Inheritance in a Deductive Object Database Language with Updates

Elisa Bertino[1], Giovanna Guerrini[2], and Danilo Montesi[1]

[1] Dipartimento di Scienze dell'Informazione
Università di Milano
Via Comelico 39/41, I20135 Milano - Italy
{bertino,montesi}@dsi.unimi.it

[2] Dipartimento di Informatica e Scienze dell'Informazione
Università di Genova
Via Dodecaneso 35, I16146 Genova - Italy
guerrini@disi.unige.it

Abstract. In this paper we introduce inheritance in deductive object databases and define an operator for hierarchically composing deductive objects with state evolution capabilities. Evolution of such objects models the expected transactional behavior while preserving many important features of deductive databases. Deductive objects can be organized in ISA schemas where each object may inherit or redefine the rules defined in other objects. The resulting inheritance mechanism handles both the deductive and the update/transactional issues. Our framework accommodates several types of inheritance such as overriding, extension, and refinement. Besides presenting the language, this paper defines its semantics and provides a description of the interpreter for the language that has been implemented.

1 Introduction

Deductive and object-oriented databases have been the focus of intense research over the last years. The former extend the mathematical foundations of relational databases towards declarative rule databases. The latter provide the modularity and encapsulation mechanisms lacking in relational databases. It is not surprising that the area of deductive object-oriented databases has been influenced, among the others, from researches in the area of databases, logic programming, artificial intelligence, and software engineering.

In our work we take the database point of view where (deductive) objects have the granularity of logical theories and extensional updates are expressed within the rule language to model methods. Cooperation among objects is supported by message passing, extending the Datalog language, as specified in [BGM95]. The aim of this paper is to extend such an approach to accommodate different types of inheritance among objects. Thus we define a language, called Obj^{inh}-Datalog, that, in addition to the above notions, expresses simple inheritance, overriding, extension, and refinement. The resulting language supports two different cooperation mechanisms among objects: message passing and inheritance.

An Obj^{inh}-Datalog database, indeed, consists of a set of objects that, besides cooperating through message exchanges, may also inherit predicate definitions one from another. When an object is defined as a specialization of another object, it must contain all methods of the parent object, but it can change their implementations. For each method it can keep the implementation defined in the parent object, can totally change it, or can slightly modify it, either by extending or by refining it. The language we propose provides those different possibilities on a per-rule rather than on a per-predicate basis, thus achieving a broader set of modeling options and enhancing flexibility.

We remark that our proposal focuses on inheritance mechanisms and on their use to provide a broad spectrum of modeling possibilities and to maximize code reuse. Thus, the language we consider is a very simple deductive object language with updates, from which we leave out all the features that are not relevant to the main stream of our investigation. In particular, the considered language does not support the notion of class, and inheritance relationships are defined among objects, which can thus be seen as prototypes [SLS$^+$94]. These objects can be very useful to design methods and to verify their properties. The proposed approach can however be extended to any deductive object language providing the notion of class and inheritance relationships among classes.

The language has a two step semantics. The first step computes the bindings and collects the updates that will be performed in an *all-or-nothing* style in the second step. The resulting semantics models the traditional query-answer process as well as the transactional behavior. The advantage of this semantics is to allow a smooth integration between the declarative rule language and the updates. Indeed, no control is introduced within rules even if updates are defined in rules.

The paper is structured as follows. The language is presented in Section 2 and its semantics is given in Section 3. Section 4 shows how the language is interpreted in the prototype that has been implemented, whereas Section 5 compares our approach with related work. Finally, Section 6 concludes the work.

2 Language

The language we propose supports two fundamentally different cooperation mechanisms among objects: message passing and inheritance. When an object o sends a message m to an object o' it asks o' to solve the goal m, thus the evaluation context is switched to o'. When, by contrast, an object o inherits a method m from an object o', it simply means that the definition of m in o' is employed, but the context of evaluation is maintained to be the initial method receiver, that is, o. Thus, inheritance can be seen as message passing *without changing self*. Indeed, messages that cannot be answered using the receiver message protocol are forwarded to the parent without changing *self*; when the forwarded message is answered by executing a parent method, every subsequent message sent to *self* will be addressed to the receiver of the initial message. Hence, the context of evaluation is maintained to be the initial message receiver. The following example illustrates the difference between the two cooperation mechanisms.

Example 1. Consider an object obj_1 whose state only consists of the fact $k(a)$ without methods, and an object obj_2 whose state only consists of the fact $k(b)$ and whose only method m is defined by the rule $m(X) \leftarrow k(X)$. Consider first message passing. obj_1 may ask obj_2 to evaluate the goal $?m(X)$ (this can be accomplished by specifying in obj_1 a rule $m(X) \leftarrow obj_2 : m(X)$); the result of evaluating $?m(X)$ in obj_1 would be $X = b$ since the evaluation is performed with respect to obj_2 state. Consider now inheritance: if obj_1 inherits method m from obj_2 (this can be specified by stating $obj_1 \prec obj_2$, since method m is not defined in obj_1) the result of evaluating $?m(X)$ in obj_1 would be $X = a$ since the evaluation is performed with respect to obj_1 state. $\diamond$

In the remainder of this section, we first introduce objects and the message passing mechanism, and then we discuss how objects can be combined through inheritance.

2.1 Objects and Cooperation through Message Passing

Each real-world entity is modeled by an object. Each object has an identifier (object identifier, shortly OID) and a state. The state of an object is represented by a set of attributes, characterizing the properties of the object. The state of the object is encapsulated, that is, can only be modified by invoking operations that are associated with the object. An object communicates with other objects through message exchanges. A message may contain a request to retrieve an object attribute or to modify its state. The use of object identifiers as possible predicate arguments allows the state of an object to contain a reference to another object, and thus to express aggregation (part-of) relationships among objects, in that the value of an object attribute may be the identifier of another object.

In a conventional logic program, all facts and rules appearing in the program can be used in a deduction step. By contrast, in an Obj^{inh}-Datalog database, there exist several sets of rules and facts collected in different objects. Therefore, at each step only the facts and rules of a specific object can be used. As a consequence, a goal must be addressed to a specific object, and the refutation is executed by using only facts and rules belonging to that object, until a rule is found containing a labeled atom in its body. When such a labeled atom is found, the refutation process "moves" to the object specified by the OID labeling this atom.

An object is modeled as a set of facts and rules, where the facts represent the attribute values of an object and the rules represent the methods. Methods are used to compute derived attributes or to perform operations that modify the object state. Rules may contain both action atoms and deduction atoms in their bodies. Action atoms represent the basic mechanism supporting object state evolutions. Moreover, rule bodies may contain (deductive) atoms labeled with

OIDs. The meaning of a labeled atom is to require the refutation of the atom by using the facts and rules of the object, whose OID labels the atom. Therefore, labeled atoms are the basic mechanism supporting message exchanges among objects. The object to which the message is sent can be fixed at program definition time (in which case the label is a (constant) OID) or can vary depending on the value of some object properties (in which case the label is an object-denoting variable).

The notion of object is formalized by the following definitions. We consider a many-sorted signature $\Sigma = \{\Sigma_o, \Sigma_v\}$, only containing constant symbols. Σ_o is the set of object identifiers, that is, the values used to denote objects, while Σ_v is the set of constant value symbols. The sets Σ_o and Σ_v are disjoint. We moreover consider a set of predicate symbols Π partitioned, as in Datalog, in extensional predicate symbols Π^e, and intensional predicate symbols Π^i. Both Π^e and Π^i are families of predicate symbols $\Pi^{e/i} = \{\Pi_w^{e/i}\}_{w \in S^*}$, where S^* denotes the set of all possible strings of sorts, $S = \{o, v\}$ (object identifiers and values). We denote with Π_w the set of predicate symbols $\Pi_w^e \cup \Pi_w^i$. A family of disjoint sets of variable symbols for each sort $V = \{V_o, V_v\}$ is considered. Terms in $Term = \{Term_o, Term_v\}$ are defined as usual for each sort of our language: a term is either a constant or a variable.

