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

IsaLog$^{(\neg)}$: a Deductive Language with Negation for Complex-Object Databases with Hierarchies

Paolo Atzeni[*], Luca Cabibbo[*], Giansalvatore Mecca[⊗]

RT-INF-13-96 **Settembre 1996**

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

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

*Preliminary versions of portions of this paper appeared under the title "*IsaLog*: A declarative language for complex objects with hierarchies" in* Ninth IEEE International Conference on Data Engineering *[8], and "*IsaLog$^{(\neg)}$*: A deductive language with negation for complex-object databases with hierarchies" in* Third Intl. Conference on Deductive and Object-Oriented Databases *[9].*
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 was partially supported by Systems & Management S.p.A.*

ABSTRACT

IsaLog$^{(\neg)}$ is a research activity aimed at developing a framework that integrates deductive and object-oriented features. The data model has complex objects with classes, relations, and isa hierarchies, and the language is rule based. The main issue is the definition of the semantics of the language. For the (positive) IsaLog framework three different semantics are given and proven to be equivalent: a model-theoretic semantics, a fixpoint semantics, and a semantics based on a reduction to ordinary logic programming with function symbols. Then the semantics of the IsaLog$^{\neg}$ language is proposed. It presents novel features mostly due to the interaction of hierarchies with negation in the body of rules. Two semantics are presented for IsaLog$^{\neg}$ programs: a stratified semantics based on an original notion of stratification, which takes into account hierarchies, and a reduction to logic programming with function symbols. The two semantics are then shown to be equivalent. The solutions are based on the use of explicit Skolem functors, which represent a powerful tool for the management of object identifiers.

1 Introduction

Deductive languages for complex-object databases have received a great deal of attention in the last few years. This effort stems from the need to find a neat and elegant semantics for object orientation in databases and, at the same time, to achieve a strong expressive power of query languages.

In this paper we introduce a simple object-oriented data model, and a rule-based query language called IsaLog for the model. The data model we propose is essentially a "structural" object model, including features such as: object, object identity, class (as a collection of objects), class hierarchies and inheritance, and, in a loose sense, typing. The main motivation which led us to consider such a simple object model is to investigate rule-based query languages in the context of object bases, where objects are uniquely identified by means of oid's and organized in class hierarchies. In particular, our goal is to study the impact of traditional object-oriented features, like as object identity and inheritance, in the context of a rule-based query language, possibly with negation.

1.1 Background

A starting point has been undoubtedly represented by the introduction of the well known Datalog$^{(\neg)}$ language for the relational model. Datalog$^{(\neg)}$ is a rule-based language with a fully declarative semantics and is more expressive than classical relational languages. Following the introduction of object-oriented features in data models, the extension of declarative languages in order to deal with complex objects has been pursued. In this context, data models include *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*. *Object identifiers (oid's)* are associated with objects, in order to allow duplicates and for object sharing and inheritance.

The first proposals in this field go back to the early eighties, and are concerned with the languages LOGIN (Aït-Kaci and Nasr [4]), and O-logic (Maier [25]).

LOGIN is a Prolog-based language for querying complex-object databases. Objects are structures built by means of ordinary logic-programming function symbols, and isa-hierarchies and inheritance are allowed among classes of objects. The semantics of the language is developed in a Prolog-like fashion, thus being proof-theoretic, and an ad-hoc unification algorithm is presented, in order to deal with built-in inheritance. The type-inference mechanism is quite appealing, but the resolution-based semantics seems hardly suitable to a database framework.

On the other side, the so called "alphabet logics"(Maier [25], Chen and Warren [16], Kifer et al. [20, 22]) represent a strong effort directed to the development of a logic-based framework for the management of objects and queries. In particular, F-logic [20, 21] proposes a first-order semantics and a higher-order syntax, thus being able to perform interesting tasks such as schema browsing. Soundness and completeness of the proposed resolution procedure were proven, along with an equivalent model-theoretic semantics.

The first attempt to develop a deductive language over an object-oriented data model within a traditional database framework has been pursued in a seminal paper by Abiteboul and Kanellakis [3], with the proposal of the IQL language. It involves a data model with a clear distinction between database scheme and instance, where complex structures are built by means of tuple and set type constructors. The rule-based language is a suitable extension of Datalog for handling object identity. The core of the language is the introduction of oid invention as a programming primitive; it allows for the creation of object identifiers (oid's) in order to manage new objects. Unfortunately, this appealing feature, along with the chosen identification mechanism, makes the semantics purely operational, since the introduction of each new object requires the task of choosing its oid among those not used in the instance yet. Moreover, the typing system does not explicitly embed hierarchies, and inheritance is supported only indirectly.

A neat and elegant semantics for oid invention has been proposed in ILOG (Hull and Yoshikawa [17]), where a *Skolemization* mechanism is adopted in order to make inventions truly declarative. However, duplicates are not allowed and isa-hierarchies are not considered.

In this paper we present IsaLog$^{(\neg)}$. Our work aims at defining a model and language that integrate object-oriented features — such as object identity and built-in inheritance — with a deductive language with negation that is an extension of Datalog$^{(\neg)}$. To simplify the treatment, we do not consider any "behavioural" feature, such as: method, encapsulation, late binding, overriding, to do not overwhelm our goal.

1.2 Contributions of the Paper

The language we propose — called IsaLog$^{(\neg)}$ — is similar to the language ILOG [17]. A distinctive feature of IsaLog$^{(\neg)}$ is the use of *explicit Skolem functors* (an extension of the *implicit* Skolem functors of ILOG) to deal with oid invention. 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 of this paper is the definition of the semantics of the language. We first define three different semantics for positive (that is, without negation) IsaLog programs and show their equivalence. The first semantics is model-theoretic, that is, purely declarative and based on the notion of a *model*. The second semantics is based on a reduction to ordinary logic programming with function symbols, following and integrating two independent approaches: ILOG [17] (in the management of functors) and LOGRES [13] (in dealing with hierarchies.) Finally, we provide a fixpoint semantics.

We then consider IsaLog$^{\neg}$ programs, in which the presence of negation in body of rules is allowed. Here the main topic is the definition of a stratified fixpoint semantics for IsaLog$^{\neg}$ programs. We introduce the notion of *isa-coherent stratification*, which is 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 formal definition of the data model. The language syntax is defined in Section 4. The semantics of IsaLog (positive) programs is investigated in Sections 5 through 9. In Section 5 we study the model-theoretic semantics. The semantics based on a reduction to ordinary logic programming with function symbols is introduced in two steps: Section 6 explains how an IsaLog instance can be represented in a logic-programming fashion, whereas the reduction is proposed in Section 7. Then, Section 8 deals with the fixpoint semantics and with the equivalence of the three proposed semantics for positive programs. The semantics of general IsaLog$^{\neg}$ programs, along with the definitions concerning the isa-coherent stratification, is proposed in Section 10. Section 11 contains our conclusions. The complete equivalence proof of the various semantics for positive programs is proposed in Appendix A.

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*, 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, which is stored in the instance. This has been done in order to keep oid's in the instance, while making functors transparent.

Tuples in relations, in object values, and in arguments of functions may contain domain values and oid's, the latter being used as references to objects.

Isa hierarchies are allowed among classes, with multiple inheritance and without any requirement of completeness or disjointness.

The IsaLog language is declarative, a suitable extension of Datalog [15] 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 also belongs to some of its subclasses.

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. In order to keep the semantics as general as possible, we do not require disjointness between input and output schemes (in contrast with other approaches, from Datalog [15, 28] to IQL [3]). Because of the presence of isa hierarchies, the usual separation between input and output would indeed be a limitation. Moreover, we do not require, as in other works [3, 10, 27], 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 nonempty intersection, the most specific class requirement would impose a class *married-students*, even when it is not really significant in the application.

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. Let us give an example (in which Δ is a *domain* of atomic values, including strings and integers): given a relation *fatherhood*, with type *(father:*Δ*, child:*Δ*)* and a relation *motherhood*, with type *(mother:*Δ*, child:*Δ*)*, assume we want to build the class *couple*, with type *(father:*Δ*, mother:*Δ*)*, that contains all the couples of parents. Intuitively, we could use the following clause:

$$\begin{aligned} \mathit{couple}(\text{OID} : x, \mathit{father} : f, \mathit{mother} : m) \;\leftarrow\;& \mathit{fatherhood}(\mathit{father} : f, \mathit{child} : c),\\ & \mathit{motherhood}(\mathit{mother} : m, \mathit{child} : c). \end{aligned}$$

The variable x represents new oid's to be created. Clearly, each object representing a couple must have assigned an oid not used in the database. Such a behavior represents a novel feature with respect to ordinary Datalog framework, so that it seems necessary to establish a different strategy to evaluate oid-invention clauses with respect to ordinary clauses.

Following a logic-programming approach, some proposals in the literature (IQL, LOGIDATA+, LOGRES) [3, 7, 13] mainly adopt a "fact for each instance" policy, that is, an oid-invention clause generates a new oid for each satisfiable ground instance of its body. This means that the clause of our example would invent as many *duplicates* (that is, objects with the same value and different oid's) for each couple *(father, mother)* as the number of children they have in common. Clearly, this is not the intended meaning of the clause. More generally, we would like to have a means to control duplicate generation in such a context.

A possible solution for this problem has been proposed in the ILOG language [17] by introducing a semantics of invention based on *Skolem functors*. Skolem functors are strongly related to the function symbols of logic programming; in this framework, they provide a neat syntactic tool to specify the variables on which oid invention depends. ILOG comes with a transparent skolemization mechanism, in which such variables are chosen to be exactly those occurring in the clause head. Thus, ILOG would interpret the clause as:

$$\begin{aligned} \mathit{couple}(\text{OID} : f_{couple}(\mathit{father} : f, \mathit{mother} : m), \mathit{father} : f, \mathit{mother} : m) \leftarrow\;&\\ & \mathit{fatherhood}(\mathit{father} : f, \mathit{child} : c),\\ & \mathit{motherhood}(\mathit{mother} : m, \mathit{child} : c). \end{aligned}$$

where f_{couple} represents the Skolem functor. We say that ILOG functors are *implicit* since they are under the control of the system.

An approach with implicit functors allows for a nice reduction to ordinary logic-programming semantics, thus making oid invention truly declarative, but it is not completely satisfactory. Mainly, it does not allow the generation of duplicates (when needed) and therefore, in the ILOG framework, equality implies identity, against the main motivation for the use of oid's.

We propose a technique that gives to the user the control over Skolem functors, and therefore over the variables (and so values) responsible for the creation of new oid's. An *explicit Skolem functor* for an ISALOG$^{(\neg)}$ program over a scheme is an identifier whose type is explicitly declared in the scheme. Each functor has a class associated with it and:

- explicit functors generalize implicit ones: an explicit functor term for an oid of a class has at least the same attributes as the objects of that class. This is necessary in order to guarantee that object values are well-defined (that is, there are no objects with the same oid and different values); in addition, a functor for a class may contain other attributes;

- different functors may be associated with the same class.

In this way the generation of duplicates is allowed. Let us give an example that outlines the importance of duplicates and the flexibility provided by the use of explicit Skolem functors, even without hierarchies.

Example 2.1 [Books and Volumes] 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 corresponding to the same book in the two libraries. Instead, if we want the class of all volumes, we have to retain duplicates.

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.

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

□

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. 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 [19].

2.2 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 even more important when both isa hierarchies 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 requirement (that is, the disjointness between the input and the output scheme), to easily deal with programs in which a subclass is newly generated and it 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.

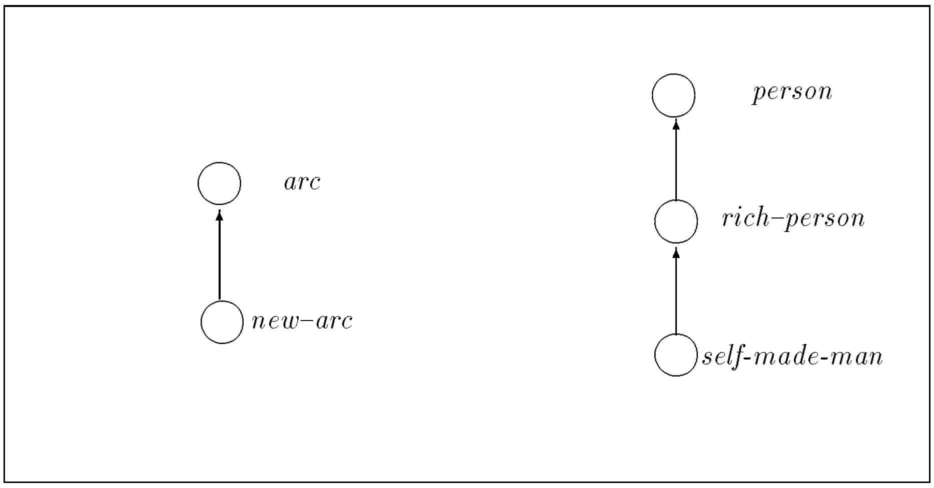


Figure 1: Examples of hierarchies

Example 2.2 [Strongly Connected Graph] Consider a graph represented by means of two classes: *node* and *arc*, with type *()* and *(from:node, to:node)* respectively. Suppose we want to trasform the graph in a strongly connected one, in which there is at least a directed path between each pair of nodes. We can do this by adding an arc for each pair of non-connected nodes, by means of the following program (where *new-arc* ISA *arc*, see Figure 1):

$$\begin{array}{lll}
\gamma_1\text{: } path(from\text{: } x,\ to\text{: } y) & \leftarrow & node(\mathrm{OID}\text{: } x), \\
 & & node(\mathrm{OID}\text{: } y), \\
 & & arc(\mathrm{OID}\text{: } z, from\text{: } x, to\text{: } y). \\
\gamma_2\text{: } path(from\text{: } x,\ to\text{: } y) & \leftarrow & path(from\text{: } x,\ to\text{: } w), \\
 & & arc(\mathrm{OID}\text{: } z, from\text{: } w, to\text{: } y). \\
\gamma_3\text{: } new\text{-}arc(\mathrm{OID}\text{:} f_{new\text{-}arc}(from{:}x, to{:}y),\ from{:}x, to{:}y) & \leftarrow & node(\mathrm{OID}\text{: } x), \\
 & & node(\mathrm{OID}\text{: } y), \\
 & & \neg\ path(from\text{: } x,\ to\text{: } y).
\end{array}$$

This program appears to be stratified: in fact, there is no apparent recursion through negation. 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. □

Example 2.2 confirms that the notion of stratification needs to be modified if hierarchies on classes exist. An intuitive proposal [13] for handling the semantics of hierarchies 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:

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

(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 program. The next example shows that this technique, well suited to a positive framework, does not catch the complete meaning of negation.

