Inheritance in Object Oriented Datalog: A Modular Logic Programming Approach

Foto Afrati
National Technical University of Athens
afrati@softlab.ece.ntua.gr

Isambo Karali
University of Athens
izambo@di.uoa.gr

Theodoros Mitakos
National Technical University of Athens
teo@softlab.ece.ntua.gr

Abstract

Modularity is a special form of logic programming composition that allows incremental development and code reusability. It is natural to be considered as the formal framework to supply semantics to languages of logic programs (such as datalog) when the underlying data model is object oriented. We show here that a) for various interesting forms of inheritance, clean and natural semantics can indeed be defined based on the modular logic programming framework that avoids the undesired complication of $datalog^{neg}$ and b) that for certain forms of inheritance, this semantics is equivalent to the semantics of $datalog^{meth}$ which is provided by transformation to $datalog^{neg}$.

1 Introduction

First order logic has been used as a way to represent knowledge and as a language for expressing operations on relations. The combination of logic and databases has proved to be promising. Deductive databases were the result of this combination and, as stated in [GM92], they contributed to understanding the semantics of databases. At the same time, as stated in the same article, logic programming [VEK76, Kow74] adds reasoning to databases. Datalog is a database query language based on logic programming and consists of functions symbols free Horn clauses. According to [Ull88], it provides the simplest data model and, for non–recursive rules, is equivalent to positive relational algebra. As more expressiveness was required, various extensions were introduced to datalog. Allowing negation in rule bodies made datalog strictly more expressive than relational algebra. However, this extra feature destroyed some nice properties inherited from the Horn clauses such as the uniqueness of minimal model and minimal fixpoint. Although various positive results have been reached by adding negation in logic programming [ABW86, VGRS91] and datalog [AVH95] the introduced semantics are significantly more complex than standard ones for definite logic programs and datalog.

Another area of extensions refers to the data model which underlies the deductive databases. These extensions range from supporting structured objects, by allowing function symbols, to object oriented extensions. The latter is a major area of research as both the deductive and object oriented paradigms can profit from each other. As stated in [Ull91], although the two approaches do not mix well, the introduction of object-oriented features to deductive databases seems more promising than the opposite as it is significantly more difficult to provide object oriented languages with a formal foundation rather than extend deductive databases with objects. There has been various approaches for introducing objects in deductive databases [AK89, KL89, ALUW93]. In most of these approaches, the major issues of object identity, classes and inheritance are treated.

Inheritance is a big advantage of object oriented systems. By declaring a class to be a subclass of another, information of the superclass is automatically inherited to the subclass. This information sharing results to smaller code and code reusability is allowed. In conventional object oriented languages definitions in subclasses override definitions of superclasses. In rule–based languages, such as datalog, inheritance may be a more interesting issue rather than in conventional object oriented languages. The reason is that a definition should not necessarily override another that refer to the same predicate. It may extend it. What is more is that it may extend it in various ways. Inheritance is examined in a variety of works. F-Logic [KL89] framework addresses the issue of behavioral inheritance in deductive object oriented database query languages but fails to give it clear semantics. In [BJ95] an object oriented language loosely related to F-Logic is presented. The language supports non monotonic behavioral inheritance and the semantics are reminiscent of the stable model semantics of logic programs with negation. $datalog^{meth}$ [ALUW93] provides a transformational view of behavioral inheritance and a solution to method resolution. $datalog^{meth}$ programs are transformed to datalog programs with negation ($datalog^{neg}$). $datalog^{neg}$ is used as a standard evaluator as well as a semantics provider. In [JL95] the semantics of behavioral inheritance that are given capture the idea of method withdrawal as a solution to the resolution of method definition conflicts. Bugliesi in [Bug92] approaches the issue of behavioral inheritance through a compositional view of extended Horn clause programs. The programs are composed via operators and semantics are given as a function over Herbrand sets rather than as a simple Herbrand set.

In this paper, we discuss various kinds of inheritance of object oriented datalog. Both extending and overriding forms of inheritance are considered as both forms seem to be useful in different cases. In addition, their applicability and suitability for deductive systems is shown. We focus on a more fine grained form of inheritance for datalog. This form is smoothly introduced in a deductive context and it is a combination of extension and overriding of method definitions. It is applicable in cases where both specialization and default information is required. The main focus of the paper is on the semantics of the various forms of inheritance. Clear and simple semantics are sought as the clarity and declarativeness of semantics is one of the main characteristics of deductive database systems. We prove that, for any form of inheritance, the semantics can be expressed by a unique minimal model. The semantics are given using concepts commonly used and widely understood and exploit modular logic programming semantics. This is a more natural choice as modularity is a feature of object oriented systems.

Modularity in logic programming has been an important subject of research in the last ten years as structuring mechanisms are required by modern advanced application development. As claimed in [BLM94] modularity in logic programming has been achieved by two approaches. In the first, meta–linguistic mechanisms have been introduced, usually considering algebras of logic programs and defining operators such as union, intersection, encapsulation, overriding union etc. to perform program composition. This approach, first introduced in [O'K85], have been followed by various researchers [Bro93, BMPT90, MP88]. In the second approach, the language of Horn clauses has been extended to allow implication goals in the bodies. The pioneer work is presented in [Mil86] and the work in [MP89] is an eminent one. In [Bug92], an attempt was made to combine the two approaches, proposing SelfLog, a framework where both paradigms co–exist.

The paper is organized as follows. In Section 2, some information concerning the background of the framework is presented. In Sections 3 and 4, two rather common forms of inheritance are discussed. In Section 5, in order to treat a more complex inheritance form, we need to introduce the language $datalog^{first}$ which is a semantical variation of SelfLog. Comparison with other approaches of inheritance in knowledge–based systems is made in Section 6. Finally, some conclusions are also included.

2 Preliminaries and Definitions

In this section we present some background concepts and some definitions that are required for the formulation of the concepts presented in the paper.