Definition 1. *(Deduction Atom). A deduction atom is defined as the application of a predicate symbol to terms of the appropriate sorts, that is, if* $p \in \Pi_w$, $n = length(w)$ *and* $\forall i, i = 1 \ldots n, t_i \in Term_o$ *if* $w.i = o$ *while* $t_i \in Term_v$ *if* $w.i = v$, *then* $p(t_1, \ldots, t_n)$ *is a deduction atom, also denoted as* $p(\tilde{t})$. □

Deduction atoms are partitioned in extensional deduction atoms, those built on predicates in Π^e, and intensional deduction atoms, those built on predicates in Π^i.

Update operations are expressed in our language (as in U-Datalog [MBM97] and in $\mathcal{LDL}$ [NT89]), by action atoms in rule bodies.

Definition 2. *(Action Atom). An action atom is an extensional deduction atom prefixed by* $+$ *(denoting insertion) or* $-$ *(denoting deletion), that is, if* $p(t_1, \ldots, t_n)$ *is an extensional deduction atom, then* $+p(t_1, \ldots, t_n)$ *and* $-p(t_1, \ldots, t_n)$ *are action atoms.* □

Cooperation among objects in the database is supported in our language by labeled atoms. A labeled atom represents a request of evaluating the deduction atom in the object denoted by the label. Two different kinds of labeled atoms are provided. C-labeled atoms model a fixed cooperation among objects, while V-labeled ones model a cooperation depending on the value to which the variable in the label is bound. Thus, let $p \in \Pi_w, a \in \Sigma_v, obj_4 \in \Sigma_o$ and $O \in V_o$, then $obj_4 : p(a)$ is a c-labeled atom (which represents the atom $p(a)$ in the context of the -fixed- object obj_4), whereas $O : p(a)$ is a v-labeled atom. Given a substitution ϑ, assigning a value to O, $O : p(a)$ represents the atom $p(a)$ in the context of the object $O\vartheta$.

Definition 3. *(C-labeled Atom). Let $obj_h \in \Sigma_o$ be an object identifier and $p(t_1, \ldots, t_n)$ be a deduction atom, then $obj_h : p(t_1, \ldots, t_n)$ is a c-labeled atom.* □

Definition 4. *(V-labeled Atom). Let $X \in V_o$ be a variable denoting an object identifier and $p(t_1, \ldots, t_n)$ be a deduction atom, then $X : p(t_1, \ldots, t_n)$ is a v-labeled atom.* □

Having introduced all kinds of atoms that can be used in our language, we are now able to introduce the notion of rule.

Definition 5. *(Rule). A rule has the form*

$$H \leftarrow U, B, B^c, B^v$$

where:

- *H is an intensional deduction atom;*
- *$U = U_1, \ldots, U_i$ is a vector of action atoms, constituting the update part of the rule;*
- *$B = B_1, \ldots, B_w$ is a vector of deduction atoms, constituting the unlabeled part of the condition, that is, of atoms referring the object where the rule is defined;*
- *$B^c = obj_1 : B'_1, \ldots, obj_z : B'_z$ is a vector of c-labeled atoms, that is, of atoms referring specific objects;*
- *$B^v = X_1 : B''_1, \ldots, X_r : B''_r$ is a vector of v-labeled atoms, that is, atoms not referring specific objects;*
- *$X_1, \ldots, X_r$ must appear as arguments of a deduction atom in $B_1, \ldots, B_w$.*

The update part (U) and the condition part (B, B^c, B^v) cannot be both empty. H is referred to as head of the rule, while U, B, B^c, B^v constitute the body of the rule. For a rule to be safe [CGT90] all the variables in H and all the variables in U[1] *must appear in the condition part of the rule (B, B^c, B^v).* □

We remark that the "," symbol in the rule bodies denotes logical conjunction, thus the order of atoms is irrelevant.

Example 2. The following is an example of Obj^{inh}-Datalog rule.

$$k(X, Y) \leftarrow -t(Z), +t(N), t(Z), p(X), obj_1 : r(Y, N), X : k(Y) \qquad \diamond$$

An object obj_j, where $obj_j \in \Sigma_o$ is the object identifier, consists of an object state and a set of methods. The object *state* EDB_j is a set of facts, that is, a set of ground extensional deduction atoms. The object state is a time-varying component, thus in the following we denote with EDB_j^i the possible states of object obj_j, i.e. EDB_j^i denotes the i-th state of object obj_j. *Methods* are expressed by rules.

[1] This ensures that only ground updates are applied to the database.

Definition 6. *(Object). An object $obj_j = \langle EDB_j, IDB_j \rangle$ consists of an identifier obj_j in Σ_o, of an extensional component EDB_j, which is a set of ground extensional deduction atoms, called object state, and an intensional component IDB_j, which is a set of rules as in Definition 5, expressing methods.* □

Referring to Definition 5, we notice that action atoms cannot be labeled. Indeed, to ensure encapsulation, the updates can only refer to the object itself. Note that, as quite usual in the database field, we do not encapsulate object attributes with respect to queries. That is, the value of an object attribute can be queried from outside the object. Otherwise, forcing strict encapsulation, a number of trivial methods only returning attribute values should be written to be used in queries.

2.2 Inheritance

In this section we describe the capabilities of our language for structuring information through specialization. An Obj^{inh}-Datalog database consists of a set of objects that, besides cooperating through labeled atoms, may also inherit predicate definitions from each other. Whenever an object obj_j specializes another object obj_i, the features of obj_i are inherited by obj_j; obj_j may in turn add more features, or *redefine* some of the inherited features. The redefinition of an inherited feature means that obj_j contains a feature with the same name and different definition of a feature in obj_i. The redefinition is a form of conflict between the two objects. In Obj^{inh}-Datalog, according to the object-oriented paradigm, we solve this type of conflict by giving precedence to the most specific information; therefore, the definition of a feature given in an object always takes the precedence over a definition of the same feature given in any of the objects the given object inherits from. This type of approach is called *overriding*. The specialization relationship among objects impacts not only the object structures, but also the behavior specified by the objects. Given objects obj_i and obj_j, such that obj_j inherits from obj_i, obj_j must contain all the methods of obj_i, but it can change their implementations. For each method, obj_j can keep obj_i implementation, can totally change it, or can slightly modify it. Consider a predicate p defined by one or more rules in obj_i, the following *modeling cases* may arise:

1. *simple inheritance*
   obj_j does not define predicate p; therefore, obj_j inherits p from obj_i;
2. *overriding*
   obj_j redefines predicate p, thus overriding the definition of p provided by obj_i;
3. *extension*
   obj_j extends the definition of p provided by obj_i, so that p in obj_j is defined by a set of clauses which is the union among the clauses for p in obj_i and the clauses for p in obj_j;
4. *refinement*
   obj_j refines the definition of p provided by obj_i; p results therefore to be defined in obj_j by a clause whose body is the conjunction of the bodies of clauses for p in obj_i and in obj_j, with the heads properly unified.

Note that object obj_j may provide additional predicates with respect to obj_i ones by defining predicates which are not defined in obj_i.

The above modeling possibilities offer a broad spectrum of reusing modalities to designers. In addition to single inheritance and overriding, that are usual in the object-oriented context, indeed, we support extension and refinement which offer novel and useful opportunities to refine object behavior. Both correspond to the idea of *behavioral subtyping* [LW90], which can be achieved in object-oriented programming languages by exploiting *super* calls or the *inner* mechanism. In particular, extension allows to handle additional cases specific to the inheriting object through the addition of clauses, whereas refinement allows to specialize the behavior by adding some conditions or actions. Note that, in some way, extension can be thought as a sort of contravariant behavior suptyping whereas refinement can be thought as a sort of covariant behavior subtyping[2].