Example 2.3 [Rich Persons] Consider the class *person* with type *(name: Δ, asset-value: integer, father: person)*, and suppose *rich-person* ISA *person, self-made-man* ISA *rich-person* (see Figure 1). Suppose we want to specialize people on the basis of their assets, distinguishing rich people with a rich father from self made men:

$$\begin{aligned}\gamma_1' :\ & \textit{rich-person}(\mathrm{OID}: x,\ name : n,\ asset : a,\ father : f) \leftarrow \\ & \qquad \textit{person}(\mathrm{OID}: x,\ name : n,\ asset : a,\ father : f),\ a > 100K.\\ \gamma_2' :\ & \textit{self-made-man}(\mathrm{OID}: x,\ name : n,\ asset : a,\ father : f) \leftarrow \\ & \qquad \textit{rich-person}(\mathrm{OID}: x,\ name : n,\ asset : a,\ father : f),\\ & \qquad \neg\ \textit{rich-person}(\mathrm{OID}: f,\ name : nf,\ asset : af,\ father : ff).\end{aligned}$$

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.* However, if the isa-clauses associated with the scheme are added to the program, the resulting set of clauses is not stratified. In fact, isa-clauses establish that *rich-person* depends on *self-made-man*, since *self-made-man* ISA *rich-person*, and, by rule γ_2', *self-made-man* negatively depends on *rich-person.* □

The examples suggest that:

- ordinary stratification, defined [6] 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 Section 10 we introduce an *isa-coherent stratified semantics* for IsaLog$^\neg$ programs, based on a notion of *isa-coherent stratification*, which 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. This reduction is based on the use of explicit Skolem functors, thus confirming their importance.

3 The Data Model

This section is devoted to the formal introduction of the structural object model we use in the remainder of the paper.

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

3.1 Schemes

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*), $\mathbf{F}$ (the *functors*) are finite, pairwise disjoint subsets of $\mathcal{L}$;
- TYP is a total function on $\mathbf{C} \cup \mathbf{R} \cup \mathbf{F}$ that associates a *tuple type* with each class name, relation name and functor, as follows:
  - the value of TYP over each class in $\mathbf{C}$ and each relation in $\mathbf{R}$ is a (flat) *tuple type* $(A_1 : \tau_1, \ldots, A_k : \tau_k)$; the A_i's are distinct elements of $\mathcal{L}$ called the *attributes*, and each τ_i (the *type* of A_i) is either a class name in $\mathbf{C}$ or the domain Δ;
  - the value of TYP over each functor $F \in \mathbf{F}$ is a pair (C, τ), where: (i) C is a class name in $\mathbf{C}$ (the *class associated with* F) with $\textsc{typ}(C) = (A_1 : \tau_1, \ldots, A_k : \tau_k)$, and (ii) τ is a tuple type $(A_1 : \tau_1, \ldots, A_k : \tau_k, A'_1 : \tau'_1, \ldots, A'_h : \tau'_h)$, with $h \geq 0$.

  TYP is essentially used to associate a structure to the constructs of the scheme. Note how a class name is associated with each functor: the functor is used to create new objects in that class; the arguments of the functor include the attribute of the class to avoid the generation of multiple objects with the same oid;
- ISA is a partial order over $\mathbf{C}$, with the following conditions:
  - 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 $\textsc{typ}(C'')$, where a type τ' is a *subtype* [14] of a type τ'' (in symbols $\tau' \preceq \tau''$) if one of the following conditions holds:
    1. $\tau' = \tau'' = \Delta$;
    2. $\tau', \tau'' \in \mathbf{C}$ and $\tau'\ \textsc{isa}\ \tau''$;
    3. τ' and τ'' are both tuple types, $\tau' = (A'_1 : \tau'_1, \ldots, A'_k : \tau'_k)$, $\tau'' = (A''_1 : \tau''_1, \ldots, A''_h : \tau''_h)$ and for each $j \in \{1, \ldots, h\}$ there is an $i \in \{1, \ldots, k\}$ such that $A'_i = A''_j$ and $\tau'_i \preceq \tau''_j$.
  - if C' and C'' have a *common ancestor* (that is, a class C such that $C'\,\textsc{isa}\,C$ and $C''\ \textsc{isa}\ C$) and a *common attribute* A, then there is a common ancestor C_1 of C' and C'' such that A is an attribute of C_1;
  - if there are $C, C', C'' \in \mathbf{C}$ such that $C\,\textsc{isa}\,C'$ and $C\,\textsc{isa}\,C''$, then C' and C'' have a common ancestor in $\mathbf{C}$; that is, *multiple inheritance* is allowed only beneath a common ancestor.

The partial order ISA has the usual role of *is-a relationship*. The condition of subtyping is imposed in order to guarantee that the elements of a subclass have a type "compatible" with that of the superclass. The definition of subtyping for tuple types expresses the idea that a tuple t belongs to a tuple type τ if it has *at least* the components of τ (with the same type or refined), and possibly some more. The condition about common attributes insures that each attribute, within a hierarchy, has a unique uppermost class defining it and its "upper type," in such a way that possible redefinitions of the type of an attribute are forced to happen in a type–compatible fashion. The condition concerning multiple inheritance implies that each class belongs to a unique hierarchy, in such a way that each distinct hierarchy corresponds to a taxonomy of the universe of discourse.

3.2 Instances

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 $\tau = \Delta$, then $\text{VAL}(\tau)$ is the domain Δ; (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 atomic types — that is, either the domain Δ or a class name — 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, in the spirit of IQL [3], value-sets of classes 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 a common requirement for object oriented systems [18, 26]).

Following ILOG [17], 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:
  1. if C' ISA C'', then $\mathbf{c}(C') \subseteq \mathbf{c}(C'')$;
  2. $\mathbf{c}(C') \cap \mathbf{c}(C'') \neq \emptyset$ only if C' and C'' have a common ancestor. [1]
- $\mathbf{r}$ 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. The *active object domain of* $\mathbf{s}$ is the set $\text{AODOM}(\mathbf{s}) = \bigcup_{C \in \mathbf{C}} \mathbf{c}(C)$ of all oid's appearing in classes of $\mathbf{s}$. For each $o \in \text{AODOM}(\mathbf{s})$, consider the set of classes that contain o:

  $$\text{CLASSES}(o) = \{C \mid C \in \mathbf{C},\ o \in \mathbf{c}(C)\}$$

  and the set of attributes $\text{ATTR}(o)$ that belong to the types of some C in $\text{CLASSES}(o)$, that is:

  $$\text{ATTR}(o) = \{A \mid \text{TYP}(C) = (\ldots, A : \tau, \ldots), C \in \text{CLASSES}(o)\}.$$

[1]As a consequence, if ISA is the identity relation (and so there are no non-trivial subset constraints) then the extensions of the classes are pairwise disjoint, as it is usually assumed in other frameworks that do not consider hierarchies [3, 17]. Also, this condition is coherent with the requirement that multiple inheritance is allowed only beneath a common ancestor.

We call $\text{TYPES}(o, A)$ the set of types associated with A in $\text{CLASSES}(o)$. Note that, from the various conditions on schemes and instances, we have that, for each A and o, $\text{TYPES}(o, A)$ is either the singleton $\{\Delta\}$ or a set of classes. Then for each o, $\mathbf{o}(o)$ is a tuple over the set of attributes $\text{ATTR}(o)$, such that, for each attribute A:

1. if $\text{TYPES}(o, A) = \{\Delta\}$, then the corresponding value belongs to the domain Δ;
2. if $\text{TYPES}(o, A)$ is a set of classes, then the corresponding value is an oid o' that belongs to each of the classes in $\text{TYPES}(o, A)$, that is, for each class $C \in \text{TYPES}(o, A)$, it happens that $o' \in \mathbf{c}(C)$.

- $\mathbf{f}$ is a function that associates with each functor $F \in \mathbf{F}$ a function $\mathbf{f}(F)$ as follows. Let $\text{TYP}(F) = (C, \tau)$; then $\mathbf{f}(F)$ is a partial injective function from the value set of the tuple type τ to (a subset of) $\mathbf{c}(C)$. The functions corresponding to the various functors are required to satisfy the following conditions:

  1. the ranges form a partition of $\text{AODOM}(\mathbf{s})$ (in particular, they are pairwise disjoint);
  2. a partial order $\leq$ is defined among the oid's in such a way that if the oid o_2 is the result of the application of a function to a tuple that involves the oid o_1, then $o_1 < o_2$ (that is, $o_1 \leq o_2$ and $o_1 \neq o_2$). [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").

Our definition of instance is more complex than similar definitions in other models, such as IQL [3], mainly because we do not require for each object a *most specific class.*

The intuition behind the definition of function $\mathbf{o}$ is that, for each oid o, $\mathbf{o}(o)$ is a tuple over the attributes in $\text{ATTR}(o)$, in such a way that the type of $\mathbf{o}(o)$ is a subtype of $\text{TYP}(C)$ for each class C containing o.

With respect to the data model of IQL [3], our functions $\mathbf{r}, \mathbf{c}$, and $\mathbf{o}$ are the analogue of their assignments ρ, π, and ν.

The definition of the function $\mathbf{f}$, used to carry information about oid creation, is not common in other data models, though somewhat suggested in the literature [1, 19].

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 $\mathbf{s}$.

Example 3.1 [Representation of Strings] The data model allows for modeling inductively defined types, such as lists, trees, and (in a loose sense) sets. This is an interesting feature, since it allows for managing complex data structures even though the data model, which has been kept as simple as possible, does not explicitly provide complex types. This ability is a consequence of having class names as user-defined types of a scheme, in such a way that the type of an attribute of a class may be another class name. At the instance level, the value associated with an object may be an oid — an indirect reference to another

[2]This condition guarantees well-definedness in the generation of oid's, by avoiding circularity, as we will see later.

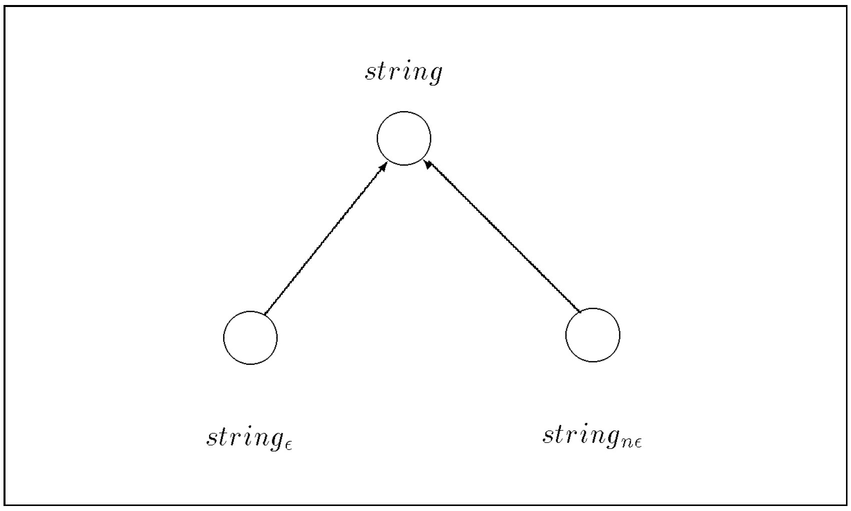


Figure 2: Hierarchy of classes for representing strings

object. Furthermore, the presence of isa hierarchies allows to define a type as the "union" of different types. And this is all we need to model inductively defined types.

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

$$\begin{array}{lcl}
\mathbf{C} & = & \{string, string_\epsilon, string_{n\epsilon}\} \\
\mathbf{R} & = & \{\} \\
\mathbf{F} & = & \{F_{string}, F_{string_\epsilon}, F_{string_{n\epsilon}}\} \\
\textsc{typ}(string) & = & () \\
\textsc{typ}(string_\epsilon) & = & () \\
\textsc{typ}(string_{n\epsilon}) & = & (ch : \Delta, s : string) \\
\textsc{typ}(F_{string}) & = & (string, ()) \\
\textsc{typ}(F_{string_\epsilon}) & = & (string_\epsilon, ()) \\
\textsc{typ}(F_{string_{n\epsilon}}) & = & (string_{n\epsilon}, ()) \\
\textsc{isa} & = & \text{the reflexive and transitive closure of} \\
 & & \{(string_\epsilon, string), (string_{n\epsilon}, string)\}
\end{array}$$

The same scheme can be described using a scheme definition language, as follows:

$$\begin{array}{lll}
CLASS\ string & & () \\
CLASS\ string_\epsilon & \textsc{isa}\ string & () \\
CLASS\ string_{n\epsilon} & \textsc{isa}\ string & (ch : char, s : string) \\
FUNCTOR\ F_{string_\epsilon} & string_\epsilon & () \\
FUNCTOR\ _{string_{n\epsilon}} & string_{n\epsilon} & (ch : char, s : string)
\end{array}$$

The scheme definition language is rather self-explanatory; the keyword *CLASS* is used to define class names, followed by the associated tuple type. The keyword ISA is used to declare isa relationships (see Figure 2). The keyword *FUNCTOR* introduces a functor definition, consisting in a class name and a tuple type.