2.1 Two Composition Operators

We start with some background definitions from modular logic programming and logic program composition. In particular, we define two composition operators, namely union and overriding union. Overriding union allows us to model a hierarchical type of program composition with overriding inheritance.

Union is the simplest composition operator that can be applied on logic programs. It simply stands for the set-theoretic union of their clauses. For example, if $P = \{q \leftarrow p, z\}$ and $Q = \{q \leftarrow t, k\}$ then $P \cup Q = \{q \leftarrow p, z, \quad q \leftarrow t, k\}$. The denotation of the union $P \cup Q$ of two programs P and Q can be expressed by the following equation [BLM94]

$$T_{P \cup Q}(I) = T_P(I) \cup T_Q(I)$$

In [BMPT90] the model theoretic semantics of the union of two logic programs is given by the minimal model of the resulting program which is:

$$M_{P \cup Q} = \cap\{M | M \models P \;\& M \models Q\} = min\{M | M \models P \;\& M \models Q\}$$

The union operator is monotonic.

On the other hand, given two logic programs P and Q, their *overriding union*, $P \lhd Q$, results to a program that contains the union of P and Q but if both P and Q contain a definition for the same predicate, then the definition of P overwrites the one found in Q. Considering the above programs P and Q, then $P \lhd Q = \{q \leftarrow p, z\}$. More formally, let $\delta(P)$ be the set of predicate symbols defined by the program P and let $Pred(A)$ denote the predicate symbol of any given atom A. Then $P \lhd Q$ denotes the program obtained as follows: $P \lhd Q = P \cup \{A \leftarrow G \in Q : Pred(A) \notin \delta(P)\}$.

The semantics of $\lhd$ is given in [Bug92] in the following way. Firstly, the operator $\diamond$ has been introduced which provides a means for modelling the overriding semantics of $\lhd$ in a set-theoretic sense. Let S_1 and S_2 be two subsets of the Herbrand Base that captures P and Q and let π any arbitrary set of predicate symbols. Then, $S_1 \diamond_\pi S_2 = S_1 \cup \{t \in S_2 | Pred(t) \notin \pi\}$. Then, the semantics of $\lhd$ rely on the following function:

$$T_{P \lhd Q}(I) = T_P(I) \diamond_{\delta(P)} T_Q(I)$$

2.2 The SelfLog Language

SelfLog [Bug92] is a modular logic programming language that captures the notion of non-monotonic inheritance, where class hierarchies can be modeled by algebraic composition and are combined with inter-unit communication by message–goals. The basic program block in SelfLog is the unit. Units, first introduced in [MP89], stand for named pieces of program which determine environments. A SelfLog unit is composed of extended Horn clauses. The difference with pure Horn clauses is that in the body of such a clause we may have a goal of the form $u : g$ (*message–goal*). This means that the goal g has to be evaluated in unit u. This kind of communication between units can be viewed as a message–passing operation in object oriented terms. The units of any program P are related through partial order $\prec$. We say that a unit u is a superunit of a unit u' if and only if $u' \prec u$. We call a unit u root unit if and only if there is no other unit u' such that $u \prec u'$. The inheritance relation between units is expressed in terms of the overriding union operator, $\lhd$, considered to be applied to units.

In the following we shall consider a restriction of SelfLog to extended datalog clauses, rather than extended Horn clauses in general. A SelfLog program $S = (U, \prec, R, \mu, \lhd)$ is built over a set of constants D and over a set Π of predicate symbols. U is a set of unit names, R is a finite set of extended rules, μ is the mapping of rules to unit names, $\lhd$ is the overriding algebraic operator that applies to units, and $\prec$ is a partial order over U that defines a tree. The Herbrand base $\mathcal{B}$ for each $u \in U$ is built from D over Π.

For the semantics of a SelfLog program S, tuples of Herbrand interpretations rather than single Herbrand interpretations are used.

Definition 2.1 *A T-interpretation I for a SelfLog program S is defined as the tuple of sets $< I(u) : u \in U >$. Every $I(u)$ is the subset of $\mathcal{B}$ that interprets the associated unit.*
The class of T-interpretations I is denoted as $\Im$.

In the following the notion of truth is defined as set membership.

Definition 2.2 *[Bug92] For any interpretation s and any T-interpretation I, we define the truth of a ground formula F with respect to s and I ($s \models_I F$) as follows:*

$$\begin{aligned} s \models_I G \;&\Leftrightarrow\; G \in s && (1)\\ s \models_I G_1, G_2 \;&\Leftrightarrow\; (s \models_I G_1 \wedge s \models_I G_2) && (2)\\ s \models_I h{:}\text{-}G \;&\Leftrightarrow\; (s \models_I G \Rightarrow s \models_I h) && (3)\\ s \models_I u : G \;&\Leftrightarrow\; I(u) \models_I G && (4)\end{aligned}$$

The set of ground instances of clauses of a unit u is denoted as $|u|$.

Definition 2.3 *Let S be a SelfLog program and let M be a T-interpretation for S. M is a T-model for S iff for every $u \in U$ such that $\mathcal{H}(u) = u \prec u_1 \prec \ldots \prec u_n$, $M(u) \models_M C$ for every clause C of $|u \lhd \ldots \lhd u_n|$.*

Proposition 2.1 *For a SelfLog program, the set of its T-models is nonempty and the intersection of T-models is a T-model.*

Definition 2.4 *The semantics of a SelfLog program is provided by the minimum model which is the intersection of all the T-models of the program.*

The semantics of a SelfLog program is given in a constructive way using a continuous transformation T_S. For $u \in U$ and $I \in \Im$, let $T_{u,I} : \mathcal{P}(\mathcal{B}) \rightarrow \mathcal{P}(\mathcal{B})$ be defined as follows: $T_{u,I}(s) = \{A | A\text{:-}G \in |u \lhd u_1 \lhd \ldots \lhd u_b|$ and $s \models_I G\}$, where $\mathcal{H}(u) = u \prec u_1 \prec \ldots \prec u_b$. Then, $T_S(I) =< T_{u,I}(I(u)) | u \in U >$. The least fixpoint of T_S coincides with the minimum model of S. That is $M_S = lfp(T_S) = \bigcup_{k=1}^{\infty} T_S^k(I_\perp)$.