The following example motivates the usefulness of the modeling possibilities above.

Example 3. Consider two objects obj_{person} and $obj_{student}$, such that $obj_{student}$ inherits from obj_{person}. The following are examples corresponding to each of the modeling cases above.

1. *Simple inheritance*: The rule defining a method to evaluate the age of a person, given his birth date, is the same for obj_{person} and $obj_{student}$.
2. *Overriding*: The predicate *young* of obj_{person}, returning True if the person is considered young, most likely will be redefined in $obj_{student}$. Indeed, the criteria for determining when a person is young is probably different from the criteria used for student.
3. *Extension*: Consider a predicate *intelligent* of obj_{person}, returning True if the *IQ* of the person is greater than a given limit. Suppose that $obj_{student}$ contains the score that a student receives on a given test. Moreover, suppose that a student is considered intelligent if either: (i) his *IQ* is greater than the given limit (the same for obj_{person}); or (ii) his score in the test is greater than a given limit. The predicate *intelligent* in $obj_{student}$ then results in being defined by two different rules.
4. *Refinement*: Consider a predicate that assigns a null value to all the facts in an object. The predicate *null* in $obj_{student}$ will likely refine the predicate *null* defined in obj_{person}, since the former should contain update atoms for all facts added in $obj_{student}$. $\diamond$

We remark that our goal is to support the above modeling possibilities on a per-rule rather than on a per-predicate basis, thus achieving a broader set of modeling options. Indeed, an object may retain a clause of a predicate definition from an object it inherits from, yet hiding or refining other clauses of that predicate definition. To support those modeling possibilities, a mechanism is needed

[2] Note that we do not address the issue of covariant-contravariant method (signature) refinement in the paper, since we do not consider typed variables nor signature definitions for our methods.

to refer specific rules in an object. *Labeled rules* are then introduced. A labeled rule has the form

$$l_x : head_a \leftarrow BODY_a$$

where $l_x \in \mathcal{L}$ with $\mathcal{L}$ denumerable set of labels. All labels in a given object must be distinct. $\mathcal{L}(obj_i)$ denotes the set of labels in object obj_i.

The meaning of labeled rules can be explained as follows. Consider objects obj_i and obj_j, such that obj_j inherits from obj_i, and suppose that $l_x \in \mathcal{L}(obj_i)$, $l_y \in \mathcal{L}(obj_j)$; then given the labeled rules

$$\begin{array}{l} l_x : head_a \leftarrow BODY_a \text{ rule of } obj_i \\ l_y : head_b \leftarrow BODY_b \text{ rule of } obj_j \end{array}$$

consider the following cases:

- $l_x = l_y$
  Then, rule $head_b \leftarrow BODY_b$ of obj_j *overrides* rule $head_a \leftarrow BODY_a$ of obj_i; the latter is *hidden* in obj_j. It is not possible to hide a predicate without redefining it. Therefore, if $l_x = l_y$, $head_a = head_b$, that is, the heads of the two clauses must be the same. Thus, a rule defining a predicate p in an object obj may only hide a rule defining the same predicate p in the object from which obj inherits.
- $l_x \neq l_y$
  Then, obj_j inherits rule $head_a \leftarrow BODY_a$ from obj_i. Therefore, both the rule $head_a \leftarrow BODY_a$ and the rule $head_b \leftarrow BODY_b$ can be exploited to evaluate a goal in obj_j.

Therefore, by labeling a rule with the label of a rule of the parent object *overriding* (modeling case 2 above) is realized, whereas by using a different label *extension* (modeling case 3 above) is realized.

Example 4. Given the following rules in object obj_i

$$\begin{array}{l} l_1 : p(X) \leftarrow q(X) \\ l_2 : k(X) \leftarrow r(X) \end{array}$$

if object obj_j inherits from obj_i and its IDB contains the rules

$$\begin{array}{l} l_1 : p(X) \leftarrow r(X) \\ l_3 : k(X) \leftarrow q(X) \end{array}$$

then the rule for predicate p is overridden, whereas the definition of predicate k is extended. The resulting set of rules available in obj_j is

$$\begin{array}{l} l_1 : p(X) \leftarrow r(X) \\ l_2 : k(X) \leftarrow r(X) \\ l_3 : k(X) \leftarrow q(X) \end{array}$$

◇

Moreover, to express *refinement* (modeling case 4 above), a syntactic mechanism is needed that allows to specify that a rule is a refinement of a rule in the parent object. Rule bodies are thus extended to contain a special kind of atom (which we call *inh-atom*) of the form $l_x : super$. Referring to the objects and rules above, if $BODY_b$ contains the inh-atom $l_x : super$, then obj_j results in containing a single rule of the form

$$(p(\tilde{u}))\vartheta \leftarrow (BODY_a, BODY_b)\vartheta$$

where $\vartheta = mgu(\tilde{t}, \tilde{u})$, where $head_a = p(\tilde{t})$, and $head_b = p(\tilde{u})$[3]. If $head_a$ and $head_b$ cannot be unified, then p is defined in obj_j only by rule $head_b \leftarrow BODY_b$.

Example 5. Referring to the rules of object obj_i of Example 4 above, a refinement of the rule labeled by l_2 can be accomplished by an object obj_k, inheriting from obj_i, by the following rule:

$$l_4 : k(X) \leftarrow q(X), l_2 : super$$

In such a way, the resulting rule in obj_k will be

$$l_4 : k(X) \leftarrow q(X), r(X)$$

which is a refinement of the original rule. $\diamond$

Labeled rules thus allow one to represent all modeling cases previously illustrated. Consider objects obj_i and obj_j, such that obj_j inherits from obj_i, and a predicate p defined in obj_i, and let us show how the different options can be realized.

1. *Simple inheritance*: It is sufficient that obj_j does not contain any clause defining p.
2. *Overriding*: For each rule defining p in obj_i, having label l_x, there must exist a rule defining p in obj_j whose label is equal to l_x.
3. *Extension*: All rules defining p in obj_j must have labels different from all the labels associated with the rules defining p in obj_i.
4. *Refinement*: In obj_j all rules defining p must contain the inh-atom $l_x : super$, where l_x is the label associated with the rule defining p in obj_i and whose body must be put in conjunction with the bodies of the rules in obj_j.

Note that through the mechanism above we can refine only some of the rules defining a predicate p, or all the rules defining it, depending on the intended behavior we want to associate with the inheriting object. Note that, moreover, when a rule labeled by l_x is refined in an inheriting object rule through an inh-atom $l_x : super$ in a rule labeled by l_y, it is also inherited by the object. To prevent this inheritance, the object must contain another rule labeled by l_x (for instance, it can simply be $l_y = l_x$).

[3] We impose no condition on rule heads for refinement. That is, we do not require that $head_b$ is at least as instantiated as $head_a$. Note indeed that head refinement is not particularly meaningful in a context like ours where function symbols are not supported.

Finally, we remark that semantically meaningful labels can be exploited to make evident which kind of behavior is being specified for a given rule. For instance, labels of rules introducing new predicates may have a *new* prefix, labels of rules extending an inherited predicate may have an *ext* prefix, and so on.

The following definitions formalize the notion of labeled rule.

Definition 7. *(Inh-Atom). An inh-atom has the form l_x : super with $l_x \in \mathcal{L}$.* □

We remark that *super* is a special 0-ary predicate symbol, which cannot appear in any other kind of atoms of our language.

Definition 8. *(Labeled Rule). A labeled rule is a rule as in Definition 5, labeled by a label $l \in \mathcal{L}$; that is, a labeled rule has the form $l : r$ where r is defined as in Definition 5 extended to contain in its body, in addition to deduction, action, and labeled atoms, inh-atoms as in Definition 7.* □

By using labeled rules we are able to specify what is inherited, what is hidden, and what is refined with the clause granularity, instead of the predicate granularity. Thus, a partial overriding of a predicate is allowed.