This scheme $\mathbf{S}_{string}$ can represent a string of any length (over a fixed alphabet, represented by the elements of domain Δ). Given a string w in Δ^*, let us define the instance $\mathbf{s}_w$ over $\mathbf{S}_{string}$ representing string w. Instance $\mathbf{s}_w$ contains, in class *string*, all and only the strings that are suffixes for w. That is, if $w = a_1 \dots a_n$, with $n \geq 0$, then $\mathbf{s}_w$ contains $n+1$ objects $o_0, o_1, \dots, o_n$, where:

- $\mathbf{c}(string_\epsilon) = \{o_0\}$ (the empty string);
- $\mathbf{c}(string_{n\epsilon}) = \{o_1, \dots, o_n\}$, with $\mathbf{o}(o_{i+1}) = (ch : a_{n-i}, s : o_i)$;
- $\mathbf{c}(string) = \{o_0, o_1, \dots, o_n\}$.

With respect to functors associated with classes in $\mathbf{S}_{string}$, we have:

- $\mathbf{f}(F_{string_\epsilon})() = o_0$;
- $\mathbf{f}(F_{string_{n\epsilon}})(ch : a_{n-i}, s : o_i) = o_{i+1}$, for $0 \leq i \leq n-1$.

$\square$

4 Syntax of the Language

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_Δ (*value variables*, to denote constants from the domain Δ) and $V_\mathbf{C}$ (*oid variables*, to denote oid's).

4.1 Terms, Atoms, and Literals

The *terms* of the language are:

- *value terms*, which are of two forms: (i) the constants in Δ and (ii) the variables in V_Δ;
- *oid terms*: (i) the oid's in $\mathcal{O}$, (ii) the variables in $V_\mathbf{C}$, and (iii) *functor terms* $F(A_1 : t_1, \dots, A_k : t_k)$, where $F \in \mathbf{F}$ and $\text{TYP}(F) = (C, \tau)$, $\tau = (A_1 : \tau_1, \dots, A_k : \tau_k)$, and each t_i is a value term or an oid term depending on whether τ_i is the domain Δ or a class in $\mathbf{C}$.

A term is said to be *ground* if it contains no variables.

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

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

Given a functor term $f = F(A_1 : t_1, \dots, A_k : t_k)$ we say that an oid term t *ranges over a class C in f* if one of the following conditions holds:

- $t = t_i$ for some i, and the type of the corresponding A_i in F is the class C; or
- for some i it is the case that t_i is a functor term, and t ranges over C in t_i.

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 $C'(\textsc{oid} : t_0, A_1 : t_1, \ldots, A_k : t_k)$ and one of the following holds:
  - $t_0 = t$ and $C' = C$;
  - $t_i = t$ and the type of A_i in C' is the class C for some $i \in \{1, \ldots, k\}$; or
  - for some $i \in \{0, \ldots, k\}$ it is the case that t_i is a functor term, and t ranges over C in t_i.

A *literal* is an atom or its negation. A *positive literal* is an atom, and a *negative literal* is the negation of an atom A, denoted as $\neg A$.

4.2 Rules, Clauses, and Programs

A *rule* has the form:

$$r : A \leftarrow L_1, \ldots, L_p.$$

where r is the *name* of the rule (often omitted), A is an atom, $L_1, \ldots, L_p$ (with $p > 0$) are literals. 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 $\textsc{head}(\gamma)$ and $\textsc{body}(\gamma)$, respectively. If γ is a rule $A \leftarrow L_1, \ldots, L_p$, then $\textsc{head}(\gamma) = A$ and $\textsc{body}(\gamma) = \{L_1, \ldots, L_p\}$. If γ is a fact A, then $\textsc{head}(\gamma) = A$ and $\textsc{body}(\gamma)$ is the empty set.

Let us introduce three relevant forms of clauses. A clause γ is:

- a *relation clause* if $\textsc{head}(\gamma)$ is a relation atom;
- an *oid-invention clause* if $\textsc{head}(\gamma)$ is a class atom $C(\textsc{oid} : t_0, A_1 : t_1, \ldots, A_k : t_k)$, where t_0 is a functor term $F(A_1 : t_1, \ldots, A_k : t_k, \ldots)$ not occurring in $\textsc{body}(\gamma)$ and C is the class associated with F;
- a *specialization clause* if $\textsc{head}(\gamma)$ is a class atom $C(\textsc{oid} : t, \ldots)$, where t is an oid term and $\textsc{body}(\gamma)$ contains (at least) a class atom $C'(\textsc{oid} : t, \ldots)$ as a positive literal, such that C and C' have a common ancestor.

Hereinafter we consider only clauses of the above three forms and call them IsaLog$^{(\neg)}$ *clauses*. An IsaLog *clause* (sometimes called a *positive clause*) is an IsaLog$^{\neg}$ clause whose body contains only positive literals.

We need to impose some constraints on the structure of clauses, in order to have a meaningful semantics. We say that a clause γ is:

- *well-typed* if, whenever an oid term t ranges in $\textsc{head}(\gamma)$ 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 $C' \text{ISA}\, C$;

- *range restricted*, if each variable in $\text{HEAD}(\gamma)$ (and each variable in a negative literal of $\text{BODY}(\gamma)$) occurs in a positive literal of $\text{BODY}(\gamma)$ as well;

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

The notion of well-typedness is very important in order to ensure that each term in a clause is associated with a set of classes compatible with each other; in particular, we require that, whenever a term t ranges over a class C in the head of a clause γ and γ is not an oid-invention clause, then t is associated in the body of γ with a subclass of C.

Note also how the visibility constraint is imposed in order to keep oid's transparent.

An IsaLog$^\neg$ *program* $\mathbf{P}$ over a scheme $\mathbf{S}$ is a set of IsaLog$^\neg$ clauses that are well-typed, range restricted, and visible. An IsaLog *program* (sometimes called a *positive program*) is a program that contains only positive clauses.

5 Declarative Semantics

The declarative semantics of IsaLog programs (that is, *positive* programs) differs from usual (say, Datalog) semantics because of the presence of classes and of the isa hierarchies on them. The use of functors in instances (with the conditions on the functions that correspond to them) allows a correct management of classes and hierarchies. We describe the declarative semantics by mainly noting the differences with the standard development.

Let $\mathbf{P}$ be a program over an IsaLog scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$. A *substitution* θ is a (typed) total function from variables to terms. Consider a pre-instance $\mathbf{s} = (\mathbf{c}, \mathbf{r}, \mathbf{f}, \mathbf{o})$ of $\mathbf{S}$.

Definition 5.1 [Instantiation] The *instantiation* $\text{INST}_\mathbf{s}$ of a ground term t is a functor-free ground term obtained by applying the functions corresponding to functors, thus recursively replacing functor terms with oid's, as follows:

- if t is a constant or an oid, then $\text{INST}_\mathbf{s}(t) = t$;

- if t is a functor term $F(A_1 : t_1, \ldots, A_p : t_p)$, then $\text{INST}_\mathbf{s}(t)$ is:
  - the value of $\mathbf{f}(F)$ over $t' = (A_1 : \text{INST}_\mathbf{s}(t_1), \ldots, A_p : \text{INST}_\mathbf{s}(t_p))$, if $\text{INST}_\mathbf{s}(t_i)$ is defined for every $i \in 1 \ldots p$ and $\mathbf{f}(F)$ is defined over t';
  - undefined otherwise.

□

Instantiation is a partial function because the functions associated with functors are partial. The notion of instantiation is extended in the natural way to atoms and sets of atoms (and so to bodies of rules).

Definition 5.2 [Satisfaction] Given a ground substitution θ and a positive literal L, we say that a pre-instance $\mathbf{s} = (\mathbf{c}, \mathbf{r}, \mathbf{f}, \mathbf{o})$ *satisfies* $\text{INST}_\mathbf{s}(\theta(L))$ (written also $\mathbf{s} \models \text{INST}_\mathbf{s}(\theta(L))$) if:

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

Given a ground substitution θ and a negative literal $L \equiv \neg L'$, we say that a pre-instance s *satisfies* $\text{INST}_\mathbf{s}(\theta(L))$ if it does not satisfy $\text{INST}_\mathbf{s}(\theta(L'))$.

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

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 along with substitution; and (ii) the weaker requirement on values of objects, refinement rather than equality.

Before giving the definition of the important notion of pre-model, we introduce a preliminary definition. Intuitively, we say that a pre-instance s is an *extension* of a pre-instance s_0 if s contains at least as much information as s_0 and possibly some more, that is, each relation and each class in s is a superset of the corresponding relation in class in s_0, and the oid's have comparable values.

Definition 5.3 [Extension] Given a scheme $\mathbf{S}$ and a pre-instance $s_0 = (\mathbf{c}_0, \mathbf{r}_0, \mathbf{f}_0, \mathbf{o}_0)$ of $\mathbf{S}$, we say that a pre-instance $s = (\mathbf{c}, \mathbf{r}, \mathbf{f}, \mathbf{o})$ of $\mathbf{S}$ is an *extension* of s_0 if:

(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 $\mathbf{c}_0(C)$;

(iii) for each oid $o \in \mathbf{c}_0(C)$, $\mathbf{o}(o)$ is a refinement of $\mathbf{o}_0(o)$; and

(iv) 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.

We say that a pre-instance $s = (\mathbf{c}, \mathbf{r}, \mathbf{f}, \mathbf{o})$ of $\mathbf{S}$ is a *proper extension* of $s_0 = (\mathbf{c}_0, \mathbf{r}_0, \mathbf{f}_0, \mathbf{o}_0)$ if s is an extension of s_0 and $s \neq s_0$ — that is, for some relation $R \in \mathbf{R}$ (or some class $C \in \mathbf{C}$), $\mathbf{r}(R)$ ($\mathbf{c}(C)$) is a proper superset of $\mathbf{r}_0(R)$ ($\mathbf{c}_0(C)$, respectively). $\square$

Definition 5.4 [Pre-Model] Given a program $\mathbf{P}$ over a scheme $\mathbf{S}$ and a pre-instance $s_0 = (\mathbf{c}_0, \mathbf{r}_0, \mathbf{f}_0, \mathbf{o}_0)$ of $\mathbf{S}$, we say that a pre-instance $s = (\mathbf{c}, \mathbf{r}, \mathbf{f}, \mathbf{o})$ of $\mathbf{S}$ is a *pre-model* for $\mathbf{P}$ over s_0 if

- s is an extension of s_0; and
- s satisfies each clause in $\mathbf{P}$.

$\square$

We have two results, dealing with the property of preserving oid-equivalence.

Lemma 5.5 *Let* $\mathbf{S}$ *be a scheme. If a pre-instance* s *is an extension of a pre-instance* s_0*, then:*

(i) for each pre-instance s'_0 *oid-equivalent to* s_0*, there is a pre-instance* s' *oid-equivalent to* s *such that* s' *is an extension of* s'_0*; and*

(ii) for each pre-instance s' *oid-equivalent to* s*, there is a pre-instance* s'_0 *oid-equivalent to* s_0 *such that* s' *is an extension of* s'_0*.*

Proof: We prove part (i). (The proof of part (ii) is similar.)

Let us consider a pre-instance s'_0 oid-equivalent to s_0, and a permutation σ of $\mathcal{O}$ such that $s'_0 = \sigma(s_0)$. We claim that the pre-instance defined by $s' = \sigma(s)$ is an extension of s'_0. Indeed, the permutation σ preserves: the superset relationships for (i) $\mathbf{r}$ and (ii) $\mathbf{c}$, the refinement for (iii) values in the image of $\mathbf{o}$, and the extension for (iv) $\mathbf{f}$. □

As a consequence, the notion of extension, originally defined for pre-instances, becomes meaningful also for instances. That is, given two pre-instances s and s_0, if s is an extension of s_0, then we can say that the instance $[s]$ is an *extension* of the instance $[s_0]$.

Lemma 5.6 *Let* $\mathbf{P}$ *be a program over a scheme* $\mathbf{S}$*. If the pre-instance* s *is a pre-model for* $\mathbf{P}$ *over the pre-instance* s_0*, then:*

(i) for each pre-instance s'_0 *oid-equivalent to* s_0*, there is a pre-model* s' *for* $\mathbf{P}$ *over* s'_0*, such that* s' *is oid-equivalent to* s*; and*

(ii) for each pre-instance s' *oid-equivalent to* s*, there is a pre-instance* s'_0 *oid-equivalent to* s_0 *such that* s' *is a pre-model for* $\mathbf{P}$ *over* s'_0*.*

Proof: We prove part (i). (The proof of part (ii) is similar.)

Consider a pre-instance s'_0 oid-equivalent to s_0, and a permutation σ of $\mathcal{O}$ such that $s'_0 = \sigma(s_0)$. Let s' be the pre-instance defined by $s' = \sigma(s)$. Because of Lemma 5.5 and the fact that s is an extension of s_0, it holds that the pre-instance defined by $s' = \sigma(s)$ is an extension of s'_0 oid-equivalent to s. Now we prove that s' is a pre-model for $\mathbf{P}$. For a clause $\gamma \in \mathbf{P}$ and a ground substitution θ' such that $s' \models \textsc{inst}_{s'}(\theta'(\textsc{body}(\gamma)))$, we have to prove that $s' \models \textsc{inst}_{s'}(\theta'(\textsc{head}(\gamma)))$ as well. For, consider the substitution $\theta = \sigma^{-1}(\theta')$; it is clearly the case that $s \models \textsc{inst}_{s}(\theta(\textsc{body}(\gamma)))$, hence $s \models \textsc{inst}_{s}(\theta(\textsc{head}(\gamma)))$. By applying σ to both members, it easily follows $s' \models \textsc{inst}_{s'}(\theta'(\textsc{head}(\gamma)))$. □

Because of Lemma 5.6 we can give a definition of *model* (with reference to instances) based on the definition of pre-model.

Definition 5.7 [Model, Minimal Model, Minimum Model] An instance $[s]$ is a *model* for a program $\mathbf{P}$ over an instance $[s_0]$ if s is a pre-model for $\mathbf{P}$ over s_0.

A model $[s]$ for $\mathbf{P}$ over $[s_0]$ is *minimal* if there is no other model $[s']$ for $\mathbf{P}$ over $[s_0]$ such that $[s]$ is a proper extension of $[s']$. If there is only one minimal model, then we call it the *minimum model.* □

Apart from technical aspects, the main difference with Datalog is the possibility that no model exists for an IsaLog program over an instance. There are two main reasons for this fact, corresponding to some of the extensions of the model and language with respect to the traditional Datalog framework, where minimum models always exist [15]. We present them in the following examples.