2.3 The Object Oriented Language *datalog*^{meth}

datalog^{meth} [ALUW93] is a deductive object oriented database query language. It is an extension of pure datalog capable of handling object identifiers (oids), classes and methods beyond standard elements and predicates. It is also capable to handle inheritance of methods and resolution of their names. An interesting investigation of various features is made in the context of this language.

In *datalog*^{meth} there are predicate atoms of the form $p(T_1, \ldots, T_n)$ and method atoms of the form $O.m@T_1, \ldots, T_n \rightarrow T$ where T_i is a term, $1 \leq i \leq n$, p is a predicate name and m is a method name, O is a receiver object and T is the result. There are also class atoms of the form $c(O)$, where c is a class name. Rules are expressions of the form $r \leftarrow L_1, \ldots, L_n$ or of the form $r \Leftarrow L_1, \ldots, L_n$. The rules of the first kind are called predicate rules and r is a predicate atom. The rules of the second kind are called method rules and in this case r is a method atom. Method rules are assigned to classes through a function α. Predicate rules apply globally to every class. Classes are configured in a tree like hierarchy. A partial order relation $<$ models the class hierarchy. For any two classes c and c' if $c < c'$ then c is called subclass of c' and c' is called superclass of c. For every class there is at most one superclass.

datalog^{meth} is considered with respect to different forms of inheritance. The semantics of *datalog*^{meth} has been given as a rewriting to *datalog*^{neg}. A different rewriting is performed according to the considered form of inheritance. However, interesting results are given only when positive *datalog*^{meth} programs are considered, i.e. *datalog*^{meth} programs without negation.

2.4 Syntactic Characteristics of our Framework

In the framework that we examine in this paper, we investigate various forms of inheritance for datalog aiming at supplying simple and clear semantics for each of them. In this paragraph we give the common syntax of our framework.

A program in our framework is an expression $\Delta = (A, < , P, M, \alpha, \beta)$, where A is the alphabet and it is a tuple $A = (C, OI, D, V, PS, MS)$. All C, OI, D, V, PS and MS sets are pairwise disjoint. C is a finite set of class names, OI is a finite set of object identifiers and D is a finite set of constants. V is a countably infinite set of variables. PS and MS are finite sets of predicate and method symbols respectively. $<$ is a partial order over C. P is a finite set of predicate rules and M is a finite set of method rules. In their heads, predicate rules have predicates with symbols from PS and method rules have predicates with symbols from MS. α is a mapping from the method rules in M into the set C of classes and β is a mapping of object identifiers to classes, i.e. $\beta : OI \rightarrow C$. The behavior of a class corresponds to the set of locally defined and inherited methods. Predicates define global general purpose predicate relations. Method rules and predicate rules follow the syntax of the ones of *datalog*^{meth} (see Paragraph 2.3).

The example that follows will clarify the syntax.

Example 2.1

CLASS employee
  METHODS
    $O.salary \rightarrow X \Leftarrow$
      $wyears(O, Years)$,
      $X = Years * 20 + 1000$.
  OBJECTS jim
ENDCLASS

CLASS permanent_employee SUPER employee
  METHODS
    $O.vacations \rightarrow X \Leftarrow$
      $wyears(O, Years), Years < 3, X = Years * 10$.
    $O.vacations \rightarrow 30 \Leftarrow$
      $wyears(O, Years), Years \geq 3$.
  OBJECTS tina
ENDCLASS

PREDICATES
  $wyears(O, Years)$:-$employment\_date(O, Year), current\_year(CYear), Years = CYear - Year$.
  $employment\_date(jim, 1996)$.
  $employment\_date(tina, 1994)$.

We refer to this example in the next section providing its semantics.

3 A Simple Form of Inheritance

In this section, a straightforward interpretation of the inheritance relation $<$ will be examined. This is the monotonic inheritance and its features match naturally with deductive languages. In monotonic inheritance, method definitions existing in higher classes hold also in lower classes taking into account the order that the inheritance imposes. That is to say, the behavior of a class is determined by all the method definitions that are applicable in its superclass together with the method definitions that exist in the class itself. For instance, in the Example 2.1, both the *vacations* and *salary* methods are applicable in the class *permanent_employee*.

When different classes in a hierarchy contain definitions of different methods, the applicability of this form of inheritance is evident. However, different classes in a hierarchy may contain different definitions for the same method. Although in conventional object oriented languages only one of these definitions should be considered, this may not be desirable in some cases. Consider, for instance, the following example:

Example 3.1

CLASS citizen
  METHODS
    $O.rights \rightarrow voting$.
ENDCLASS

CLASS prime_minister SUPER citizen
  METHODS
    $O.rights \rightarrow asylum$.
ENDCLASS

The definition of *rights* in the class *prime_minister* is assumed to extend the definition of *rights* that exists in class *citizen* rather than overriding it. That is to say, rules of both definitions are merged in the subclass, the definition in the subclass extending rather than invalidating the definitions existing in higher classes.

This form of inheritance can be modelled using the union operator as this was described in Paragraph 2.1. More precisely, the semantics of this form of inheritance is given by a rewriting of the initial program to datalog taking into account the union operator. The rewriting, $rwrt_m$, is performed in the following way.