An Obj^{inh}-Datalog database consists of a set of objects related by an inheritance hierarchy. The intensional component of each object is a set of labeled rules. Extensional facts, by contrast, are not labeled, so that only simple inheritance and overriding are supported on facts.

Definition 9. *(Database). An Obj^{inh}-Datalog database is a pair*
$O\text{-}DB = \langle \{obj_1, obj_2, \ldots, obj_s\}, \prec \rangle$
where:

- *$\{obj_1, obj_2, \ldots, obj_s\}$ is a set of objects according to Definition 6 such that the intensional component IDB_j of each obj_j, $1 \leq j \leq s$, is a set of labeled rules, as in Definition 8, whose labels are all distinct;*
- *$\prec \subseteq \Sigma_o \times \Sigma_o$ is a relation on objects representing the inheritance hierarchy. Since we consider only single inheritance, the inheritance relationship $\prec$ is a tree, that is, if objects obj_i, obj_j, obj_k $(1 \leq i, j, k \leq s)$ exist such that $obj_j \prec obj_i$ and $obj_j \prec obj_k$, then either $obj_i \prec obj_k$ or $obj_k \prec obj_i$.* □

Given $obj_i, obj_j \in \Sigma_o$, $obj_j \prec obj_i$ denotes that object obj_j inherits from object obj_i. Moreover, $\preceq$ denotes the partial order obtained from the non-reflexive relation $\prec$, that is, $obj_j \preceq obj_i$ denotes the relation $obj_j \prec obj_i \vee obj_i = obj_j$.

Note that we restrict ourself to single inheritance, to avoid name conflicts that will introduce unnecessary complications in the definition of the language, without bringing in any relevant issue with respect to the main focus of the paper. The approach can however be extended to multiple inheritance, by adopting one of the existing approaches to handle name conflicts, such as superclass ordering or explicit qualification.

Example 6. $\langle \{obj_1, obj_2, obj_3\}, \prec \rangle$ is an example of Obj^{inh}-Datalog database, with $obj_3 \prec obj_1$ and

$$
\begin{array}{lcl}
EDB_1 & = & q(a) \quad r(b) \quad s(obj_2) \\
IDB_1 & = & l_1 : p(X) \leftarrow -q(X), q(X) \\
 & & l_2 : k(X,Y) \leftarrow s(X), r(Y), X : h(Y) \\
 & & l_3 : t(X) \leftarrow r(X) \\
 & & l_5 : mr(X) \leftarrow -r(X), r(X)
\end{array}
$$

$$EDB_2 = h(b)$$

$$
\begin{array}{lcl}
EDB_3 & = & f(b) \quad q(b) \\
IDB_3 & = & l_1 : p(X) \leftarrow -q(X), f(X) \\
 & & l_4 : t(X) \leftarrow f(X) \\
 & & l_6 : mf(X) \leftarrow -f(X), f(X) \\
 & & l_2 : k(X,Y) \leftarrow l_2 : super, f(Y)
\end{array}
$$

Referring to the inheritance relationship between obj_1 and obj_3, we point out that predicate p is overridden, predicate k is refined, predicate t is extended, predicate mr is simply inherited, while predicate mf is an additional one. For what concerns the extensional component, obj_3 simply inherits facts $r(b)$ and $s(obj_2)$ from obj_1, whereas it overrides fact $q(a)$ by providing a local definition for predicate q (that is, fact $q(b)$). $\diamond$

Note that our inheritance mechanisms based on rule labeling offer a number of alternatives with respect to redefinition of predicates (e.g. partial overriding, rule addition, rule refinement). Such alternatives could not be supported at the rule level if labeled rules were not provided. Our mechanism obviously requires that rule labels are visible in subclasses, thus requiring that predicate definitions not be encapsulated with respect to subclasses. However, note that some solutions can be devised to the problem of encapsulation. Our language could easily be extended to support both labeled and un-labeled rules, with a traditional overriding model on a per-predicate basis for some predicates. In such a way, when defining a class, the user can decide whether to encapsulate a predicate definition with respect to subclasses (by not labeling the rules defining it) or to let its definition be visible to subclasses, in which different choices can be adopted for inherited rules.

We also point out that overriding of extensional facts is supported in our model. However, since extensional facts are not labeled, on the extensional side overriding works on a per-predicate basis. This means that is not possible to inherit a fact on a predicate and to override another fact on the same predicate. We took this decision since requiring fact labeling seems an unnecessary burden for the user, given that the sophisticated mechanisms provided for rules (useful for code reuse) does not seem very useful at the data level.

We finally remark that we do not consider here the issue of dynamically creating and deleting objects from the database, which thus consists of a fixed set of cooperating objects. This possibility can obviously be added to the language, but the deletion of objects from which other objects inherit must be handled carefully (as in all prototype-based languages).

2.3 Queries

An important requirement of our language is to model both queries typical of deductive databases, as well as queries typical of object-oriented databases. In deductive databases, a query (goal) has usually the form $?p_1(\tilde{t_1}), \ldots, p_n(\tilde{t_n})$ $(n \geq 1)$ where each $p_i(\tilde{t_i})$ $(1 \leq i \leq n)$ is a deductive atom. The meaning of such query is to find all substitutions for the variables in the query so that the conjunction of predicates $p_1(\tilde{t_1}), \ldots, p_n(\tilde{t_n})$ has the truth value True. On the other hand, in object-oriented databases, queries are usually addressed to a specific object, in form of messages. To support all above querying modalities, two different types of queries are defined:

1. *Conjunction of deduction atoms*: The meaning of this type of query is to find all solutions satisfying the query, independently from the objects where the deduction atoms, appearing in the query, are defined.
2. *Conjunction of object-labeled deduction atoms*: The meaning of this type of query is to find all solutions satisfying the query starting from the objects whose OIDs appear in the query. Note, however, that an object may need to send messages to other objects in order to answer the query.

Note that, whenever a query of the first type is issued, the objects on which the query applies may not be related by inheritance relationships. We refer to Obj^{inh}-Datalog queries as *transactions* to emphasize that, since update methods can be invoked, they do not only return sets of bindings but they can also modify the database state. However, updates are not defined in the query but only in the object methods (expressed by rules).

Definition 10. *(Transaction). A transaction has the form*

$$?\ B, B^c$$

where

- $B = B_1, \ldots, B_w$ *is a vector of deduction atoms, that is, they refer to any object of the object database,*
- $B^c = obj_1 : B'_1, \ldots, obj_z : B'_z$ *is a vector of c-labeled atoms, that is, they refer to specific objects,*

and B *and* B^c *cannot be both empty.* □

Note that no updates are explicitly stated in the transaction because each object uses its own methods (rules) to manipulate the object state.

Example 7. Examples of transactions are $T_1 = obj_3 : k(X, Y), obj_1 : t(Y)$ and $T_2 = p(X)$. ◇

3 Semantics

The semantics of Obj^{inh}-Datalog language is given in two steps. The first step is called *marking phase* and the second one *update phase*. The first step is similar to the query-answer since it computes the bindings for the query and collects the updates. Updates are not executed in this phase. They are executed, if there are not complementary updates, in the second phase altogether, modeling the expected transactional behavior.

3.1 Marking Phase Semantics

In this section we model the behavior of a transaction execution. We formalize the rules for evaluating a call taking into account the different options for behavior inheritance we support.

A transaction may contain two kinds of atoms: labeled and unlabeled ones. The labeled atoms must be refuted in the object whose identifier labels the atom, while for unlabeled atoms a refutation is searched for in any object in the object database.

The behavior of a predicate call in an object depends on the labels of the rules defining the predicate in that object as well as on the inheritance hierarchy. In case of overriding, the notion of *most specific behavior* is applied, that is, each object inherits a predicate from the closest ancestor in the hierarchy that contains a definition for that predicate. By contrast, in the case of extension, an object may inherit the union of the definition of a predicate in all its ancestors in the hierarchy. In case of refinement, finally, the rule obtained by combining the rule in the most specific object with the referred rules in its ancestors is exploited.