Recursion through oid invention can lead to the generation of infinite sets of facts, against the hypothesis of finite structures.

Example 5.8 [Finiteness] For example, given a class C, whose tuple type is $(C_{ref} : C)$, and the program that contains only the rule

$$\gamma : C(\text{OID} : F(C_{ref} : x), C_{ref} : x) \leftarrow C(\text{OID} : x, C_{ref} : y).$$

has no (finite) 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.

Example 5.9 [Functionality] Consider the following scheme:

CLASS person	*(name:Δ);*
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:

$$\begin{aligned} \textit{husband}(\text{OID}{:}x,\ \textit{name}{:}h,\ \textit{wife}{:}y) \ \leftarrow\ & \textit{marriage}(\textit{husband}{:}x,\ \textit{wife}{:}y), \\ & \textit{person}(\text{OID}{:}x,\ \textit{name}{:}h),\ \textit{person}(\text{OID}{:}y,\ \textit{name}{:}w). \end{aligned}$$

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. □

Definition 5.10 [Declarative Semantics] We define the *declarative semantics* of an IsaLog program **P** over a scheme **S** as a partial function D-SEM$_{\mathbf{P}}$ from instances of **S** to instances of **S**:

$$\text{D-SEM}_{\mathbf{P}}([\mathbf{s}]) = \begin{cases} \text{the minimum model of } \mathbf{P} \text{ over } [\mathbf{s}] & \text{if it exists} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

□

Examples 5.8 and 5.9 suggest two important properties for IsaLog programs: model finiteness and functionality. *Model finiteness* refers to the property of a program of having a finite model over every input instance. This is a strong requirement in a object-oriented database context, since the generation of an infinite number of new objects must be carefully avoided (because it would correspond to a non-terminating computation). On the other side, a program is said to be *functional* if it preserves the requirement that each object — existing or newly created — has a unique, well-defined, associated value. This is a desired property of programs, since the semantics of a non-functional program cannot be properly defined. We say that a program is *finite* (resp., *functional*) if admits a finite (resp., functional) model over every input instance — possibly allowing for non-functionality (resp., non-finiteness). Unfortunately, it turns out [12] that the problems of deciding whether a given program is finite or functional is in general unsolvable. In particural, functionality is undecidable even for programs without oid invention [2].

6 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 other semantics for IsaLog programs.

Given a scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$, the *Herbrand universe* $\mathcal{U}_\mathbf{S}$ for $\mathbf{S}$ is the set of all ground terms of $\mathbf{S}$. The *Herbrand base* $\mathcal{H}_\mathbf{S}$ for $\mathbf{S}$ is the set of all ground facts of the language, that is, the facts with predicate symbols from $\mathbf{R}$ and $\mathbf{C}$ and terms with function symbols from $\mathbf{F}$ and values from $\mathcal{O}$ and Δ. A *Herbrand interpretation* (or simply an *interpretation*) over $\mathbf{S}$ is a finite subset of the Herbrand base $\mathcal{H}_\mathbf{S}$.

Now 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:

   - 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 $\mathbf{r}(R)$;
   - 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 associated with the corresponding class.

2. $\phi(\mathbf{s})$ is defined as $\phi^1(\phi^0(\mathbf{s}))$, where ϕ^1 is a function that recursively replaces 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 uniquely 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 it is not surjective: there are Herbrand interpretations over $\mathbf{S}$ that are not in the image of ϕ. Given a scheme $\mathbf{S}$, we introduce five conditions, defined over interpretations, called *consistency constraints associated with* $\mathbf{S}$, and show that Herbrand interpretations belong to the image of ϕ if and only if they satisfy such constraints.

WT (*well-typedness*): for each fact, all attributes of the predicate symbol appear and the corresponding terms (nested functors included) have the appropriate type.

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

DIS (*disjointness*): for each oid term t_0 and each pair of classes C_1 and C_2, if both facts $C_1(\text{OID} : t_0, \ldots)$ and $C_2(\text{OID} : t_0, \ldots)$ appear, then 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 in a fact (or a functor) 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 $C'(\text{OID} : t'_0, \ldots, A : t', \ldots)$ and $C''(\text{OID} : t''_0, \ldots, A : t'', \ldots)$, with $t'_0 = t''_0$ and $t' \neq t''$. That is, two facts for the same oid term have respectively identical values for the common attributes.

Lemma 6.1 *A Herbrand interpretation over a scheme* $\mathbf{S}$ *satisfies the consistency constraints associated with* $\mathbf{S}$ *if and only if it belongs to the image of* ϕ *over the pre-instances of* $\mathbf{S}$.

Proof: Given a Herbrand interpretation $I_{\mathbf{S}}$ over $\mathbf{S}$, if $I_{\mathbf{S}}$ violates any of the consistency constraints, we can see that it does not correspond to a pre-instance. In fact:

- a violation of condition WT would imply a violation of typing in tuples;
- a violation of CON (respectively, DIS) would imply a violation of the first (respectively, the second) condition defined over $\mathbf{c}$;
- a violation of condition COH would imply the existence of a dangling reference in the pre-instance;
- finally, a violation of FUN would mean that the tuple the function $\mathbf{o}$ associates with an oid is not well defined, because it has two different values for the same attribute.

For the converse direction, given a Herbrand interpretation $I_{\mathbf{S}}$ satisfying the consistency constraints associated with $\mathbf{S}$, we show a pre-instance $\mathbf{s} = (\mathbf{c}, \mathbf{r}, \mathbf{f}, \mathbf{o})$ such that $\phi(\mathbf{s})$ equals $I_{\mathbf{S}}$.

First, we define a suitable $\mathbf{f}$, recursively replacing in $I_{\mathbf{S}}$ each different (flat) ground functor term $F(A_1 : v_1, \ldots, A_k : v_k)$ by a different (new) oid o and adding $F(A_1 : v_1, \ldots, A_k : v_k)$ to the domain of $\mathbf{f}(F)$, with image o. This replacement always terminates (by the boundedness of functor structures in $I_{\mathbf{S}}$), and the various requirements for the function $\mathbf{f}$ in a pre-instance are satisfied (because each different ground functor is replaced by a different oid). In this way, we obtain an interpretation $I'_{\mathbf{S}}$ corresponding to $I_{\mathbf{S}}$ and containing no functor terms. [3] This $I'_{\mathbf{S}}$ satisfies the consistency constraints, since $I_{\mathbf{S}}$ satisfies them.

Then, we define the other functions corresponding to $\mathbf{s}$:

- $\mathbf{c}$, such that an oid o belongs to $\mathbf{c}(C)$ iff a fact $C(\text{OID} : o, \ldots)$ exists in $I'_{\mathbf{S}}$;
- $\mathbf{r}$, such that a tuple $(A_1 : v_1, \ldots, A_k : v_k)$ belongs to $\mathbf{r}(R)$ iff a fact $R(A_1 : v_1, \ldots, A_k : v_k)$ exists in $I'_{\mathbf{S}}$;
- $\mathbf{o}$, such that $\mathbf{o}(o)(A)$ equals v iff a fact $C(\text{OID} : o, \ldots, A : v, \ldots)$ exists in $I'_{\mathbf{S}}$.

$\square$

Another property of function ϕ is that $\phi(\mathbf{s}_1) = \phi(\mathbf{s}_2)$ if and only if $\mathbf{s}_1$ and $\mathbf{s}_2$ are oid-equivalent pre-instances. Therefore, we can define a function Φ that maps instances to

[3] Let us note how this transformation is not univocally defined, because every possible choice of unused oid's in each step is admissible, leading to different but oid-equivalent interpretations.

Herbrand interpretations: $\Phi : [\mathbf{s}] \mapsto \phi(\mathbf{s})$. Since $\phi(\mathbf{s}_1)$ is equal 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 Herbrand interpretations that satisfy the consistency constraints. The inverse of Φ is therefore defined over Herbrand interpretations that satisfy the consistency constraints.

7 Reduction to Logic Programming

Given an IsaLog program $\mathbf{P}$ over a scheme $\mathbf{S}$ and a pre-instance $\mathbf{s}$ of $\mathbf{S}$, the function ϕ defined in the previous section allows to build the set of IsaLog clauses $\mathbf{P} \cup \phi(\mathbf{s})$, which is essentially a set of clauses of ordinary logic programming with function symbols [24]. [4]

The main difference with respect to Datalog or ILOG consists in the presence of hierarchies: isa relationships require generation of facts for the satisfaction of containment constraints — intuitively, facts corresponding to the propagation of oid's through class hierarchies. 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 [13]).

Definition 7.1 [Isa-clauses] Given a scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$, we define the *isa-clauses* $\Gamma_\mathbf{S}$ for $\mathbf{S}$ as follows:

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

□

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 an IsaLog 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 $\mathbf{P} \cup \Gamma_\mathbf{S} \cup \phi(\mathbf{s})$, which is essentially a set of clauses of ordinary logic programming with function symbols. Again, this set has a unique minimal (Herbrand) 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 Example 5.9; 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.

Definition 7.2 [Logic Programming Semantics] We define the *logic programming semantics (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):

$$\text{LP-SEM}_\mathbf{P}([\mathbf{s}]) = \begin{cases} \Phi^{-1}([\mathcal{M}_{\mathbf{P}\cup\Gamma_\mathbf{S}\cup\phi(\mathbf{s})}]) & \text{if } \mathcal{M}_{\mathbf{P}\cup\Gamma_\mathbf{S}\cup\phi(\mathbf{s})} \text{ is finite and} \\ & \text{satisfies condition FUN} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

□

[4]Since our model adopts a non-positional notation, we actually refer to a suitable *rewriting* of the clauses of the program in a positional notation. It can be shown, however, that the required rewriting can always be performed, so that in the following, we will refer interchangeably to a set of IsaLog$^{(\neg)}$ clauses and to its rewriting.

We will show in Section 9 that, for every IsaLog (positive) program **P** the declarative semantics and the LP-semantics coincide. It should be noted that this 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).

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. Second, and more important, as we have shown in Example 2.3, it cannot be directly extended to programs with negation: there are programs with a reasonable model that, if extended with isa clauses in order to deal with inheritance, are not stratified, and thus have no stratified semantics in the ordinary sense. We will present an alternative solution in Section 10.

8 Fixpoint semantics

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

Let an IsaLog scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$ be fixed. We say that an interpretation $I_\mathbf{S}$ *satisfies* a ground literal L if one of the following conditions holds:

- L is a positive literal, and $L \in I_\mathbf{S}$;
- L is a negative literal $\neg L'$, and $L' \notin I_\mathbf{S}$.

Similarly for a set of ground literals. 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 $\theta(\text{HEAD}(\gamma))$.

Given a scheme $\mathbf{S}$, let us now consider the Herbrand base $\mathcal{H}_\mathbf{S}$ associated with it, and the set $\mathcal{P}(\mathcal{H}_\mathbf{S})$ of all the possible interpretations over $\mathbf{S}$. We can easily show that, if we consider the partial order among interpretations defined by the containment relation, $\subseteq$, $(\mathcal{P}(\mathcal{H}_\mathbf{S}), \subseteq)$ is a *complete lattice* [5]. The main step in the definition of a fixpoint semantics is the introduction of a continuous transformation [5] over the lattice associated with a program.

The presence of isa requires a modification of the traditional approach, as follows.

Definition 8.1 [Isa Closure] Given a scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$ and the associated Herbrand base $\mathcal{H}_\mathbf{S}$ we define the *closure* T_{ISA} *with respect to* ISA as a mapping from $\mathcal{P}(\mathcal{H}_\mathbf{S})$ to itself, defined as follows:

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

□

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. We say that a fact α is an *isa-fact* of another fact β if α is derived from β by means of T_{ISA}.

Definition 8.2 [Immediate Consequence Operator] 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 $\mathcal{P}(\mathcal{H}_{\mathbf{S}})$ to itself, as follows:

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

Given a set of clauses Γ over a scheme $\mathbf{S}$, and an interpretation $I_{\mathbf{S}}$, the *immediate consequence operator* T_Γ associated with Γ is a mapping from interpretations to interpretations, defined as follows:

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

□

Let us note that in Datalog frameworks, the immediate consequence operator essentially coincides with our operator $T_{\Gamma,0}$.

We can prove that the transformation associated with a set of IsaLog clauses is *continuous* [5].

Lemma 8.3 *Let Γ be a set of* IsaLog *(positive) clauses over a scheme* $\mathbf{S}$. *Then, the transformation T_Γ is continuous.*

Proof: An operator $\mathbf{T}$ is said to be *continuous* [5] if it is *monotonic* and *finitary.* An operator T on a complete lattice is *monotonic* if, for all I, J, $I \subseteq J$ implies $T(I) \subseteq T(J)$. T is *finitary* if, for every infinite sequence of elements $I_0 \subseteq I_1 \subseteq \ldots$, it is the case that

$$T(\cup_{n=0}^{\infty} I_n) \subseteq \cup_{n=0}^{\infty} T(I_n)$$

Intuitively, monotonicity means that the operator preserves order, whereas continuity means that, given a growing sequence of elements converging towards a "limit", the sequence of transformed elements also converges towards an element that can be obtained by applying the operator T to the limit of the sequence.

First note that monotonicity is immediate by definition.