¿From a program Δ we can built a datalog program P such that:

- *To every class c_i we associate a program p_i.*
- *Let r be a method–rule in Δ with $\alpha(r) = c_i$ and O the receiver object in the head of r. Then p_i has a rule r' which is obtained from r by rewriting the method–atoms in r and appending the atom $in(O, p_i)$ to the body of r'.*
- *Every method–atom is rewritten to an atom with predicate symbol the method symbol and arguments with one to one correspondance to the ones of the method–atom.*
- *There is an extra program p_G that contains all the predicate rules.*
- *For every object o_i that belongs to class c_j we introduce a fact $in(o_i, p_j)$ that belongs to p_G.*

- *For every* $c_i, c_j \in C$ *if* $c_i < c_j$ *then there is a clause* $in(X, p_j)$:-$in(X, p_i)$ *that belongs to* p_G.
- $P = \{\cup_{c_i \in C}\ rwrt_m(c_i)\} \cup p_G$.

For example, the classes of the Example 2.1 are rewritten in the following way:

$salary(O, X)$:-$wyears(O, Years), X = Years * 20 + 1000, in(O, employee)$.
$vacations(O, X)$:-$wyears(O, Years), Years < 3, X = Years * 10, in(O, permanent\_employee)$.
$vacations(O, X)$:-$wyears(O, Years), Years \geq 3, in(O, permanent\_employee)$.
$wyears(O, Years)$:-$employment\_date(O, Year), current\_year(CYear), Years = CYear - Year$.
$employment\_date(jim, 1996)$.
$employment\_date(tina, 1994)$.
$in(jim, employee)$.
$in(tina, permanent\_employee)$.
$in(O, employee) : -in(O, permanent\_employee)$.

Type–checking goals that are appended to the method–rules ensure that the rule is applied to the correct objects. The additional rules of the form $in(X, p_j)$:-$in(X, p_i)$ implement the class hierarchies. These two features of P capture the class encapsulation and the inheritance of the initial classes.

Considering the above rewriting, the semantics of the initial program Δ is given as the minimal model of the datalog program $P = rwrt_m(\Delta)$.

4 The Overriding Inheritance

Conventional object oriented languages consider an overriding form of inheritance. A method defined in a subclass overrides any definition of the method that may exist in a higher class. That is to say, in a subclass specialization rather than extension of method definitions take place. Methods that are not defined in a subclass are inherited from its superclass. This form of inheritance is useful in various cases. Such an interpretation may also be considered for the inheritance relation $<$ of our framework. For example

Example 4.1

CLASS cars
METHODS
$O.tax \rightarrow X \Leftarrow$
$age(O, Age), X = 100 * Age.$
$O.insurance \rightarrow X \Leftarrow$
$O.tax \rightarrow Tax, X = Tax * 0.5.$
OBJECTS cr
ENDCLASS

CLASS 4_wdrive SUPER cars
METHODS
$O.tax \rightarrow X \Leftarrow$
$age(O, Age), X = 500 * Age.$
OBJECTS wd
ENDCLASS

CLASS cabrio SUPER cars
METHODS
$O.tax \rightarrow X \Leftarrow$
$age(O, Age), X = 1000 * Age.$
OBJECTS ca
ENDCLASS

PREDICATES
$age(cr, 5)$.
$age(wd, 2)$.
$age(ca, 3)$.

The method *insurance* defined in class *cars* is accessible from all the three classes. The method *tax* is redefined in both the *4_wdrive* and *cabrio* subclasses of the class *cars*. In each subclass only the local definition is visible and the definition of *tax* that exists in the class *cars* is not acessible.

The semantics of this form of inheritance can be expressed by the rewriting of a program to SelfLog considering the operator $\lhd$ with dynamic semantics, as SelfLog assumes. The rewriting, $rwrt_o$, is performed in the following way:

¿From a program Δ *we can build a SelfLog program* S *such that:*

- *To every class* c_i *we associate a unit* u_i.

- *Let r be a method–rule in Δ with $\alpha(r) = c_i$ and O the receiver object in the head of r. Then u_i has a rule r' which is obtained from r by rewriting the method–atoms in r and appending the atom $un(O)$ to the body of r'.*
- *Every method–atom $O.m@T_1, \ldots, T_n \rightarrow Y$ in rule bodies is rewritten as $Un : m(O, T_1, \ldots, T_n, Y)$ and is immediately preceeded by an $in(O, Un)$ goal.*
- *There is a root unit u_b that contains all the predicate rules.*
- *For every object o_i that belongs to class c_j we have a fact $in(o_i, u_j)$ that belongs to u_b.*
- *For every object o_i that belongs to class c_j we have an atom $un(o_i)$ that belongs to unit u_j.*
- *We define $u_i \prec u_j$ if $c_i < c_j$.*

For example, the rewriting of the program of the Example 4.1 is the following:

UNIT root
  $age(cr, 5)$. $age(wd, 2)$. $age(ca, 3)$.
  $in(cr, cars)$. $in(wd, 4\_wdrive)$. $in(ca, cabrio)$.
UNIT cars
  $tax(O, X)$:-$age(O, Age), X = 100 * Age, un(O)$.
  $insurance(O, X)$:-$in(O, Un), Un : tax(O, Tax), X = Tax * 0.5, un(O)$.
  $un(cr)$.
UNIT 4_wdrive
  $tax(O, X)$:-$age(O, Age), X = 500 * Age, un(O)$.
  $un(wd)$.
UNIT cabrio
  $tax(O, X)$:-$age(O, Age), X = 1000 * Age, un(O)$.
  $un(ca)$.
$4\_wdrive \prec cars$, $cabrio \prec cars$, $cars \prec root$

The inheritance relation between classes is represented by the inheritance relation $\prec$ between the corresponding units. Consequently, no need for extra global rules are needed —on the contrary such rules are harmful—. In addition, we have introduced facts of the form $un(o)$ in every unit u. This fact declares that object o belongs to unit u. The introduction of these facts to the body of every method rule ensures that the receiver of every method will be an object. Goals of the form $in(O, Un), Un : m(O, T_1, \ldots, T_n, Y)$ make the environment change, if required, and allow to use the definition of method m at unit Un that corresponds to the class where the object O belongs to.