The combination of those mechanisms results in the following rule for evaluating a call. Consider a transaction $T = p(\tilde{t})$ to be evaluated in an object obj_i belonging to a database $\langle\{obj_1, \ldots, obj_s\}, \prec\rangle$. The evaluation of T proceeds according to the criteria outlined below:

1. if $p \in \Pi^e$ is an extensional atom:
   (a) if p is locally defined in obj_i, then the definition in obj_i is used;
   (b) the definition in the closest ancestor of obj_i that defines p is used, otherwise;
2. if $p \in \Pi^i$ is an intensional atom:
   (a) if p is locally defined in obj_i, then the definition in obj_i is used; moreover, if p is also defined in an ancestor obj_j of obj_i, and the labels of the rules for p in obj_i and in obj_j are different, then the p definition of obj_j is used as well;
   (b) if p is not locally defined in obj_i, then the definition in the closest ancestor of obj_i that defines p is used;
   (c) if p is locally defined in obj_i (or defined in an ancestor obj_j of obj_i, and not overridden), and its definition is a refinement (that is, is a rule containing inh-atoms), then the refined definition for p is used.

Labeled atoms are evaluated by simply changing the evaluation context to the object denoted by the label. Action atoms are evaluated by simply adding the action to the appropriate update set.

The operational semantics of Obj^{inh}-Datalog is given below. For any database $O\text{-}DB = \langle\{obj_1, \ldots, obj_s\}, \prec\rangle$ and transaction T, we denote by $O\text{-}DB \vdash_{\vartheta,S} T$ the fact that there is a derivation sequence of T in $O\text{-}DB$ with answer ϑ and collecting a tuple of update sets S. We reserve the symbol ϵ to denote the empty (identity) answer, whereas $\vartheta\vartheta'$ denotes the composition of substitutions ϑ and ϑ'. Moreover, let S and S' be s-tuples of update sets, $S \cup S'$ denotes the componentwise union of update sets, that is, $(S \cup S') \downarrow i = S \downarrow i \cup S' \downarrow i$, for all i, $1 \leq i \leq s$. A set of updates $\{u_1, \ldots, u_n\}$ is consistent if it does not contain complementary updates (i.e. $+p(a)$ and $-p(a)$). A tuple of update sets S is consistent if all its component update sets are consistent, that is, if for all i, $1 \leq i \leq s$, in $S \downarrow i$ there are no complementary updates.

The derivation relation is defined by rules of the form

$$\frac{Assumptions}{Conclusion} Conditions$$

asserting the *Conclusion* whenever the *Assumptions* and *Conditions* hold. $O\text{-}DB \vdash_{\vartheta,S} T$ is a finite successful derivation of T in $O\text{-}DB$ that computes ϑ and collects S. A successful derivation is computed as a sequence of derivation steps. Each derivation step is performed according to the rules in Fig. 1.

The index i denotes the current context, that is, the object of the database in which the computation is being carried on. It is not present in the first rule, and in the *Conclusion* of rules 2 and 3, modeling the fact that a query is issued against the whole database, and the selection of the evaluation context depends on the query.

Rule 1 models the semantics of queries, which are conjunctions of two subqueries, in terms of the semantics of the subqueries. Rules 2 and 3 model queries which are deduction atoms and object-labeled deduction atoms, respectively, according to the meaning introduced in Section 2.3. For object-labeled deduction atoms in queries the evaluation context is set to the object labeling the atom (Rule 3), while for unlabeled deduction atoms a refutation is looked for in any object of the database (Rule 2). Rule 4 models the semantics of action atoms (the atom is simply added/removed to the set of updates related to the current object). Rule 5 handles the empty conjunction, that is, an empty rule body. Rules 6 and 7 handle c-labeled atoms and v-labeled atoms, respectively, modeling the change of evaluation context. Rule 8 handles extensional atoms: conditions (a) and (b) are related to the two possibilities for evaluating extensional atoms in presence of inheritance hierarchies: the fact is locally defined in the current object or it is simply inherited from a most specific ancestor of its. Rule 9 handles intensional atoms:

- Condition (a) refers to the case of a predicate which is defined locally to the current object (either a new predicate definition, or overriding) and to the case of predicate extension; it states that all the local rules and each rule

$$\text{(1)}\quad \frac{\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta,S} T_1 \qquad \langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta',S'} T_2\vartheta}{\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta\vartheta',S\cup S'} T_1,T_2}\; CONS$$

$$\text{(2)}\quad \frac{i,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta,S} p(\tilde t)}{\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta,S} p(\tilde t)}\; 1\leq i\leq s$$

$$\text{(3)}\quad \frac{i,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta,S} p(\tilde t)}{\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta,S} obj_i : p(\tilde t)}$$

$$\text{(4)}\quad \frac{}{i,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\epsilon,S} \oplus p(\tilde t)}\; S\downarrow i=\{\oplus p(\tilde t)\},\ S\downarrow k=\emptyset, \forall k=1\ldots s, k\neq i$$

$$\text{(5)}\quad \frac{}{i,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\epsilon,\emptyset} \Box}$$

$$\text{(6)}\quad \frac{j,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta,S} p(\tilde t)}{i,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta,S} obj_j : p(\tilde t)}$$

$$\text{(7)}\quad \frac{i,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta,S} T \qquad j,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta',S'} p(\tilde t)\vartheta}{i,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta\vartheta',S\cup S'} T, X : p(\tilde t)}\; COND_7$$

$$\text{(8)}\quad \frac{}{i,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta,\emptyset} p(\tilde t)}\; p\in\Pi^e$$

if one of the following conditions holds:
(a) $p(\tilde s)\in obj_i$, $mgu(\tilde s,\tilde t)=\vartheta$;
(b) $p(\tilde s)\in obj_j$, $i\neq j$, $mgu(\tilde s,\tilde t)=\vartheta$, $obj_i\preceq obj_j$ and $\forall obj_k$ such that $obj_i\preceq obj_k\prec obj_j$ $\not\exists$ in obj_k an extensional fact which unifies with $p(\tilde t)$.

$$\text{(9)}\quad \frac{i,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\sigma,S} T\vartheta}{i,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\sigma\vartheta,S} p(\tilde t)}\; p\in\Pi^i$$

if one of the following conditions holds:
(a) $l_x : p(\tilde s)\leftarrow T\in obj_j$, $mgu(\tilde s,\tilde t)=\vartheta$, $obj_i\preceq obj_j$, T does not contain inh-atoms, and $\forall obj_k$ such that $obj_i\preceq obj_k\prec obj_j$ and $l_y : p(\tilde u)\leftarrow T'\in obj_k$, $l_x\neq l_y$;
(b) $l_x : p(\tilde s)\leftarrow T\in obj_j$, $i\neq j$, $mgu(\tilde s,\tilde t)=\vartheta$, T does not contain inh-atoms, $obj_i\preceq obj_j$ and $\forall obj_k$ such that $obj_i\preceq obj_k\prec obj_j$ $\not\exists r$ in obj_k whose head unifies with $p(\tilde t)$;
(c) $l_x : p(\tilde s)\leftarrow T'\in obj_j$, $l_y : super\in T'$, $T'\setminus l_y : super = T''$, $l_y : p(\tilde u)\leftarrow T'''\in obj_k$, ($\forall obj_p$ such that $obj_i\preceq obj_p\prec obj_j$ and $l_y : p(\tilde w)\leftarrow T'\in obj_p$, $l_x\neq l_y$), $obj_j\prec obj_k$ and not exists obj_h such that $obj_j\prec obj_h$ and $obj_h\prec obj_k$, and $mgu(\tilde s,\tilde u)=\vartheta^*$, $mgu(\tilde s\vartheta^*,\tilde t)=\vartheta$, $T=T''\vartheta^*,T'''\vartheta^*$.

$$\text{(10)}\quad \frac{i,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta,S} A_1 \qquad i,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta',S'} A_2\vartheta}{i,\langle\{obj_1,\ldots,obj_s\},\prec\rangle \vdash_{\vartheta\vartheta',S\cup S'} A_1,A_2}\; CONS$$

With: $\oplus\in\{+,-\}$, $CONS$ is $S\vartheta\vartheta'\cup S'\vartheta\vartheta' consistent$, $COND_7$ is $obj_j = X\vartheta\wedge CONS$.