We now show that the transformation is finitary, that is, that for every infinite sequence of interpretations $I_0 \subseteq I_1 \subseteq \ldots$, it is the case that $T_\Gamma(\cup_{n=0}^{\infty} I_n) \subseteq \cup_{n=0}^{\infty} T_\Gamma(I_n)$. Let us consider a fact $\alpha \in T_\Gamma(\cup_{n=0}^{\infty} I_n)$ and show that there is an interpretation I_n in the sequence such that $\alpha \in T_\Gamma(I_n)$, so that $\alpha \in \cup_{n=0}^{\infty} T_\Gamma(I_n)$. If $\alpha \in T_\Gamma(\cup_{n=0}^{\infty} I_n)$, then, by definition of $T_{\Gamma,0}$ and T_{ISA}, there are a clause $\gamma \in \Gamma$ and a substitution θ such that $\cup_{n=0}^{\infty} I_n$ satisfies $\theta(\textsc{body}(\gamma))$ and either α is equal to $\theta(\textsc{head}(\gamma))$ or α is an isa-fact of $\theta(\textsc{head}(\gamma))$. This implies that for some I_n, namely the first one in the sequence containing all literals in $\theta(\textsc{body}(\gamma))$, it is the case that I_n satisfies $\theta(\textsc{body}(\gamma))$. So, $\alpha \in T_\Gamma(I_n)$. □

We now recall some definitions [5]. An interpretation I such that $I = T(I)$, is called a *fixpoint* of T. The *powers* of an operator T are defined as follows:

$$\begin{aligned} T{\uparrow}0(I) &= I \\ T{\uparrow}(n+1)(I) &= T(T{\uparrow}n(I)), \text{ for every } n \geq 0 \\ T{\uparrow}\omega(I) &= \cup_{n=0}^{\infty} T{\uparrow}n(I) \end{aligned}$$

As a consequence of Lemma 8.3, by Knaster-Tarski Theorem [5, p.517], we know that, if Γ is a set of IsaLog (positive) clauses, then:

- the transformation T_Γ has at least one fixed point,
- the set of the fixed points of T_Γ is a complete lattice,
- the least fixed point of T_Γ can be computed as $T_\Gamma\!\uparrow\!\omega(\emptyset)$.

In order to define the fixpoint semantics of ISALOG programs, we need to discuss whether the transformation $T_\Gamma\!\uparrow\!\omega$ preserves the conditions satisfied by interpretations that correspond to pre-instances.

Lemma 8.4 *For every scheme* **S** *and for every set of* ISALOG *(positive) clauses* Γ *over* **S***, the application of the transformation* $T_\Gamma\!\uparrow\!\omega$ *to a Herbrand interpretation over* **S** *that satisfies conditions* WT*,* CON*,* DIS*, and* COH *produces a Herbrand interpretation over* **S** *that also satisfies those conditions.*

Proof: It suffices to show that satisfaction of the conditions by an interpretation $I_{\mathbf{S}}$ implies satisfaction by $T_\Gamma(I_{\mathbf{S}})$. Satisfaction of condition CON comes because of the closure operator T_{ISA} in the definition of T_Γ. Satisfaction of condition DIS directly descends from the syntax of clauses in Γ; in fact:

- oid-invention clauses possibly generate new functor terms and, thus, new oid's;
- in specialization clauses, oid's assigned to subclasses come from superclasses in the body.

Since oid's can be assigned to classes in no other way, it cannot be the case that classes C and C' without a common ancestor, that is, belonging to different hierarchies, share an oid. The satisfaction of conditions COH and WT follows the well-typedness requirement over clauses in Γ; it imposes that, whenever an oid-term t_0 ranges over a class C in the head of some clause γ in Γ, it must be the case that:

- t_0 belongs to (at least) one class C' occurring in BODY(γ) (this rules out violations of oid-coherence);
- the class is appropriate with respect to C, that is, C' ISA C (this guarantees well-typedness, since the condition is trivially satisfied by values).

□

The above lemma does not say anything about condition FUN. Again, it is not in general preserved (see Example 5.9).

Definition 8.5 **[Fixpoint Semantics]** Given an ISALOG program **P** over a scheme **S**, we can define the *fixpoint semantics* of **P** as a partial function FP-SEM$_{\mathbf{P}}$ that maps instances to instances, using for each instance [s], the set of clauses $\mathbf{P} \cup \phi(\mathbf{s})$ (the program plus the interpretation corresponding to the pre-instance **s**), as follows:

$$\text{FP-SEM}_{\mathbf{P}}([\mathbf{s}]) = \begin{cases} \Phi^{-1}([T_{\mathbf{P}\cup\phi(\mathbf{s})}\!\uparrow\!\omega(\emptyset)]) & \text{if } T_{\mathbf{P}\cup\phi(\mathbf{s})}\!\uparrow\!\omega(\emptyset) \text{ is finite and} \\ & \qquad\qquad\qquad\text{satisfies condition FUN} \\ \textit{undefined} & \text{otherwise} \end{cases}$$

□

It can be shown that this definition is independent of the choice of the representative $\mathbf{s}$ of the instance, and therefore it is well formed.

The fixpoint semantics is the third semantics for IsaLog programs. In the next section we prove that the three defined semantics, the declarative semantics, the reduction to logic programming semantics and the fixpoint semantics are all equivalent, thus introducing a robust concept.

9 Equivalence of the various semantics for positive programs

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

Theorem 9.1 [Equivalence of the Semantics for Positive Programs] *For every positive* IsaLog *program* $\mathbf{P}$*, the following semantics coincide:*

- *the declarative semantics* $\text{D-SEM}_{\mathbf{P}}$
- *the logic programming semantics* $\text{LP-SEM}_{\mathbf{P}}$
- *the fixpoint semantics* $\text{FP-SEM}_{\mathbf{P}}$

Proof: See Appendix A. □

10 IsaLog with negation

In this section we deal with IsaLog$^{\neg}$ programs, that is, IsaLog programs in which negation is allowed in the body of rules.

The definition of the semantics of such programs requires a suitable notion of stratification, called *isa-coherent stratification*, which keeps into account the presence of isa hierarchies among classes in the scheme.

10.1 Isa-coherent Stratification

We need some preliminary definitions. Assume that a scheme $\mathbf{S} = (\mathbf{C}, \mathbf{R}, \mathbf{F}, \text{TYP}, \text{ISA})$ is fixed.

Definition 10.1 [Definition of a Predicate Symbol] Given a clause γ, we say that γ *defines* a predicate symbol Q (and also that Q *is defined by* γ) 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 a positive literal $C'(\text{OID} : t, \ldots)$ in $\text{BODY}(\gamma)$, with $C' \in \mathbf{C}$, such that $C' \text{ ISA } Q$.

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

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 fact for others already exists). Clearly, this distinction is relevant only for a language with class hierarchies: in languages without hierarchies, each clause defines exactly one predicate symbol.

Definition 10.2 [Isa-coherent Stratification] A partition $P_1 \dot\cup \ldots \dot\cup P_n$ of the clauses of **P** is an *isa-coherent stratification* of **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 a subset of $\cup_{j \leq i} P_j$; that is, the definition of Q is contained in the set of strata non higher than P_i;

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 a subset of $\cup_{j<i} P_j$, that is, the definition of Q is contained in the set of strata that are stricly lower than P_i.

An IsaLog$^\neg$ program **P** is *isa-coherently stratified* if it has an isa-coherent stratification. □

This is a generalization of the standard notion of stratification: in particular, it differs from that given by Apt [5, p.557] in the notion of "definition" of a predicate symbol Q within a program **P** (which, in Datalog, is the set of clauses in **P** whose head's predicate symbol is Q).

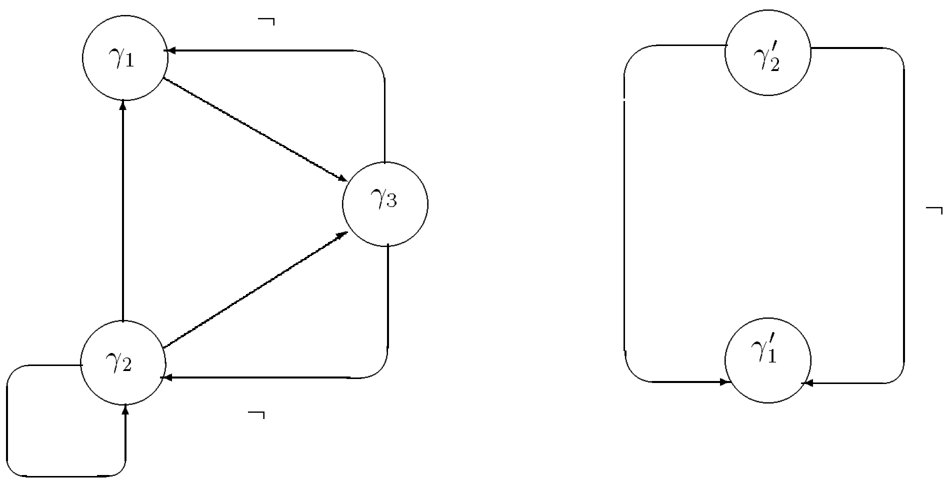
Isa-coherently stratified programs can be characterized by means of properties of clauses (rather than predicate symbols, as it happens in the Datalog framework). We need a few definitions.

Definition 10.3 [Clause Dependency Graph] 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 we say that γ_1 *negatively refers to* γ_2. Given a program **P** we define its *clause dependency graph* CDG$_\mathbf{P}$ as a directed graph representing the relation *refers to* between the clauses of **P**. An edge (γ_1, γ_2) is *negative* if γ_1 negatively refers to γ_2 (see Figure 3). □

It is easy to prove the following result, showing that isa-coherent stratification can be checked in polynomial time with respect to the size of the program.

Lemma 10.4 *A program* **P** *is isa-coherently stratified if and only if its clause dependency graph* CDG$_\mathbf{P}$ *does not contain a cycle with a negative edge.*

Example 10.5 [Clause Dependency Graph] Consider the rules in Example 2.2.



Clause Dependency Graph
for Example 10.5

Clause Dependency Graph
for Example 10.6

Figure 3: Examples of clause dependency graphs

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).*

In this case, rule γ_1 and γ_2 define predicate symbol *path*; rule γ_3 defines both *arc* and *new-arc*. Thus, the clause dependency graph is the one in Figure 3; it is easy to see that the program is not isa-coherently stratified, since the graph contains a cycle with a negative edge. □

Example 10.6 [Clause Dependency Graph (Continued)] Consider now the rules in Example 2.3.

γ'_1 : *rich-person(*OID*: x, name : n, asset : a, father : f)* $\leftarrow$
*person(*OID*: x, name : n, asset : a, father : f), a* $>$ *100K.*

γ'_2 : *self-made-man(*OID*: x, name : n, asset : a, father : f)* $\leftarrow$
*rich-person(*OID*: x, name : n, asset : a, father : f),*
$\neg$ *rich-person(*OID*: f, name : nf, asset : af, father : ff).*

In this case, rule γ'_1 defines *rich-person*; rule γ'_2 defines only *self-made-man*, and the clause dependency graph does not contain cycles with negative edges (see Figure 3). Thus, the program is isa-coherently stratified. □

10.2 Fixpoint Semantics of Isa-Coherently Stratified Programs

In this section we present the fixpoint semantics for ISALOG$^{\neg}$ isa-coherently stratified programs. It is defined following the same steps as in the traditional framework [5, 6]. 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. [5]

Consider an operator T on a complete lattice; the *cumulative powers* [5] of T are defined as follows:

$$\begin{array}{rcl} T{\Uparrow}0(I) & = & I \\ T{\Uparrow}(n+1)(I) & = & T(T{\Uparrow}n(I)) \cup T{\Uparrow}n(I), \text{ for every } n \geq 0 \\ T{\Uparrow}\omega(I) & = & \cup_{n=0}^{\infty} T{\Uparrow}n(I) \end{array}$$

First note that cumulative powers preserve conditions on Herbrand interpretations. The proof of the following lemma (which generalizes Lemma 8.4 to programs with negation) is rather straightforward, but rather intricate because of different cases, and therefore omitted.

Lemma 10.7 *Let* Γ *be a set of* ISALOG$^{\neg}$ *clauses over a scheme* **S**. *Then, the application of the transformation* $T_{\Gamma}{\Uparrow}\omega$ *to a Herbrand interpretation over* **S** *that satisfies conditions* WT, CON, DIS, *and* COH *with respect to* **S** *produces a Herbrand interpretation over* **S** *that also satisfies those conditions.*

Let us now consider an isa-coherently stratified program **P** over a scheme **S**. We use a stratification of **P** to build a meaningful Herbrand interpretation over **S**, which is then proven to be independent of the particular stratification.

Consider an isa-coherently stratified program **P** over a scheme **S**, an isa-coherent stratification $P_1, \ldots, P_n$ of **P**, an instance [s] of **S**, and the interpretation $\phi(\mathbf{s})$ corresponding to **s**. The immediate consequence operator T_{P_i} associated with a stratum P_i is defined as in Section 8, that is, $T_{P_i}(I_{\mathbf{S}}) = T_{\mathrm{ISA}}(T_{P_i,0}(I_{\mathbf{S}}))$, where now $T_{P_i,0}$ takes into account also the satisfaction of negative literals.

We define the following sequence of Herbrand interpretations over **S**:

$$\begin{array}{rcl} M_{0,\phi(\mathbf{s})} & = & \phi(\mathbf{s}) \\ M_{i,\phi(\mathbf{s})} & = & T_{P_i}{\Uparrow}\omega(M_{i-1,\phi(\mathbf{s})}), \quad \text{for } 1 \leq i \leq n \\ M_{\mathbf{P},\phi(\mathbf{s})} & = & M_{n,\phi(\mathbf{s})} \end{array}$$

The interpretation $M_{\mathbf{P},\phi(\mathbf{s})}$, obtained by sequentially applying each operator $T_{P_1}, \ldots, T_{P_n}$ plays a crucial role in the definition of the semantics of **P**. In fact, we can prove that $M_{\mathbf{P},\phi(s)}$ is a minimal fixpoint of the transformation associated with P over $[s]$.

Lemma 10.8 *Given an isa-coherently stratified program* **P** *over a scheme* **S** *and an instance* [s] *of* **S**, *for each isa-coherent stratification* $P_1, \ldots, P_n$, $M_{\mathbf{P},\phi(\mathbf{s})}$ *is a minimal fixpoint of the transformation* $T_{\mathbf{P}\cup\phi(\mathbf{s})}$ *associated with program* **P** *and instance* [s]. *Moreover, the fixpoint is independent on the chosen stratification.*

[5]In the following, we will not specify "isa-coherent" when clearly understood from the context.

Proof: See Appendix B. □

As a consequence, we can eliminate any reference to the specific stratification, thus motivating the notation $M_{\mathbf{P},\phi(\mathbf{s})}$. This property is similar to that arising in Datalog and leads to the following definition of *isa-coherently stratified semantics* ST-SEM of IsaLog$^\neg$ programs, as a partial function from instances to instances.