The semantics of the initial program Δ is given by the semantics of the SelfLog program S that is $S = rwrt_o(\Delta)$. This semantics is supplied by the minimal model of S.

5 A Fine Grained Form of Inheritance for Datalog Systems

In this section we examine another form of inheritance in object oriented datalog systems by first considering it in a simpler datalog context. This kind of inheritance allow specialization of method definitions in subclasses but when a local definition is not applicable, definitions in the class hierarchy are tried until a successful one is reached. In this way, we obtain more expressiveness as statements deducible in the overriding interpretation of inheritance can still be deduced but new information can be deduced as well. It also allows to support a kind of default reasoning as this assumed in artificial intelligence. For example

Example 5.1

CLASS family
  METHODS
    $O.allowance \rightarrow A \Leftarrow$
      $no\_of\_children(O, NoOfChildren),$
      $A = NoOfChildren * 100.$
  OBJECTS smith
ENDCLASS

CLASS big_family SUPER family
  METHODS
    $O.allowance \rightarrow A \Leftarrow$
      $income(O, Income),$
      $Income < 2000,$
      $no\_of\_children(O, NoOfChildren),$
      $A = NoOfChildren * 200.$
  OBJECTS jones, clark
ENDCLASS

PREDICATES
  $no\_of\_children(smith, 1).$ $no\_of\_children(jones, 5).$ $no\_of\_children(clark, 7).$
  $income(jones, 1500).$ $income(clark, 4000).$

The definition of *allowance* in the class *big_family* is applicable to *jones* though for *clark* the definition in the class *family* is sought.

5.1 The Language $datalog^{first}$

Firstly, we consider a language, $datalog^{first}$, which is datalog with units à la SelfLog. Both languages have the same definition for the truth value but the semantics of the hierarchy are different.

Consider units u_1 and u_2 such that $u_1 \prec u_2$ and

```
u1:                    u2:
  p(a,b,c).              p(a,b,z).
  p(a,b,c').             p(a,c,d).
  p(a,z,v).
```

We require `p(a,c,d)` to be inherited in u_1 though `p(a,b,z)` to be overwritten in u_1 by `p(a,b,c)` and `p(a,b,c')`

We make a kind of distinction between arguments of a goal. If the predicate has arity n, the first $n-1$ arguments can be roughly considered as input though the last as output. Even if the last is instantiated, the constant is compared to the one appearing in the result.

Now, we continue to give the definition of a model for $datalog^{first}$ program. In [Bug92], a T-interpretation is a model if each individual interpretation is a model of the overriding union of the unit and all the units (recursively) that appear in its hierarchy. This is not the case in $datalog^{first}$. Ensuring that the clauses of a unit are made true is not sufficient. "External" information should be also taken into account. Overriding union is not the means of composition that suplies all this information. The ordinary union, on the other hand, is too generous as it provides atoms that are not actually visible and usable in a unit. The precise form of "external" information that should be taken into account is given in the following definition. It refers to the deducible atoms, called R(esult)-atoms, that are provided using information by the closest, in the hierarchy, possible units.

Recall that by $\mathcal{H}(u)$ we denote the hierarchy from a unit u upwards, i.e. $u \prec u_1 \prec \ldots \prec u_r$.

Definition 5.1 *Let Atom be a ground atomic formula or a message–goal. Atom is R-atom of a unit u (with R-atom level k in u) iff*

1. if *Atom* is an ordinary atom:

either $\exists$ *a fact: Atom:-* $\in |u|$. *(In this case $k = 1$.)*

or $\exists$ *a rule: Atom:-Gs* $\in |u|$ *such that*

$\forall g_i \in Gs, 1 \leq i \leq \nu : g_i = p(x_1, \ldots, x_{n-1}, x_n)$ *is a R-atom of* $u_i \in \mathcal{H}(u)$ *(with R-atom level k_i in u_i)*

and $\neg\exists u_j \in \mathcal{H}(u)$ *with* $j < i$ *such that:* $p(x_1, \ldots, x_{n-1}, x'_n)$ *is R-atom of* u_j.

(Then $k = \sum_{i=1}^{\nu} k_i + 1$*)*

2. if $Atom = u' : Atom'$:
   $Atom'$ is R-atom of u' (with R-atom level k' in u', then $k = k'$).

Then, the definition of a model for a $datalog^{first}$ program is:

Definition 5.2 *A T-interpretation I is a model for a $datalog^{first}$ program DU iff $\forall u \in U$, where U is the set of unit names of DU, the following hold:*

1. *$\forall C \in |u|$ it holds: $I(u) \models_I C$*
2. *$F(u) \subseteq I(u)$ where*

$$\begin{aligned} F(u) \quad = \quad & \{Atom | Atom = p(x_1, \ldots, x_{n-1}, x_n) \in I(u_i),\ u_i \in \mathcal{H}(u), u_i \neq u,\ Atom \text{ is R-atom of } u_i \\ & \text{and } \neg\exists Atom' = p(x_1, \ldots, x_{n-1}, x'_n) \in I(u_j) \text{ such that } Atom' \text{ is R-atom of } u_j \text{ and} \\ & u_j \in \mathcal{H}(u) \text{ with } j < i\} \end{aligned}$$

Lemma 5.1 *Let DU be a $datalog^{first}$ program and U the set of unit names of DU. For any model I of DU and any $u \in U$, it holds: if A is an R-atom of u then $A \in I(u)$.*

Sketch: By induction on the R-atom level of A. $\Box$

Lemma 5.2 *The T-interpretation consisting of individual interpretations that are equal to the Herbrand Base is a model for any $datalog^{first}$ program DU. Consequently the set of models of a program DU is nonempty.*

Lemma 5.3 *Let I_1 and I_2 be models for a $datalog^{first}$ program DU and let U be the set of unit names of DU. Then:*

$$I = I_1 \sqcap I_2 = \forall u \in U \ I_1(u) \cap I_2(u)$$

is also a model for DU.