Fig. 1. Rules defining the derivation relation

for predicate p in ancestors of object obj_i whose label does not appear in a more specific ancestor of obj_i, or in obj_i itself, can be used in the refutation.
- Condition (b) handles simple inheritance, that is, it considers the case in which the current object does not provide a definition for the predicate in the goal. In this case, the definition for that predicate in the most specific ancestor obj_j of the current object is employed.
- Condition (c) models refinement. It considers a rule which is applicable for refutation (as the ones in the cases above) and solves the inh-atoms in it. This means looking for the rule labeled by l_y (where l_y is the label of the inh-atom l_y : *super*) in the most specific object obj_k from which the object obj_j (containing the rule we are solving) inherits, and then substituting the inh-atom with the body T''' of that rule, properly instantiated.

Finally, Rule 10, which is similar to Rule 1, handles conjunctions in rule bodies.

The operational semantics of an Obj^{inh}-Datalog database *O-DB* is defined as the set of ground atoms, for which an Obj^{inh}-Datalog *proof* exists. These ground atoms are constrained by the set of ground updates their deduction collects, that is, $S\vartheta$. The semantics of an Obj^{inh}-Datalog database consists of atoms of the form $H \leftarrow \bar{U}$, where H is a ground atom (either intensional or extensional) and $\bar{U}$ are updates. The presence of the atom $H \leftarrow \bar{U}$ in the semantics means that H is true and that its evaluation causes the execution of the updates $\bar{U}$.

Definition 11. *(Operational Semantics). The operational semantics of an Obj^{inh}-Datalog database O-DB is defined as the set*

$$\begin{aligned}\mathcal{O}(O\text{-}DB) = \{A \leftarrow \bar{U} \mid\ & O\text{-}DB \vdash_{\vartheta,S} T, A = T\vartheta, \bar{U} = \bar{U}_1 \cup \ldots \cup \bar{U}_s,\\ & \textit{each } \bar{U}_i,\ 1 \le i \le s, \textit{ is the conjunction of } obj_i : u_{i_j}\vartheta,\\ & \textit{for } u_{i_j} \in S \downarrow i\}\end{aligned}$$

□

Example 8. Referring to the Obj^{inh}-Datalog database *O-DB* of Example 6 and to transactions T_1 and T_2 of Example 7, the following holds:

- $O\text{-}DB \vdash_{\vartheta_1,\emptyset} T_1$ with $\vartheta_1 = \{X/obj_2, Y/b\}$;
- $O\text{-}DB \vdash_{\vartheta_2,S_2} T_2$ with $\vartheta_2 = \{X/a\}$, $S_2 = \langle\{-q(X)\}, \emptyset, \emptyset\rangle$
  and
  $O\text{-}DB \vdash_{\vartheta'_2,S'_2} T_2$ with $\vartheta'_2 = \{X/b\}$, $S_2 = \langle\emptyset, \emptyset, \{-q(X)\}\rangle$.

The Obj^{inh}-Datalog proofs for those transactions are shown in Fig. 2 and Fig. 3, respectively. ◇

3.2 Update Phase Semantics

As we have said, in the marking phase updates are collected and their consistency is checked but they are not executed. The most common approach to introduce updates in declarative rules is that updates (very often defined in rules

$$\dfrac{\text{(3)}\;\dfrac{\text{(9c)}\;\dfrac{\text{(7)}\;\dfrac{\text{(8a)}\;\dfrac{}{2, O\text{-}DB \vdash_{\vartheta'_1,\emptyset} h(Y)} \qquad \dfrac{\text{(8b)}\;\dfrac{}{3, O\text{-}DB \vdash_{\vartheta''_1,\emptyset} s(X)} \qquad \dfrac{\text{(8b)}\;\dfrac{}{3, O\text{-}DB \vdash_{\epsilon,\emptyset} r(b)} \qquad \dfrac{}{3, O\text{-}DB \vdash_{\epsilon,\emptyset} f(b)}\;\text{(8a)}}{3, O\text{-}DB \vdash_{\epsilon,\emptyset} r(b), f(b)}\;\text{(10)}}{3, O\text{-}DB \vdash_{\vartheta''_1,\emptyset} s(X), r(b), f(b)}\;\text{(10)}}{3, O\text{-}DB \vdash_{\vartheta_1,\emptyset} s(X), r(Y), X : h(Y), f(Y)}}{3, O\text{-}DB \vdash_{\vartheta_1,\emptyset} k(X,Y)}}{O\text{-}DB \vdash_{\vartheta_1,\emptyset} obj_3 : k(X,Y)} \qquad \dfrac{\dfrac{\dfrac{}{1, O\text{-}DB \vdash_{\epsilon,\emptyset} r(b)}\;\text{(8a)}}{1, O\text{-}DB \vdash_{\epsilon,\emptyset} t(b)}\;\text{(9a)}}{O\text{-}DB \vdash_{\epsilon,\emptyset} obj_1 : t(b)}\;\text{(3)}}{O\text{-}DB \vdash_{\vartheta_1,\emptyset} obj_3 : k(X,Y), obj_1 : t(Y)}\;\text{(1)}$$

$$\vartheta'_1 = \{Y/b\} \qquad \vartheta''_1 = \{X/obj_2\} \qquad \vartheta_1 = \vartheta'_1\vartheta''_1 = \{Y/b, X/obj_2\}$$

Fig. 2. Obj^{inh}-Datalog proof for transaction T_1 of Example 7

$$\cfrac{\cfrac{\cfrac{(4)\ \cfrac{}{1, O\text{-}DB \vdash_{\epsilon,S_2} -q(X)} \qquad \cfrac{}{1, O\text{-}DB \vdash_{\vartheta_2,\emptyset} q(X)}\ (8a)}{1, O\text{-}DB \vdash_{\vartheta_2,S_2} -q(X), q(X)}\ (10)}{1, O\text{-}DB \vdash_{\vartheta_2,S_2} p(X)}\ (9a)}{O\text{-}DB \vdash_{\vartheta_2,S_2} p(X)}\ (2)$$

$$\cfrac{\cfrac{\cfrac{(4)\ \cfrac{}{3, O\text{-}DB \vdash_{\epsilon,S'_2} -q(X)} \qquad \cfrac{}{3, O\text{-}DB \vdash_{\vartheta'_2,\emptyset} f(X)}\ (8a)}{3, O\text{-}DB \vdash_{\vartheta'_2,S'_2} -q(X), f(X)}\ (10)}{3, O\text{-}DB \vdash_{\vartheta'_2,S'_2} p(X)}\ (9a)}{O\text{-}DB \vdash_{\vartheta'_2,S'_2} p(X)}\ (2)$$

$$\vartheta_2 = \{X/a\} \quad S_2 = \langle\{-q(X)\}, \emptyset, \emptyset\rangle \qquad \vartheta'_2 = \{X/b\} \quad S'_2 = \langle\emptyset, \emptyset, \{-q(X)\}\rangle$$

Fig. 3. Obj^{inh}-Datalog proof for transaction T_2 of Example 7

bodies) are executed as soon as they are evaluated [MW87]. Under this assumption the evaluation of a rule is performed in a sequence of states and thus the declarativeness of the query part is lost. Under the marking and update phases, the first phase is declarative and preserve this property for the query part while accommodating update specification that are executed altogether in the update phase. This allows one to express within this semantics the transactional behavior where all or none of the updates must be executed. At logical level, this semantics avoids to undo updates that form a transaction. Indeed, updates collected with the marking phase must be executed in the update phase and it is not possible that some of them will be undone due to the checking in former phase. Let us now see the update phase semantics.