Definition 10.9 [Isa-coherently Stratified Semantics] Given an isa-coherently stratified program **P** over a scheme **S**, and an instance [s] of **S**, we define the *isa-coherently stratified semantics* ST-SEM of **P** over [s] as follows:

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

□

10.3 Reduction to LP for Programs with Negation

We have shown in Section 7 how an equivalent semantics for IsaLog positive programs can be defined as a reduction to logic programming. Given an IsaLog program **P** over a scheme **S** and an instance [s], this semantics is defined by means of three steps:

1. compute the Herbrand interpretation $\phi(\mathbf{s})$ associated with **s**;
2. compute the minimum model $\mathcal{M}$ of the logic program composed of: (i) (a syntactic variation of) the IsaLog program **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 **P** over [s] be $\Phi^{-1}([\mathcal{M}])$ (or, equivalently, $[\phi^{-1}(\mathcal{M})]$), otherwise let it be undefined.

It turns out that this approach is not satisfactory with respect to isa-coherent stratified IsaLog$^\neg$ programs. Consider the program in Example 2.3 in the introduction: 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,...)* ←*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, we can use additional clauses only with reference to the clauses of the program that require oid propagation. Specifically, for each clause γ that defines (according to the notion of definition of predicate symbol given in Definition 10.1 in Section 10.1) more than one predicate symbol, we add a set of clauses, defined as follows.

Definition 10.10 [Defined-symbol Clauses] Given a clause γ and a scheme $\mathbf{S}$, the *defined-symbol clauses of γ with respect to* $\mathbf{S}$) are the following clauses:

$$\{C_2(\text{OID} : t_0, A_1 : x_1, \ldots, A_k : x_k) \leftarrow \text{BODY}(\gamma) \mid$$
$$\text{HEAD}(\gamma) = C_1(\text{OID} : t_0, A_1 : x_1, \ldots, A_{k+h} : x_{k+h}),$$
$$C_1 \text{ ISA } C_2,\ C_2(\neq C_1) \text{ is a defined symbol of } \gamma,$$
$$\text{TYP}(C_2) = (A_1 : \tau_1, \ldots, A_k : \tau_k) \text{ and } \text{TYP}(C_1) = (A_1 : \tau_1, \ldots, A_{k+h} : \tau_{k+h})\}$$

□

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, 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-a) the isa clauses associated with the scheme (for the IC-semantics) or (iii-b) 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 [s] be $\Phi^{-1}([\mathcal{M}])$, otherwise let it be undefined.

It is easy to see that if the IC-reduction of a program is stratified, then also the DS-reduction is stratified. In fact, the transitive closure of the relation "refers to" among predicate symbols in the DS-reduction is always a subset of the corresponding relation among predicate symbols in the IC-reduction.

Indeed, the DS-reduction generalizes the notion of IC-reduction. In fact, the next theorem states the equivalence of DS-semantics and the stratified semantics based on the notion of isa-coherent stratification.

Theorem 10.11 [Equivalence of the Stratified Semantics] *For every scheme* $\mathbf{S}$ *and for every* ISALOG$^\neg$ *program* $\mathbf{P}$*:*

1. *the DS-reduction of* $\mathbf{P}$ *is stratified if and only if* $\mathbf{P}$ *is isa-coherently stratified;*

2. *the isa-coherently stratified semantics* ST-SEM$_\mathbf{P}$ *and the DS-semantics of* $\mathbf{P}$ *coincide.*

Proof: We need to introduce some notation. Given a set Γ of ISALOG$^{(\neg)}$ clauses over a scheme $\mathbf{S}$, let us denote with DS-RED$_\mathbf{S}(\Gamma)$ the DS-reduction of Γ with respect to $\mathbf{S}$, that is, Γ plus its defined-symbol clauses.

First, we prove part 1. Suppose that $\mathbf{P}$ is isa-coherently stratified, so that it has an isa-coherent stratification $P_1 \dot\cup \ldots \dot\cup P_n$. It follows from the definitions that DS-RED$(P_1) \dot\cup \ldots \dot\cup$ DS-RED(P_n) is a stratification for DS-RED$(\mathbf{P})$, so that the DS-reduction of $\mathbf{P}$ is also stratified. Now suppose that $\mathbf{P}$ is not isa-coherently stratified; let us consider a partition $P_1, \ldots, P_n$ of $\mathbf{P}$ and suppose that, for some i, there exists a clause $\gamma \in P_i$, such that a negative (positive) literal whose predicate symbol is Q occurs in $\textsc{body}(\gamma)$ and the definition of Q is not contained within $\cup_{j<i}P_j$ ($\cup_{j\leq i}P_j$). This happens if and only if there exists a clause $\gamma' \in P_k$, with $k \geq i$ ($k > i$), such that γ' defines Q. This implies that the partition $\textsc{ds-red}(P_1) \dot{\cup} \ldots \dot{\cup}\ \textsc{ds-red}(P_n)$ does not represent a stratification for $\textsc{ds-red}(\mathbf{P})$, since there is a clause $\widehat{\gamma'} \in \textsc{ds-red}(P_k)$, whose head predicate symbol is Q, such that the stratification policy is violated. Moreover, suppose by the way of contradiction, that we can move the clause $\widehat{\gamma'}$ to some stratum h, with $h < i$ ($h \leq i$), in order to enforce the stratification policy and obtain a partition that represents a stratification for $\textsc{ds-red}(\mathbf{P})$. But if it is possible to do this, it is also possible to move all the defined symbol clauses of γ' to stratum h, thus obtaining another stratification of $\textsc{ds-red}(\mathbf{P})$. But this implies that it is possible to move clause γ' to stratum P_h, obtaining an isa-coherent stratification for $\mathbf{P}$, against the hypothesis.

To prove part 2, consider an isa-coherent stratified program $\mathbf{P}$ and choose a stratification $P_1 \dot{\cup} \ldots \dot{\cup}\ P_n$ of $\mathbf{P}$. For each $i \in \{1, \ldots, n\}$, consider the immediate consequence operator T_{P_i}. The claim is that T_{P_i} is equivalent to the immediate consequence operator $\mathcal{T}_{P'_i}$, as defined for ordinary logic programs with negation, where P'_i equals $\textsc{ds-red}(P_i)$. It suffices to show that, for every Herbrand interpretation $I_\mathbf{S}$ over $\mathbf{S}$, it holds that $T_{P_i}(I_\mathbf{S})$ equals $\mathcal{T}_{P'_i}(I_\mathbf{S})$. A fact α belongs to $T_{P_i}(I_\mathbf{S})$ if and only if it belongs to $\mathcal{T}_{P'_i}(I_\mathbf{S})$ because of one of the following conditions:

- $\alpha \in I_\mathbf{S}$, so that it belongs to both $T_{P_i}(I_\mathbf{S})$ and $\mathcal{T}_{P'_i}(I_\mathbf{S})$;
- there exist a clause $\gamma \in P_i$ and a substitution θ such that $I_\mathbf{S}$ satisfies $\theta(\textsc{body}(\gamma))$, and α equals $\theta(\textsc{head}(\gamma))$, and then it belongs to both $T_{P_i}(I_\mathbf{S})$ and $\mathcal{T}_{P'_i}(I_\mathbf{S})$;
- the same as the previous reason, except for having $\theta(\textsc{head}(\gamma))$ equals α' and α being an isa-fact of α'. In this case α belongs to $T_{P_i}(I_\mathbf{S})$ because of the closure with respect to isa in the definition of T_{P_i}, whereas it belongs to $\mathcal{T}_{P'_i}(I_\mathbf{S})$ because of the existence of another clause $\gamma' \in P'_i$ originated as a defined-symbol clause of γ. The clause to be considered is the one having in the head the same predicate symbol as α. Moreover, the body of this clause is satisfied by $I_\mathbf{S}$ because it is the same as the body of γ.

□

It is worth noting that the DS-reduction relies upon 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:*Δ*)* and *(name:*Δ*,id-number:*Δ*)*, and a program with an oid-invention clause:

$$\gamma : \mathit{student}(\textsc{oid} : f_{\mathit{student}}(\mathit{name} : n, \mathit{id\text{-}number} : \mathit{id}), \mathit{name} : n, \mathit{id\text{-}number} : \mathit{id}) \leftarrow \textsc{body}(\gamma)$$

The DS-reduction introduces a clause

$$\gamma' : \mathit{person}(\textsc{oid} : f_{\mathit{student}}(\mathit{name} : n, \mathit{id\text{-}number} : \mathit{id}), \mathit{name} : n) \leftarrow \textsc{body}(\gamma)$$

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. Note that, if implicit functors were used, the clause would produce two different functor terms for the two classes.

11 Conclusions

The IsaLog$^{(\neg)}$ model and languages have been presented.

The IsaLog data model is (structurally) object–oriented, including classes, isa hierarchies among them, and relationships. Object identity is supported, and the use of memorized Skolem functors lets objects carry information about their creation.

Two languages have been presented, namely IsaLog and IsaLog$^{\neg}$. Both of them are rule-based, the former referring to a positive framework, whereas the latter allows for the use of negation in body of rules. Rules in programs allow for intensional definitions of relations (relation clauses) and of extensions of classes. Oid-invention clauses are object generating, in the sense that they allow for the specification of newly generated objects in the database. This generation is governed using the technique of explicit Skolem functors. On the other hand, specialization clauses are object preserving, and are used to populate classes in hierarchies of existing objects, specifying additional properties.

For IsaLog (positive) programs, three different semantics have been provided and proven to be equivalent to each other. The model-theoretic semantics is purely declarative and based on a notion of a model of a program. The logic-programming semantics reduces the problem of computing a model of an IsaLog program to the (more traditional) problem of computing a model of a logic program with function symbols, in which some special rules are used to enforce the constraints associated with inheritance. Finally, the fixpoint semantics is operational, based on an immediate consequence operator.

The introduction of negation in this framework has been exploited in the IsaLog$^{\neg}$ language. In this context, an original notion of stratification, called isa-coherent stratification, which takes into account the presence of isa hierarchies, has been defined. For this class of programs two different semantics have been proposed and proven equivalent. One is a reduction to logic programming with function symbols, and the other a fixpoint semantics.

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

[3] 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.

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

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

[6] 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 Kaufmann, San Mateo, California, 1988.

[7] P. Atzeni, editor. *LOGIDATA+: Deductive Databases with Complex Objects, Lecture Notes in Computer Science 701.* Springer-Verlag, 1993.

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

[9] P. Atzeni, L. Cabibbo, and G. Mecca. IsaLog$^{\neg}$: a deductive language with negation for complex-object databases with hierarchies. In *Third Int. Conf. on Deductive and Object-Oriented Databases , Lecture Notes in Computer Science, 760*, pages 222–235. Springer-Verlag, 1993.

[10] J. Banerjee et al. Data model issues for object–oriented applications. *ACM Trans. on Off. Inf. Syst.*, 5(1):3–26, January 1987.

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

[12] L. Cabibbo and G. Mecca. Model finiteness and functionality in a declarative language with oid invention. Technical Report n. RT-INF-3-1996, Dipartimento di Discipline Scientifiche, Università degli Studi di Roma Tre, 1996.

[13] 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.

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

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

[16] 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.

[17] 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.

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

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

[20] 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.

[21] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. Technical Report 93/06, SUNY at Stony Brook, 1993.

[22] 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.

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

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

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

[26] 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.

[27] J. Mylopoulos, P.A. Bernstein, and E. Wong. A language facility for designing database-intensive applications. *ACM Trans. on Database Syst.*, 5(2):185–207, June 1980.

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

A Equivalence of the Semantics for Positive Programs

The proofs of the equivalence results of Theorem 9.1 are given in the following subsections.

A.1 Equivalence of fixpoint semantics and logic programming semantics

In order to prove the equivalence of fixpoint and logic programming semantics of IsaLog programs, we need some preliminary definitions and lemmas.

Lemma A.1 **([5])** *Let T be a monotonic operator. Then, for any $n \geq 0$,*

$$T{\uparrow}n(\emptyset) \subseteq T{\uparrow}(n+1)(\emptyset).$$

The logic programming semantics of Γ is defined as the semantics of the logic program Γ' obtained by adding the isa clauses $\Gamma_{\mathbf{S}}$ for $\mathbf{S}$ to Γ. We now consider the ordinary fixpoint semantics for Γ', which is based on the immediate consequence operator $\mathcal{T}_{\Gamma'}$, [6] defined as follows:

$$\mathcal{T}_{\Gamma'}(I_{\mathbf{S}}) = \{\theta(\textsc{head}(\gamma)) \mid \gamma \in \Gamma', I_{\mathbf{S}} \text{ satisfies } \theta(\textsc{body}(\gamma)) \text{ for a substitution } \theta\}.$$

The powers of the operator $\mathcal{T}_{\Gamma'}$ are defined as for T_Γ.

The relationship between the two operators is highlighted in the following:

Lemma A.2 *Let* $\mathbf{S}$ *be a scheme,* Γ *a set of* IsaLog *clauses* Γ *over* $\mathbf{S}$, $\Gamma_{\mathbf{S}}$ *the isa clauses for* $\mathbf{S}$*, and* Γ' *the set of clauses* $\Gamma \cup \Gamma_{\mathbf{S}}$. *Then, for every $n \geq 0$ it holds that:*

$$\mathcal{T}_{\Gamma'}{\uparrow}n(\emptyset) \subseteq T_\Gamma{\uparrow}n(\emptyset) \subseteq \mathcal{T}_{\Gamma'}{\uparrow}2n(\emptyset).$$

Proof: The proof is by induction on n.
(*Basis step*) For $n = 0$, it is the case that $\mathcal{T}_{\Gamma'}{\uparrow}0(\emptyset) = T_\Gamma{\uparrow}0(\emptyset) = \emptyset$.

(*Induction step*) Assume the formula holds for n, we show that it holds for $n+1$, that is, $\mathcal{T}_{\Gamma'}{\uparrow}(n+1)(\emptyset) \subseteq T_\Gamma{\uparrow}(n+1)(\emptyset) \subseteq \mathcal{T}_{\Gamma'}{\uparrow}(2n+2)(\emptyset)$.