Sketch: The proof distinguishes the two cases in the model definition. For proving that $F(u) \subseteq I(u)$ we use the following observation. If both models contain the same R-atom in the interpretations of the same unit, there is no interpretation of a unit lower in the hierarchy which contains a R-atom conflicting with it and this R-atom is not contained in the corresponding interpretation of the other model. We say that an atom $A = p(x_1, \ldots, x_{n-1}, x_n)$ conflicts with an atom $A' = p(x_1, \ldots, x_{n-1}, x'_n)$ iff $x_n \neq x'_n$. $\Box$

Theorem 5.4 *For any $datalog^{first}$ program DU, there exists a unique minimal model.*

Proof: By Lemma 5.3, for any models I_1 and I_2 of DU, $I_1 \sqcap I_2$ is also a model for DU. By, Lemma 5.2 the set of models of DU is nonempty. Then, the intersections of the corresponding interpretations of all models is a model of DU which is also the minimal model. $\Box$

This unique minimal model provides the semantics of a $datalog^{first}$ program.

Example 5.2 *Let's consider a program DU consisting of the units u_1, u_2, u_3 and u_4 such that $u_1 \prec u_2 \prec u_3 \prec u_4$.*

```
u1:            |     u2:            |     u3:            |     u4:
               |                    |                    |
p(X,Y) :-      |     q(X,Y):-       |     q(X,Y) :-      |
  w(X,Y).      |       v(X,Y),      |       l(X,Y).      |
               |       glb(X,Y).    |                    |
p(X,Y) :-      |                    |                    |
  q(X,Y).      |                    |                    |
               |                    |                    |
p(X,Y) :-      |                    |                    |
  q(X,V),      |                    |                    |
  p(V,Y).      |                    |                    |
               |                    |                    |
w(a,a).        |     v(b,b).        |     l(b,c).        |     glb(b,b).
```

The T-interpretation

$$\begin{aligned} I \;=<\; & I(u_1) = \{glb(b,b), l(b,c), v(b,b), w(a,a), q(b,b), p(b,b), p(a,a)\}, \\ & I(u_2) = \{glb(b,b), l(b,c), v(b,b), q(b,b)\}, \\ & I(u_3) = \{glb(b,b), l(b,c), q(b,c)\}, \\ & I(u_4) = \{glb(b,b)\} \;> \end{aligned}$$

is a model —the minimal model in fact— of DU. Notice that $q(b,c)$ which is R-atom of u_3 is not inherited in u_2 as there is $q(b,b)$ in $I(u_2)$. For the same reason it is not inherited in u_1 as well.

The following theorem relates the concept of R-atoms with the minimal model.

Theorem 5.5 *Let I be a T-interpretation of a datalog^{first} program DU and U be the set of unit names of DU. Then, I is the minimal model for DU $\Leftrightarrow I(u) = F(u) \cup \{Atom | Atom \text{ is R-atom of } u\},\ \forall u \in U$*

Sketch: The proof is divided into two parts: the one for proving that $F(u) \cup R\_atoms(u) \subseteq I(u)$ and the other for proving the opposite. The first is straightforward by the model definition and Lemma 5.1. The second concentrates on the proof of $I(u) - F(u) \subseteq F(u) \cup R\_atoms(u)$ and it is carried out by induction on the derivation depth of atoms $A \in I(u) - F(u)$. □

5.2 Application in Object Oriented Datalog

Now, we shall continue with showing how *datalog^{first}* is used to express the semantics of the fine grained form of inheritance when object identity is significant. In our framework, a class inherits methods that apply only on objects of this class. This means that the "external" information that was mentioned previously consists only of these atoms. In order to be able to express this we should restrict $F(u)$ to contain only these atoms. This restriction does not change the results that we obtained previously. More precisely, the definition of models becomes

Definition 5.3 *A T-interpretation I is a model for a datalog^{first} program DU iff $\forall u \in U$, where U is the set of unit names of DU, the following hold:*

1. *$\forall C \in |u|$ it holds: $I(u) \models_I C$*
2. *$F(u) \subseteq I(u)$ where*

$$\begin{aligned} F(u) \;=\; & \{Atom | Atom = p(x_1, \ldots, x_{n-1}, x_n) \in I(u_i), \\ & u_i \in \mathcal{H}(u),\ u_i \neq u,\ PROP(x_1),\ Atom \text{ is R-atom of } u_i \text{ and} \\ & \neg\exists Atom' = p(x_1, \ldots, x_{n-1}, x'_n) \in I(u_j) \text{ such that } Atom' \text{ is R-atom of } u_j \text{ and} \\ & u_j \in \mathcal{H}(u) \text{ with } j < i\} \end{aligned}$$

where $PROP(x_1)$ determines some constants x_1.

The rewriting, $rwrt_f$ is performed in the following way.
¿From a program Δ we can build a datalog^{first} program DU such that:

- *To every class c_i we associate a unit u_i.*
- *Let r be a method–rule in Δ with $\alpha(r) = c_i$ and O the receiver object in the head of r. Then u_i has a rule r' which is obtained from r by rewriting the method–atoms in r and appending the atom $un(O)$ to the body of r'.*
- *Every method atom $O.m@\ \vec{T} \rightarrow Y$ in the body of a rule is rewritten as $Un : m(O, \vec{T}, Y)$ and is immediately preceded by an $in(O, Un)$ atom.*
- *There is a root unit u_b that contains all the predicate rules.*
- *For every object o_i that belongs to class c_j we have a fact $in(o_i, u_j)$ that belongs to u_b.*
- *For every object o_i that belongs to class c_j we have a fact $un(o_i)$ that belongs to any unit $u_k \in \mathcal{H}(u_j)$.*
- *We define $u_i \prec u_j$ if $c_i < c_j$.*

The rewriting of the program of the Example 5.1 is:

UNIT root

$no\_of\_children(smith, 1)$. $no\_of\_children(jones, 5)$. $no\_of\_children(clark, 7)$.
$income(jones, 1500)$. $income(clark, 4000)$.
$in(smith, family)$. $in(jones, big\_family)$. $in(clark, big\_family)$.