First of all we define the semantics of a query T with respect to an Obj^{inh}-Datalog database $O\text{-}DB$. First we note that database systems use a default set-oriented semantics, that is, the query-answering process computes a set of answers. We denote with $Set(T, O-DB)$ the set of pairs (bindings and updates) computed as answers to the transaction T.

$Set(T, O\text{-}DB) = \{\langle \vartheta, \hat{u}\rangle \mid O\text{-}DB \vdash_{\vartheta, S} T, \hat{u} = S\vartheta\}$

We now define a function which takes a set of ground updates, the current extensional components of the objects constituting the database and returns the new extensional components.

Definition 12. *Let $EDB_1^{i_1}, \ldots, EDB_s^{i_s}$ be the current extensional components of the objects constituting the database and $u_1, \ldots, u_s$ be a s-tuple of consistent sets of ground updates. Then the new databases $EDB_1^{i_1+1}, \ldots, EDB_s^{i_s+1}$ are computed by means of the function $\Delta : \mathcal{EC} \times \mathcal{U} \rightarrow \mathcal{EC}$ as follows:*

$\Delta(\langle EDB_1^{i_1}, \ldots, EDB_s^{i_s}\rangle, \langle u_1, \ldots, u_s\rangle) = \langle EDB_1^{i_1+1}, \ldots, EDB_s^{i_s+1}\rangle$

where each $EDB_j^{i_j+1}$, with $j = 1 \ldots s$, is computed from $EDB_j^{i_j}$ and u_j as

$(EDB_j^{i_j} \setminus \{p(\tilde{t}) \mid -p(\tilde{t}) \in u_j\}) \cup \{p(\tilde{t}') \mid +p(\tilde{t}') \in u_j\}$

where $\mathcal{EC}$ denotes all possible s-tuples of extensional components (i.e. of sets of facts) and $\mathcal{U}$ denotes all possible s-tuples of updates sets. □

The update phase semantics models as observable property of a transaction the *set of answers*, the *object states* and *the result of the transaction* itself. It is called $Oss = \langle Ans, State, Res\rangle$ where Ans is the set of answers, $State$ is an s-uple constituted by the extensional components of objects in the database and Res is the transactional result, that is, either Commit or Abort. The set of possible observables Oss is OSS.

Definition 13. *Let $O\text{-}DB^i$[4] be an Obj^{inh}-Datalog database, with EDB^i the tuple of current object states and $O-IDB$ the tuple of method sets of objects. The semantics of a transaction is denoted by function $\mathcal{S}_{O-IDB}(T) : \mathcal{EC} \rightarrow OSS$.*

$$\mathcal{S}_{O-IDB}(T)(EDB^i) = \begin{cases} Oss^{i+1} & \textit{if OK} \\ \langle \emptyset, EDB^i, Abort\rangle & otherwise(inconsistency) \end{cases}$$

[4] Here we denote with $O\text{-}DB^i$ the Obj^{inh}-Datalog database to emphasize that we consider object states EDB^i at time i.

where $Oss^{i+1} = \langle \{\vartheta_j \mid \langle \vartheta_j, \hat{u}_j \rangle \in Set(T, O\text{-}DB_i)\}, EDB^{i+1}, Commit\rangle$, EDB^{i+1} *is computed by means of* $\Delta(EDB^i, \bar{u})$. *The condition* OK *expresses the fact that all the components of the tuple of sets* $\bar{u} = \bigcup_j \hat{u}_j$ *are consistent, that is, there are no complementary ground updates on the same object.* □

Note that, according to the above definition, in Obj^{inh}-Datalog the abort of a transaction may be caused by a transaction that generates an update set with complementary updates on the same atom in the same object (both the insertion and the deletion of the atom). In this case the resulting object state would depend on the execution order of updates, so we disallow this situation by aborting the transaction. In such a way we ensure that the defined semantics is deterministic.

Example 9. Referring to the object database of Example 6, and to transaction T_2 of Example 7, whose answers have been computed in Example 8

$$\Delta(\langle EDB_1, EDB_2, EDB_3\rangle, \langle \{-q(a)\}, \emptyset, \{-q(b)\}\rangle) = \langle EDB_1', EDB_2, EDB_3'\rangle$$

with

- $EDB_1' = \{r(b), s(obj_2)\}$ and
- $EDB_3' = \{f(b)\}$.

Moreover,

$$\mathcal{S}_{IDB}(EDB, T_2), \langle \{\{X/a\}, \{X/b\}\}, EDB', Commit\rangle. \qquad \diamond$$

Note that the update phase semantics specified in [MBM97] for U-Datalog handles transactions composed from atomic transactions, as the ones we support, through the sequence (";") operator. That semantics can trivially be extended to Obj^{inh}-Datalog, by taking into account that tuples of object states and tuples of update sets must be considered rather than a single database state and a single update set.

4 Obj^{inh}-Datalog Interpreter

A prototype implementation of the Obj^{inh}- Datalog language has been developed at the University of Genova, using KBMS1, a knowledge base management system developed in HP laboratories at Bristol [MCH$^+$90]. The language of KBMS1, kbProlog, is an extension of Prolog with modularization facilities, declarative update operations and persistence support. The implementation of the language has been realized in two steps: (*i*) development of a translator from Obj^{inh}-Datalog to U-Datalog; (*ii*) development of a bottom-up interpreter for U-Datalog. The bottom-up interpreter for U-Datalog handles updates with a non-immediate semantics and provides the transactional behavior. The use of a bottom-up evaluation strategy ensures termination. The choice of implementing Obj^{inh}-Datalog via a translation in U-Datalog is due to the fact that the definition and implementation of Obj^{inh}-Datalog is part of a project which aims at developing an enhanced database language, equipped with an efficient implementation. Several optimization techniques for U-Datalog have been developed [BC96] that will lead to an optimized U-Datalog interpreter and therefore to an optimized Obj^{inh}-Datalog interpreter.

An alternative implementation might realize a "direct" interpreter for Obj^{inh}-Datalog, adapting one of the several evaluation techniques developed for deductive databases to object deductive databases (so taking into account message passing, object state evolution and method inheritance). This is a possible issue for future investigation. Our prototype is based on the following steps: (*i*) an Obj^{inh}-Datalog program OP is translated into an Obj-U-Datalog [BGM95] program OP', that is, inheritance relationships are eliminated and each object is extended so that it explicitly contains its structural and behavioral information (thus, flattening the inheritance hierarchy); (*ii*) the Obj-U-Datalog program OP' is translated into a U-Datalog program UP; (*iii*) each Obj^{inh}-Datalog query OQ is first of all translated in a U-Datalog query UQ, and then executed against the program UP using the U-Datalog interpreter. In what follows, we describe each step.

Step 1: Flattening the inheritance hierarchy This step makes explicit the set of facts and rules available for refutation in each object of the object database. Consider an object obj_j whose direct parent object is object obj_i.

– EDB_j^f is obtained from EDB_j as follows

$$EDB_j^f = EDB_j \cup \{\ p(\tilde{c}) \mid p \in \Pi^e, \tilde{c} \text{ tuple of constants in } \Sigma,\ p(\tilde{c}) \in EDB_i^f, \\ EDB_j \text{ does not contain any fact on predicate } p\ \}$$

– IDB_j^f is obtained from IDB_j as follows
  - IDB_j^f contains all the rules of IDB_j, whose bodies are modified by solving the inh-atoms in rule bodies;
    an inh-atom l_x : *super* is solved by replacing it with the body of the rule labeled by l_x in the parent object obj_i, after having properly unified the rule heads and applied the obtained mgu to the rule body[5];
  - IDB_j^f contains all rules of IDB_i^f whose labels do not appear in $\mathcal{L}(obj_j)$.

