$ *coincide.*

8 Conclusions

This paper has presented the IsaLog model and language. The main novel feature is the use of explicit Skolem functors for the generation and manipulation of object identifiers within classes and through hierarchies. The major result is the equivalence of the different semantics introduced. Several issues need to be further investigated, as follows.

- The characterization of (or at least sufficient conditions for) the definedness of the semantics of programs over instances.
- The extension of the language, specifically with the introduction of negation. A stratified semantics coherent with respect to isa seems possible.
- The management of integrity constraints, especially with regard to their preservation in derived data.

References

[1] 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.

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

[3] P. Atzeni, L. Cabibbo, and G. Mecca. IsaLog a declarative language with hierarchies and negation. Submitted for publication.

[4] 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.

[5] C. Beeri. A formal approach to object-oriented databases. *Data and Knowledge Engineering*, 5:353–382, 1990.

[6] 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.

[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] 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.

[10] S. Khoshafian and G. Copeland. Object identity. In *ACM Symp. on Object Oriented Programming Systems, Languages and Applications*, 1986.

[11] 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.

[12] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, second edition, 1987.

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

[14] D. Maier. Why isn't there an object-oriented data model. Technical Report CS/E-89-002, Oregon Graduate Center, 1989. A condensed version was an invited paper at the *IFIP 11th World Computer Congress*, San Francisco, August-September 1989.

[15] J.D. Ullman. *Principles of Database and Knowledge Base Systems*, volume 1. Computer Science Press, Potomac, Maryland, 1988.