UNIT family

$allowance(O, A)$:-$no\_of\_children(O, NoOfChildren), A = NoOfChildren * 100, un(O)$.
$un(smith)$. $un(jones)$. $un(clark)$.

UNIT big_family

$allowance(O, A)$:-$income(O, Income), Income < 2000, no\_of\_children(O, NoOfChildren),$
$A = NoOfChildren * 200, un(O)$.
$un(jones)$. $un(clark)$.

$big\_family \prec family \prec root$

The above rewriting $rwrt_f$ to $datalog^{first}$ is similar to the one carried out in the rewriting $rwrt_o$. The only difference is that $un(o)$ facts for an object o that belongs to a class c are included to all the units that correspond to the classes of the hierarchy from c upwards. This is done to allow the superclasses to do reasoning for these objects. The class where an object belongs to, i.e. its base class, is stored in the $in/2$ facts in the root unit.

The semantics of the initial program Δ is given by the $datalog^{first}$ program DU that is $DU = rwrt_f(\Delta)$. This semantics is supplied by the minimal model of DU. We require that in the Definition 5.3 of models, $PROP(x_1)$ is satisfied if the base class of x_1 is translated in unit u. That is to say, when we define $F(u)$ for a unit u and a T-interpretation, $PROP(x_1)$ is equivalent to $rwrt_f(\beta(x_1)) = u$.

6 Comparison with Other Approaches

The most widespread form of inheritance both in conventional object oriented systems and rule–based systems is the overriding form. This has been discussed in Section 4 and it is captured by the rewriting in SelfLog.

KEE [KEE93] is a commercial frame–based system with object oriented features. It is used to implement knowledge bases. Every knowledge base is composed of different hierarchies of units. KEE supports both single and multiple inheritance. It also provides various types of behavioral inheritance. In the case of single inheritance, the default type for a method slot is overriding of method implementations. The user, though, can choose the value for a slot to be the union of inherited and local method implementations. These types of inheritance correspond to overriding and monotonic inheritance in our framework.

In the domain of object oriented extensions of datalog, $datalog^{meth}$ [ALUW93] provides an interesting investigation of inheritance. It examines a monotonic form of inheritance as well as non–monotonic ones. The non–monotonic forms of inheritance of $datalog^{meth}$ are two: the so-called static and dynamic ones — note that these terms are used with a different meaning than the ones used by inheritance in modular logic programming [BLM94, Bug92]—. The static form of inheritance corresponds to the overriding one and the dynamic is related to the fine grained form as this is applied to object oriented datalog. The semantics of $datalog^{meth}$ has be given by a rewriting to $datalog^{neg}$. In the case of overriding (static) inheritance, the rewriting is based on rewriting each method–rule r by appending a type–checking goal and the litterals $\neg c_1(O), \ldots, \neg c_k(O)$. The set $\{c_1, \ldots, c_k\}$ is the maximum set of classes where a redefinition of method–rule r exists. In the case of dynamic inheritance, method–rules are duplicated. The one copy provides a method–rule and the second denotes the objects where this rule is applicable. In addition, in the rewritten method–rules, apart from type–checking goals, litterals of the form $\neg appl_{c_1,m}(O, \vec{T}), \ldots, \neg appl_{c_n,m}(O, \vec{T})$ are appended where $\{c_1, \ldots, c_n\}$ is the maximum set of classes where a redefinition of method m exists.

In both cases, the semantics of $datalog^{meth}$ programs are supplied by the semantics of the corresponding $datalog^{neg}$ ones. The semantics of negation, though, differs considerably from the simple semantics of definite Horn clauses. The stratified and well–founded semantics are its simpler approaches. The rewriting of $datalog^{meth}$ is stratified for static inheritance and initial positive $datalog^{meth}$ programs. Similarly, the rewriting of $datalog^{meth}$ is no better than well–founded for dynamic inheritance even for initial positive

$datalog^{meth}$ programs. In the cases of positive programs, though, the semantics of negation may not be necessary.

Indeed, we have rewritten the $datalog^{meth}$ programs with static inheritance in SelfLog and $datalog^{meth}$ programs with dynamic inheritance in $datalog^{first}$. We have proved that the resulting programs are equivalent to the $datalog^{meth}$ ones.

More precisely, for the case of static inheritance the following hold. Let D^M be a $datalog^{meth}$program, Λ be its static rewriting in $datalog^{neg}$, I an interpretation of Λ and T_Λ the immediate consequence operator for stratified $datalog^{neg}$ programs. We define $K_\Lambda^\infty = lfp(T_\Lambda)$.
If D^M is a $datalog^{meth}$ program and S is its rewriting in $SelfLog$, we define $K_S^\infty = \cup_{u \in U} lfp(\mathcal{T}_{u,\emptyset})$.
In addition we define the following sets :
$IN = \{in(o_i, u_j) : o_i$ is an object that belongs to class $c_j\}$, $\mathcal{C} = \{c(o) \in K_\Lambda\}$, $UN = \{un(o_i) : o_i$ is an object that belongs to a class $c\}$, $K'_\Lambda = K_\Lambda - \mathcal{C}$, $K'_S = K_S^\infty - (IN \cup UN)$

Theorem 6.1 *If D^M is a $datalog^{meth}$ program, S is the corresponding SelfLog program and Λ is the corresponding static $datalog^{neg}$ program then S and Λ have equivalent semantics in the sense that $K'_S = K'_\Lambda$.*

Sketch: By induction on the stages of immediate consequence operators for both programs. □

For the case of dynamic inheritance, the equivalence is as follows:

Definition 6.1 *Let D^M be a $datalog^{meth}$ program, and let DU be the dynamic rewritting of D^M in $datalog^{first}$. If M is the minimal model of DU then $M =< M(u) : u \in U >$. We define as $M' = \cup_{u \in U} M(u)$ and $M^{meth} = M' - (UN \cup IN)$, where every ground predicate $m(o, \vec{t}, y)$ is rewritten as $o.m@\vec{t} \to y$, if m is a method name.*

Dynamic resolution selects the minimal class c providing an applicable implementation [ALUW93]. For a method–call $o.m@\vec{t}$ and an interpretation I it is denoted as $c = resolve^{dyn}(o.m@\vec{t}, I)$.