For the first inclusion, suppose $\alpha \in \mathcal{T}_{\Gamma'}{\uparrow}(n+1)(\emptyset)$; we can distinguish two cases:

- there exists a clause $\gamma \in \Gamma$ and a substitution θ such that α equals $\theta(\textsc{head}(\gamma))$, $\theta(\textsc{body}(\gamma)) = \{\alpha_1, \ldots, \alpha_p\}$, with $\alpha_1, \ldots, \alpha_p \in \mathcal{T}_{\Gamma'}{\uparrow}n(\emptyset)$. By the induction hypothesis, $\alpha_1, \ldots, \alpha_p \in T_\Gamma{\uparrow}n(\emptyset)$ as well, so that the same clause γ and the same substitution θ allow to derive $\alpha \in T_\Gamma{\uparrow}(n+1)(\emptyset)$;

- α is an isa-fact of some fact $\alpha' \in \mathcal{T}_{\Gamma'}{\uparrow}n(\emptyset)$; in this case, α is derived at step $n+1$ by means of an isa-clause $\gamma \in \Gamma_{\mathbf{S}}$ and a substitution θ. In this case, by the induction hypothesis and the closure with respect to isa in the definition of T_Γ, $\alpha', \alpha \in T_\Gamma{\uparrow}n(\emptyset)$. Then, by Lemma A.1, $\alpha \in T_\Gamma{\uparrow}(n+1)(\emptyset)$ as well.

For the second inclusion, suppose $\alpha \in T_\Gamma{\uparrow}(n+1)(\emptyset)$. Again, we can distinguish two cases:

[6] In the following, we will use the symbol T to denote an immediate consequence operator as in the IsaLog framework, whereas the symbol $\mathcal{T}$ denotes operators as in the traditional logic programming setting.

- α is a fact derived because of a clause $\gamma \in \Gamma$ and a substitution θ such that α equals $\theta(\text{HEAD}(\gamma))$, $\theta(\text{BODY}(\gamma)) = \{\alpha_1, \ldots, \alpha_p\}$, with $\alpha_1, \ldots, \alpha_p \in T_\Gamma \uparrow n(\emptyset)$. By the induction hypothesis, $\alpha_1, \ldots, \alpha_p \in \mathcal{T}_{\Gamma'} \uparrow 2n(\emptyset)$, so that α can be derived in $\mathcal{T}_{\Gamma'} \uparrow (2n+1)(\emptyset)$. Then, by Lemma A.1, $\alpha \in \mathcal{T}_{\Gamma'} \uparrow (2n+2)(\emptyset)$ as well.

- reasoning as in the previous item, except for having $\theta(\text{HEAD}(\gamma)) = \alpha'$ and α being an isa-fact of α' (that is, α is derived because of the closure with respect to ISA of T_Γ). In this case, γ and θ allow to derive $\alpha' \in \mathcal{T}_{\Gamma'} \uparrow (2n+1)(\emptyset)$, and $\alpha \in \mathcal{T}_{\Gamma'} \uparrow (2n+2)(\emptyset)$ because of an isa-clause in Γ'.

□

Lemma A.3 *Let* **P** *be an* ISALOG *program over a scheme* **S**, *and* [s$_0$] *an instance of* **S**. *Then, the logic-programming semantics* [s] = LP-SEM$_{\mathbf{P}}$([s$_0$]) *of* **P** *over* [s$_0$] *is defined if and only if the fixpoint semantics* FP-SEM$_{\mathbf{P}}$([s$_0$]) *of* **P** *over* [s$_0$] *is also defined and coincides with* [s].

Proof: Consider the ISALOG set of clauses $\Gamma = \mathbf{P} \cup \phi(\mathrm{s}_0)$, and let Γ' be the set of clauses $\Gamma \cup \Gamma_{\mathbf{S}}$, i.e., Γ extended with the isa-clauses for **S**. If operator T_Γ (operator $\mathcal{T}_{\Gamma'}$, resp.) has a finite fixpoint, it is reached in a finite number of steps, say n. Because of Lemma A.2, $\mathcal{T}_{\Gamma'}$ (T_Γ, resp.) has a finite fixpoint that is reached in at most $2n+2$ (n, resp.) steps; moreover, the two fixpoints coincide.

Similarly, if one of the two operators has no finite fixpoint, the same holds for the other.
□

A.2 Equivalence of the fixpoint semantics and the declarative semantics

The proof is made of two steps. First, we prove that if the declarative semantics of a program over an instance is defined, then the fixpoint semantics is also defined and the two semantics coincide. Then, we will prove the converse.

We first introduce some preliminary results, needed in order to establish a connection among a model [s] of a program **P** over an instance [s$_0$] and a fixpoint I of the trasformation $T_{\mathbf{P} \cup \phi(\mathrm{s}_0)}$.

Lemma A.4 *Let* **P** *be an* ISALOG *program over a scheme* **S**, *and* [s$_0$] *an instance of* **S**. *The following properties hold:*

1. *if* [s] *is a model for* **P** *over* [s$_0$], *then, for each representative pre-instance* s$_0$ *of* [s$_0$], *there is a representative pre-instance* s *of* [s] *such that* $\phi(\mathrm{s})$ *is a fixpoint for* $T_{\mathbf{P} \cup \phi(\mathrm{s}_0)}$;

2. *if* I *is a finite fixpoint for* $T_{\mathbf{P} \cup \phi(\mathrm{s}_0)}$ *and satisfies the consistency constraints, then* $\Phi^{-1}(I)$ *is a model for* **P** *over* [s$_0$].

Proof: Let us first prove part 1. Consider a program **P** over a scheme **S** and an instance [s$_0$] of **S**. Suppose [s] is a model for **P** over [s$_0$]. We take a representative pre-instance s$_0$ of [s$_0$] and a representative pre-instance s of [s] such that s is an extension of s$_0$.

We need to prove that $\phi(\mathbf{s})$ is a fixpoint for $T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}$. Let us first note that $\phi(\mathbf{s})$ trivially satisfies the consistency constraints, since it is the image of a pre-instance. Moreover, since $\phi(\mathbf{s})$ satisfies condition CON, T_{ISA} is the identity on $\phi(\mathbf{s})$. So, in order to prove that $\phi(\mathbf{s})$ is a fixpoint for $T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}$, it suffices to prove that $\phi(\mathbf{s})$ is a fixpoint for $T_{\mathbf{P}\cup\phi(\mathbf{s}_0),0}$.

For each clause $\gamma \in \mathbf{P} \cup \phi(\mathbf{s}_0)$, we have to prove that, whenever for some substitution θ, $\phi(\mathbf{s}) \models \theta(\text{BODY}(\gamma))$, then $\phi(\mathbf{s}) \models \theta(\text{HEAD}(\gamma))$, that is, $\theta(\text{HEAD}(\gamma)) \in \phi(\mathbf{s})$.

Consider a clause $\gamma \in \mathbf{P} \cup \phi(\mathbf{s}_0)$; we can distinguish two cases, as follows.

If γ is a ground fact $\alpha \in \phi(\mathbf{s}_0)$, then $\phi(\mathbf{s}) \models \theta(\text{BODY}(\gamma))$, trivially, and $\theta(\text{HEAD}(\gamma)) = \alpha \in \phi(\mathbf{s})$, since $\mathbf{s}$ is an extension of $\mathbf{s}_0$.

On the other side, if γ is a clause in $\mathbf{P}$, then suppose that, for some substitution θ, $\phi(\mathbf{s}) \models \theta(\text{BODY}(\gamma))$, that is, $\theta(\text{BODY}(\gamma)) \subseteq \phi(\mathbf{s})$. Consider now a substitution θ' such that:

$$\theta(\text{BODY}(\gamma)) = \phi^1(\text{INST}_{\mathbf{s}}(\theta'(\text{BODY}(\gamma))))$$

Such a substitution can be easily obtained from θ.

We have that $\theta(\text{BODY}(\gamma)) = \phi^1(\text{INST}_{\mathbf{s}}(\theta'(\text{BODY}(\gamma)))) \subseteq \phi(\mathbf{s})$. By applying the transformation ϕ^{-1} to both members, we obtain that:

$$\text{INST}_{\mathbf{s}}(\theta'(\text{BODY}(\gamma))) \subseteq \phi^0(\mathbf{s})$$

which means that $\mathbf{s} \models \text{INST}_{\mathbf{s}}(\theta'(\text{BODY}(\gamma)))$. Since, by definition of model, $[\mathbf{s}]$ satisfies γ, we know that $\mathbf{s} \models \text{INST}_{\mathbf{s}}(\theta'(\text{HEAD}(\gamma)))$, that is, $\text{INST}_{\mathbf{s}}(\theta'(\text{HEAD}(\gamma))) \in \phi^0(\mathbf{s})$; by applying again the transformation ϕ^1 to both members, we can conclude that $\theta(\text{HEAD}(\gamma)) \in \phi(\mathbf{s})$, which proves the claim.

We now prove part 2, that is, if I is a finite fixpoint for $T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}$ and satisfies the consistency constraints, then $\Phi^{-1}(I)$ is a model for $\mathbf{P}$ over $[\mathbf{s}_0]$.

Let us call $[\mathbf{s}']$ the instance obtained from I by means of the transformation Φ^{-1} (Φ^{-1} is defined over I by Lemma 6.1). Such an instance is a model for $\mathbf{P}$ over $[\mathbf{s}_0]$. In fact, consider representative pre-instances $\mathbf{s}' = (\mathbf{c}, \mathbf{r}, \mathbf{f}, \mathbf{o})$ of $[\mathbf{s}]$ and $\mathbf{s}_0 = (\mathbf{c}_0, \mathbf{r}_0, \mathbf{f}_0, \mathbf{o}_0)$ of $[\mathbf{s}_0]$.

We first note that $\mathbf{s}'$ is an extension of $\mathbf{s}_0$. With respect to the definition of extension, the proof of items (i) and (ii) is straightforward since we know that $\phi(\mathbf{s}_0) \subseteq \phi(\mathbf{s}') = I$ (I is a fixpoint for $T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}$); condition (iii) holds since, for each oid o in $\mathbf{s}_0$, we know that:

- $\mathbf{o}'(o)$ is defined and the set $\mathcal{F}_{o,\mathbf{s}_0} = \{\alpha \mid \alpha = C(\text{OID} : o, \ldots) \in \phi(\mathbf{s}_0), C \in \mathbf{C}\}$, that is, the set of all facts in $\phi(\mathbf{s}_0)$ that specify the classes o belongs to, is a subset of the corresponding set of facts $\mathcal{F}'_{o,\mathbf{s}} \subseteq \phi(\mathbf{s}')$;
- $\phi(\mathbf{s}')$ satisfies conditions CON and COH.

Finally, note that condition (iv) holds as well since we can choose $\mathbf{s}'$ such that for each functor $F \in \mathbf{F}$, the function $\mathbf{f}(F)$ is a convenient extension of $\mathbf{f}_0(F)$.

We now show that $\mathbf{s}'$ satisfies each clause $\gamma \in \mathbf{P}$; in fact, suppose that, for some clause $\gamma \in \mathbf{P}$ and substitution θ, $\mathbf{s}' \models \text{INST}_{\mathbf{s}'}(\theta(\text{BODY}(\gamma)))$. Consider a substitution θ' such that:

$$\theta'(\text{BODY}(\gamma)) = \phi^1(\text{INST}_{\mathbf{s}}(\theta(\text{BODY}(\gamma))))$$

Such a substitution can be easily obtained from θ. Then, from the fact that

$$\mathbf{s}' \models \text{INST}_{\mathbf{s}'}(\theta(\text{BODY}(\gamma)))$$

we can derive that:

$$\text{INST}_{\mathbf{s}'}(\theta(\text{BODY}(\gamma))) \subseteq \phi^0(\mathbf{s}')$$

By applying the transformation ϕ^1 to both members, we obtain that

$$\phi^1(\text{INST}_{\mathbf{s}}(\theta(\text{BODY}(\gamma)))) \subseteq \phi(\mathbf{s}'),$$

that is, $\theta'(\text{BODY}(\gamma)) \subseteq \phi(\mathbf{s}')$.

Since $\phi(s')$ is a fixpoint for $T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}$, we can derive that $\theta'(\text{HEAD}(\gamma)) \in \phi(\mathbf{s}')$, which, in turn, can be written as $\phi^1(\text{INST}_{\mathbf{s}'}(\theta(\text{HEAD}(\gamma))) \in \phi(\mathbf{s}')$.

By applying the transformation ϕ^{-1} to both members, it follows that

$$\text{INST}_{\mathbf{s}'}(\theta(\text{HEAD}(\gamma))) \in \phi^0(\mathbf{s}')$$

that is, $\mathbf{s}' \models \text{INST}_{\mathbf{s}'}(\theta(\text{HEAD}(\gamma)))$, which proves the claim. □

We can now prove that, if the declarative semantics of a program over an instance is defined, then the fixpoint semantics is also defined and the two semantics coincide. Then we will prove the converse.

Lemma A.5 *Let* $\mathbf{P}$ *be an* ISALOG *program over a scheme* $\mathbf{S}$*, and* $[\mathbf{s}_0]$ *an instance of* $\mathbf{S}$*. If the declarative semantics* $[\mathbf{s}] = \text{D-SEM}_{\mathbf{P}}([\mathbf{s}_0])$ *of* $\mathbf{P}$ *over* $[\mathbf{s}_0]$ *is defined, then the fixpoint semantics* $\text{FP-SEM}_{\mathbf{P}}([\mathbf{s}_0])$ *of* $\mathbf{P}$ *over* $[\mathbf{s}_0]$ *is also defined and it coincides with* $[\mathbf{s}]$*.*

Proof: Suppose the declarative semantics $[\mathbf{s}] = \text{D-SEM}_{\mathbf{P}}([\mathbf{s}_0])$ of $\mathbf{P}$ over $[\mathbf{s}_0]$ is defined. We need to prove that the fixpoint semantics $\text{FP-SEM}_{\mathbf{P}}([\mathbf{s}_0])$ of $\mathbf{P}$ over $[\mathbf{s}_0]$ is also defined and coincides with $[\mathbf{s}]$.