Theorem 6.2 *Let D^M be a $datalog^{meth}$ program, and let DU be the dynamic rewritting of D^M in $datalog^{first}$. Assume that M is the minimal model of DU. Then a method atom $m(o, \vec{t}, s)$ is present in M if and only if there is an applicable method implementation $P_{m,c}$ with respect to M^{meth} and $o.m@\vec{t} \to s$ such that $c = resolve^{dyn}(o.m@\vec{t}, M^{meth})$.*

Sketch: The proof is given following the definitions of R-atoms, applicable method implementations, the definition of M^{meth} and the definition of $resolve^{dyn}$. □

Consequently, we have shown that all the above cases of inheritance can be expressed by a unique minimal model in a clear way.

7 Conclusions

In this paper, we examined the formal semantics of various kinds of inheritance in object oriented datalog. Concepts from modular logic programming and logic program composition are sufficiently powerful to supply these forms of inheritance with simple and clear semantics. Firstly, monotonic inheritance has been examined and its semantics has been defined by a rewriting to datalog exploiting the semantics of the union operator on logic programs. Secondly, the overriding inheritance has been given semantics by a rewriting to the modular logic programming language SelfLog. Finally, a more fine grained form of inheritance for datalog has been investigated proposing the language $datalog^{first}$ which is a semantical variation of SelfLog. This form of inheritance combines both specialization and extension of definitions and can be applied also in cases where default reasoning is required. The inheritance supported by $datalog^{first}$ has been given semantics by a unique minimal model. Object oriented datalog has been rewritten to $datalog^{first}$. The semantics of all the languages: datalog with union, SelfLog and $datalog^{first}$, are given as the unique minimal model defined for each language accordingly.

In addition, the semantic framework has been used to model the kinds of inheritance provided by the object oriented language $datalog^{meth}$. To our knowledge, $datalog^{meth}$ provides one of the most interesting investigations of inheritance in the context of object oriented datalog. We have proved that its semantics can be expressed using the framework that we propose.

Currently, we examine multiple inheritance. We strongly believe that the modular logic programming framework is suitable to model this feature for object oriented datalog systems even in cases of non–conventional forms of multiple inheritance.

References

[ABW86] K. R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. Res Report RC 11681, IBM, April 1986.

[AK89] S. Abiteboul and P. C. Kanellakis. Object identity as a query language primitive. In *SIGMOD*, pages 159–173, 1989.

[ALUW93] S. Abiteboul, G. Lausen, H. Upholl, and E. Waller. Methods and rules. In *SIGMOD*, pages 32–41, 1993.

[AVH95] S. Abiteboul, V. Vianu, and R. Hull. *Foundations of Databases.* Addison Wesley, 1995.

[BJ95] M. Bugliesi and H. M. Jamil. A stable model semantics for behavioural inheritance in deductive object oriented languages. In *International Conference on Database Theory, ICDT'95*, pages 222–237, 1995.

[BLM94] M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *Journal of Logic Programming*, 19/20:443–502, 1994.

[BMPT90] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Composition operators for logic theories. In J. W. Lloyd, editor, *Proceedings of the Computational Logic Symposium*, pages 117–134, November 1990.

[Bro93] A. Brogi. *Program Composition in Computational Logic.* PhD thesis, Università di Pisa, 1993.

[Bug92] M. Bugliesi. A declarative view of inheritance in logic programming. In *Joint International Conference and Symposium on Logic Programming*, pages 112–127, 1992.

[GM92] J. Grant and J. Minker. The impact of logic programming on databases. *Communications of the ACM*, 35:66–81, 1992.

[JL95] H. M. Jamil and V. S. L. Lakshmanan. A declarative semantics for behavioural inheritance and conflict resolution. In *International Symposium on Logic Programming*, pages 130–144, 1995.

[KEE93] *KEE V. 4.1: User Manual*, 1993.

[KL89] M. Kifer and G. Lausen. F-Logic: A higher–order language for reasoning about objects, inheritance, and scheme. In *SIGMOD*, pages 134–146, 1989.

[Kow74] R. Kowalski. Predicate logic as programming language. *Information Processing, IFIP*, pages 569–574, 1974.

[Mil86] D. Miller. A theory of modules for logic programming. In *Proceedings of the 1986 Symposium on Logic Programming*, pages 106–114, 1986.

[MP88] P. Mancarella and D. Pedreschi. An algebra for logic programs. In R. Kowalski and A. Bowen, editors, *5th International Conference in Logic Programming*, pages 1006–1023, 1988.

[MP89] L. Monteiro and A. Porto. Contextual logic programming. In *6th International Conference in Logic Programming*, pages 284–299, 1989.

[O'K85] R. O'Keefe. Towards an algebra for constructing logic programs. In *IEEE Symposium on Logic Programming*, pages 152–160, 1985.

[Ull88] J. D. Ullman. *Principles of Databases and Knowledge Base Systems.* Computer Science Press, 1988.

[Ull91] J. D. Ullman. A comparison between deductive and object oriented database systems. In *Deductive and Object Oriented Databases: International Conference, DOOD'91*, pages 263–277, 1991.

[VEK76] M. Van Emden and A. Kowalski. The semantics of predicate logic as a programming language. *Journal of the ACM*, 23:733–742, 1976.

[VGRS91] A. Van Gelder, K. Ross, and J. S. Schlipf. The well–founded semantics for general logic programs. *Journal of the ACM*, 38:620–650, 1991.