The fixpoint semantics of $\mathbf{P}$ over $[\mathbf{s}_0]$ is defined as $\Phi^{-1}(T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}\uparrow\omega(\emptyset))$, if $T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}\uparrow\omega(\emptyset)$, that is, the least fixpoint of $T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}$, is finite and satisfies condition FUN.

Let us consider representative pre-instances $\mathbf{s}_0$ of $[\mathbf{s}_0]$ and $\mathbf{s}$ of $[\mathbf{s}]$ such that $\mathbf{s}$ is an extension of $\mathbf{s}_0$. In order to prove the lemma, we show that $\phi(\mathbf{s})$ is the least fixpoint of $T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}$.

We know by lemma A.4 that $\phi(\mathbf{s})$ is a fixpoint for $T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}$. We need to prove that $\phi(\mathbf{s})$ is actually the least fixpoint of the trasformation $T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}$.

By way of contradiction, suppose that $T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}$ admits a least fixpoint $I = T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}\uparrow\omega(\emptyset)$ which is properly contained in $\phi(\mathbf{s})$. We note that I satisfies the consistency constraints, since:

- it satisfies conditions WT, COH, CON, and DIS by Lemma 8.4;
- condition FUN is satisfied since I is a proper subset of $\phi(\mathbf{s})$.

This means that we can find the instance $[\mathbf{s}'] = \Phi^{-1}(I)$ such that $[\mathbf{s}]$ is a proper extension of $[\mathbf{s}']$. Such an instance, by Lemma A.4, is a model for $\mathbf{P}$ over $[\mathbf{s}_0]$. This means that there is a model for $\mathbf{P}$ over $[\mathbf{s}_0]$ such that $[\mathbf{s}]$ is a proper extension of it. But, by definition of declarative semantics, $[\mathbf{s}]$ is the minimum model for $\mathbf{P}$ over $[\mathbf{s}_0]$ and such a model cannot exist. This proves the claim. □

We now prove the converse, that is, that if the fixpoint semantics of a program over an instance is defined, then the declarative semantics is also defined and the two semantics coincide.

Lemma A.6 *For each* ISALOG *program* $\mathbf{P}$ *over a scheme* $\mathbf{S}$ *and an instance* $[\mathbf{s}_0]$ *of* $\mathbf{S}$*, if the fixpoint semantics* $[\mathbf{s}] = \text{FP-SEM}_{\mathbf{P}}([\mathbf{s}_0])$ *of* $\mathbf{P}$ *over* $[\mathbf{s}_0]$ *is defined, then the declarative semantics* $\text{D-SEM}_{\mathbf{P}}([\mathbf{s}_0])$ *of* $\mathbf{P}$ *over* $[\mathbf{s}_0]$*, is also defined and it coincides with* $[\mathbf{s}]$.

Proof: Let us consider an ISALOG program $\mathbf{P}$ over a scheme $\mathbf{S}$ and an instance $[\mathbf{s}_0]$ of $\mathbf{S}$. Suppose the fixpoint semantics $[\mathbf{s}] = \text{FP-SEM}_{\mathbf{P}}([\mathbf{s}_0])$ of $\mathbf{P}$ over $[\mathbf{s}_0]$ is defined. We need to prove that $[\mathbf{s}]$ is the minimum model of $\mathbf{P}$ over $[\mathbf{s}_0]$.

We know by Lemma A.4 that $[\mathbf{s}]$ is a model for $\mathbf{P}$ over $[\mathbf{s}_0]$. We claim that $[\mathbf{s}]$ is the minimum model of $\mathbf{P}$ over $[\mathbf{s}_0]$.

Suppose, by way of contradiction, that there exists a model $[\mathbf{s}']$ for $\mathbf{P}$ over $[\mathbf{s}_0]$ such that $[\mathbf{s}]$ is a proper extension of $[\mathbf{s}']$. Taken a representative pre-instance $\mathbf{s} = (\mathbf{c}, \mathbf{r}, \mathbf{f}, \mathbf{o})$ of $[\mathbf{s}]$, we can find a representative pre-instance $\mathbf{s}' = (\mathbf{c}', \mathbf{r}', \mathbf{f}', \mathbf{o}')$ of $[\mathbf{s}']$ such that $\mathbf{s}$ is a proper extension of $\mathbf{s}'$, that is, $\phi(\mathbf{s}') \subset \phi(\mathbf{s})$. Since $[\mathbf{s}']$ is a model for $\mathbf{P}$ over $[\mathbf{s}_0]$, we know that $\phi(\mathbf{s}')$ is a fixpoint for $T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}$; but this goes against the hypothesis, since $\phi(\mathbf{s})$ is the least fixpoint of $T_{\mathbf{P}\cup\phi(\mathbf{s}_0)}$. This proves the claim. □

B Proof of Lemma 10.8

We need to recall some definitions, concerning nonmonotonic operators and their properties [5].

An operator T is said to be *growing* if, for all I, J, M, $I \subseteq J \subseteq M \subseteq T{\Uparrow}\omega(I)$ implies $T(J) \subseteq T(M)$. Thus, the property of an operator of being growing is a restricted form of monotonicity.

Let $T_1, \ldots, T_n$ be a sequence of operators and I an interpretation; consider the following sequence of interpretations based on I:

$$N_0 = I, \quad N_1 = T_1{\Uparrow}\omega(N_0), \quad \ldots, \quad N_n = T_n{\Uparrow}\omega(N_{n-1}).$$

Clearly, it holds that $N_0 \subseteq N_1 \subseteq \ldots \subseteq N_n$. The sequence of operators $T_1, \ldots, T_n$ is *local* if, for all interpretations I and J, for $1 \leq i \leq n$, $I \subseteq J \subseteq N_n$ implies $T_i(J) = T_i(J \cap N_i)$.

Lemma B.1 *Let* Γ *be a set of* ISALOG$^{\neg}$ *clauses over a scheme* $\mathbf{S}$*. Then, the transformation* T_Γ *is finitary.*

Proof: Similar to the proof of Lemma 8.3. In order to show that, for every infinite sequence of interpretations $I_0 \subseteq I_1 \subseteq \ldots$, it is the case that $T_\Gamma(\cup_{n=0}^{\infty} I_n) \subseteq \cup_{n=0}^{\infty} T_\Gamma(I_n)$, we consider a fact $\alpha \in T_\Gamma(\cup_{n=0}^{\infty} I_n)$ and show that there is an interpretation I_n in the sequence such that $\alpha \in T_\Gamma(I_n)$, so that $\alpha \in \cup_{n=0}^{\infty} T_\Gamma(I_n)$. If $\alpha \in T_\Gamma(\cup_{n=0}^{\infty} I_n)$, then, by definition of $T_{\Gamma,0}$ and T_{ISA}, there are a clause $\gamma \in \Gamma$ and a substitution θ such that $\cup_{n=0}^{\infty} I_n$ satisfies $\theta(\text{BODY}(\gamma))$ and either α is equal to $\theta(\text{HEAD}(\gamma))$ or α is an isa-fact of $\theta(\text{HEAD}(\gamma))$. Note that, since each negative literal in $\theta(\text{BODY}(\gamma))$ is not in $\cup_{n=0}^{\infty} I_n$, then it does not belong to any of the I_n. But this implies that for some I_n, namely the first one in the sequence containing all positive literals in $\theta(\text{BODY}(\gamma))$, it is the case that I_n satisfies $\theta(\text{BODY}(\gamma))$. So, $\alpha \in T_\Gamma(I_n)$. □

From now on, consider an isa-coherently stratified program $\mathbf{P}$ over a scheme $\mathbf{S}$, an isa-coherent stratification $P_1, \ldots, P_n$ of $\mathbf{P}$, an instance $[\mathbf{s}]$ of $\mathbf{S}$, and the interpretation $\phi(\mathbf{s})$ corresponding to $\mathbf{s}$. The immediate consequence operator T_{P_i} associated with a stratum

P_i is defined as in Section 8, that is, $T_{P_i}(I_{\mathbf{S}}) = T_{\mathrm{ISA}}(T_{P_i,0}(I_{\mathbf{S}}))$, where now $T_{P_i,0}$ takes into account also the satisfaction of negative literals.

Recall that we define the following sequence of Herbrand interpretations over $\mathbf{S}$:

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

Lemma B.2 *The operator T_{P_i} is growing, for $1 \le i \le n$.*

Proof: Let us assume that, for some interpretations I, J, M, it is the case that $I \subseteq J \subseteq M \subseteq T_{P_i}{\Uparrow}\omega(I)$. We need to prove that $T_{P_i}(J) \subseteq T_{P_i}(M)$. Consider a fact $\alpha \in T_{P_i}(J)$. For some clause $\gamma \in P_i$ and substitution θ, we have that J satisfies $\theta(\text{BODY}(\gamma))$, with $\theta(\text{BODY}(\gamma)) = \{\alpha_1, \ldots, \alpha_n\}$. Consider a ground literal α_q in $\theta(\text{BODY}(\gamma))$. We can distinguish two cases. *(i)* If α_q is a positive fact, then also M satisfies α_q. *(ii)* If α_q is a negative fact, say $\neg\alpha'_q$, consider the predicate symbol Q of α'_q; clearly $\alpha'_q \in I$, since $I \subseteq J$ and Q must be a predicate symbol defined in some strata that are lower than P_i. This suffices to show that $\alpha'_q \notin T_{P_i}{\Uparrow}\omega(I)$, that is, $\alpha'_q \notin M$. This implies that $\alpha \in T_{P_i}(M)$. $\Box$

Lemma B.3 *The sequence of operators $T_{P_1}, \ldots, T_{P_n}$ is local.*

Proof: Consider an interpretation I and the sequence

$$N_0 = I, \quad N_1 = T_{P_1}{\Uparrow}\omega(N_0), \quad \ldots, \quad N_n = T_{P_n}{\Uparrow}\omega(N_{n-1}).$$

We prove that $T_{P_i}(J) = T_{P_i}(J \cap N_i)$ for every $i \in \{1, \ldots n\}$.

First we prove that, for every $i \in \{1, \ldots, n\}$, $T_{P_i}(J) \subseteq T_{P_i}(J \cap N_i)$.

Let $\alpha \in T_{P_i}(J)$; this means that, for some clause $\gamma \in P_i$ and substitution θ, it is the case that J satisfies $\theta(\text{BODY}(\gamma))$, with $\theta(\text{BODY}(\gamma)) = \{\alpha_1, \ldots, \alpha_n\}$ and: *(i)* either $\alpha = \theta(\text{HEAD}(\gamma))$; *(ii)* or α is an isa-fact of $\theta(\text{HEAD}(\gamma))$. Consider a ground literal α_q in $\theta(\text{BODY}(\gamma))$. If α_q is a negative fact, say $\neg\alpha'_q$, then $\alpha'_q \notin J$, and therefore $\alpha'_q \notin J \cap N_i$. If α_q is a positive fact, whose predicate symbol is Q, then Q occurs (positively) in the body of γ; we now show, by way of contradiction, that $\alpha_q \in N_i$. Suppose that $\alpha_q \notin N_i$; the hypothesis requires $\alpha_q \in N_n - N_i$. Then, for some stratum P_j (with $j > i$), clause $\gamma' \in P_j$, and substitution θ', J satisfies $\theta'(\text{BODY}(\gamma'))$, with either $\alpha_q = \theta'(\text{HEAD}(\gamma'))$ or α_q is an isa-fact of $\theta'(\text{HEAD}(\gamma'))$, that is, Q is a predicate symbol defined by γ'. This implies that $\gamma \in P_i$ refers to $\gamma' \in P_j$, with $j > i$, against the hypothesis of stratification.

Now we prove the converse containment $T_{P_i}(J \cap N_i) \subseteq T_{P_i}(J)$.

Let $\alpha \in T_{P_i}(J \cap N_i)$, that is, for some clause $\gamma \in P_i$ and substitution θ, it is the case that $J \cap N_i$ satisfies $\theta(\text{BODY}(\gamma))$, with $\theta(\text{BODY}(\gamma)) = \{\alpha_1, \ldots, \alpha_n\}$ and $\alpha = \theta(\text{HEAD}(\gamma))$ or α is an isa-fact of $\theta(\text{HEAD}(\gamma))$. Consider a ground literal α_q in $\theta(\text{BODY}(\gamma))$. If α_q is a positive fact, then $\alpha_q \in J \cap N_i$, and therefore $\alpha_q \in J$. If α_q is a negative fact, say $\neg\alpha'_q$, and its predicate symbol is Q, then Q occurs (negatively) in the body of γ, and $\alpha'_q \notin J \cap N_i$; we now show, by way of contradiction, that $\alpha'_q \notin J$. Suppose that $\alpha'_q \in J$; because $J \subseteq N_n$, it follows $\alpha'_q \in N_n - N_i$. Then, for some stratum P_j (with $j > i$), clause $\gamma' \in P_j$, and substitution θ', J satisfies $\theta'(\text{BODY}(\gamma'))$, with either $\alpha_q = \theta'(\text{HEAD}(\gamma'))$ or α_q is an isa-fact of $\theta'(\text{HEAD}(\gamma'))$, that is, Q is a predicate symbol defined by γ'. This implies that $\gamma \in P_i$ negatively refers to $\gamma' \in P_j$, with $j > i$, against the hypothesis of stratification. $\Box$

By Lemmas B.1 B.2 B.3, we know that, given an isa-coherently stratified program $\mathbf{P}$ over a scheme $\mathbf{S}$, each isa-coherent stratification $P_1, \ldots, P_n$ yields a sequence of operators which is local and such that each operator is finitary and growing. Thus, by known results [5, 6], we know that, given an instance $[\mathbf{s}]$, $M_{\mathbf{P},\phi(\mathbf{s})}$ is a minimal fixpoint of the transformation $T_{\mathbf{P}\cup\phi(\mathbf{s})}$ associated with the program $\mathbf{P}$ over instance $[\mathbf{s}]$.

Moreover [5, 6], the fixpoint is independent on the chosen stratification.