The flattening process described above is recursively applied starting from the objects roots of the inheritance hierarchy (that is, the objects obj_j such that $\nexists\ obj_i\ obj_j \prec obj_i$), and visiting the inheritance tree in a top-down style till the leaves of the tree are reached.

At the end of the flattening process, the Obj^{inh}-Datalog rules are transformed in Obj-U-Datalog rules by omitting the rule labels.

[5] If the rule heads cannot be unified, the inh-atom is simply removed from the IDB_j^f rule body.

Step 2: Translation of facts and rules The translation from Obj-U-Datalog to U-Datalog is simple. For each object $obj_i \in O\text{-}DB$, for each predicate p of arity n defined in obj_i we have a corresponding predicate p of arity $n+1$ defined in U-Datalog DB. The argument added to each predicate refers to the object in which the predicate is defined. The extensional component of an object obj_i, i.e. EDB_i, is translated as follows. For each fact in EDB_i, $p(\tilde{a})$, with $\tilde{a}$ tuple of constants, we have a fact $p(obj_i, \tilde{a})$ in DB. The extensional database of the U-Datalog program consists of the union of the translation of the extensional components of each object.

The intensional rules are translated as follows. Consider the rule, defined in object obj_i,

$$\begin{aligned} p(\tilde{X}) \leftarrow {} & B_1(\tilde{Y}_1), \ldots, B_k(\tilde{Y}_k), obj_1 : B_{k+1}(\tilde{Y}_{k+1}), \ldots, obj_n : B_{k+n}(\tilde{Y}_{k+n}), \\ & X_1 : B_{k+n+1}(\tilde{Y}_{k+n+1}), \ldots, X_p : B_{k+n+p}(\tilde{Y}_{k+n+p}). \end{aligned}$$

This rule is translated in the following U-Datalog rule:

$$\begin{aligned} p(obj_i, \tilde{X}) \leftarrow {} & B_1(obj_i, \tilde{Y}_1), \ldots, B_k(obj_i, \tilde{Y}_k), B_{k+1}(obj_1, \tilde{Y}_{k+1}), \ldots, \\ & B_{k+n}(obj_n, \tilde{Y}_{k+n}), B_{k+n+1}(X_1, \tilde{Y}_{k+n+1}), \ldots, B_{k+n+p}(X_p, \tilde{Y}_{k+n+p}). \end{aligned}$$

The intensional database of the U-Datalog program consists of the union of the translations of all the rules of the intensional component of each object.

Step 3: Translation of transactions A transaction is translated in the conjunction of the translation of (eventually labeled) atoms that constitute it. A labeled atom $obj_i : p(\tilde{X})$ is translated in a U-Datalog atom $p(obj_i, \tilde{X})$. An unlabeled atom $p(\tilde{X})$ in a transaction, that -as we have seen- is interpreted as a transaction directed to the whole database, is translated in $p(O, \tilde{X})$, where O is a new variable. Note that in this way we obtain in the solution not only the instances of $p(\tilde{X})$ satisfied by the database, but also the objects in which such instances were found.

Example 10. The U-Datalog program resulting from the translation of the object database of Example 6 is the following.

$$\begin{array}{lcllll} EDB & = & q(obj_1, a) & r(obj_1, b) & s(obj_1, obj_2) & h(obj_2, b) \\ & & f(obj_3, b) & q(obj_3 b) & r(obj_3, b) & s(obj_3, obj_2) \end{array}$$

$$\begin{array}{lcl} IDB & = & p(obj_1, X) \leftarrow -q(obj_1, X), q(obj_1, X) \\ & & k(obj_1, X, Y) \leftarrow s(obj_1, X), r(obj_1, Y), h(X, Y) \\ & & t(obj_1, X) \leftarrow r(obj_1, X) \\ & & mr(obj_1, X) \leftarrow -r(obj_1, X), r(obj_1, X) \\ & & p(obj_3, X) \leftarrow -q(obj_3, X), f(obj_3, X) \\ & & k(obj_3, X, Y) \leftarrow s(obj_3, X), r(obj_3, Y), h(X, Y), f(obj_3, Y) \\ & & t(obj_3, X) \leftarrow q(obj_3, X) \\ & & t(obj_3, X) \leftarrow f(obj_3, X) \\ & & mr(obj_3, X) \leftarrow -r(obj_3, X), r(obj_3, X) \\ & & mf(obj_3, X) \leftarrow -f(obj_3, X), f(obj_3, X) \end{array}$$

Moreover, the transactions of Example 7 are translated as follows:

- $T_1 = k(obj_3, X, Y), t(obj_1, Y)$;
- $T_2 = p(O, X)$. ◇

5 Related Work

Several research proposals attempt to combine object-orientation, databases, and logical languages. There are different orthogonal dimensions along which the approaches to the integration of the deductive and object paradigms may be classified. A survey of those proposals can be found in [BGM95].

Most of the approaches do not consider state evolution of deductive objects. More precisely, the characterization of objects as logic theories, coming from object-oriented extensions of logic programming, does not account for any notion of state. McCabe suggests that the change of state for an instance can be simulated by creating new instances [McC88]. Other proposals simulate state changes by using *assert* and *retract* but this approach lacks any logical foundation. In [CW88] intensional variables are introduced to keep trace of state changes without side effects. In other proposals, multi-headed clauses are used for similar purposes. However, the notion of updating object state does not fit well in object-oriented extensions of logic programming. In addition, also approaches developed in the database field, like e.g. [Fre94,GLR90,LO91], do not consider state evolution. Many of the approaches [AK89,GLR90,CCCR$^+$89], moreover, do not consider the behavioral component of objects, that is, methods. We think that this is an important issue because it overcomes the dichotomy between data and operations of the relational model.

Few proposals moreover, deal with behavioral inheritance and overriding. In addition to [ALUW93,LO91,McC88], these topics have been addressed in [BJ95,DT95,JL95]. All these proposals extend F-logic [KL90] (or F-logic variations) with behavioral inheritance. In F-logic, indeed, only structural inheritance is directly captured. For behavioral inheritance, the non-monotonic aspects introduced by the combination of overriding and dynamic binding are modeled only indirectly by means of an iterated fixpoint construction. Moreover, in F-logic, only ground data expressions, that is, values resulting from the application of a method, and not method implementations, can be inherited along the inheritance hierarchy.

In GuLog [DT95] overriding and conflicts arising from multiple inheritance are investigated, in a model similar to F-logic. In GuLog, however, the schema and instance levels are separated. In ORLog [JL95] overriding and withdrawal of properties are supported. Withdrawal is used to prevent the inheritance of some properties in subclasses. It can thus result in non-monotonic inheritance of signatures. A reasonable use of that mechanism is for preference specification for conflict resolution in case of multiple inheritance. In [BJ95] Bugliesi and Jamil also deal with the behavioral aspects of deductive object languages. Their language, moreover, also allows dynamic subclassing, that is, the definition of inheritance relationships through rules on schemas (which are not allowed in Gu-Log and ORLog). Dynamic subclassing raises non-monotonicity problems and leads to the introduction of a notion of i-stratification to guarantee the existence of a unique stable model. All these proposals, however, despite of their differences, deal with overriding on a per-predicate basis and do not consider any form of state evolution[6].

A finer granularity of rule composition is offered by languages supporting embedded implication [BGM96,Fre92,Mil89]. Embedded implication allows one to realize also some of the other features of our language (such as message passing and conservative inheritance), but does not account for all of them (for instance, overriding). We remark, moreover, that our way of supporting such features is very closely related to the basic modeling notions of the object paradigm. This makes it easier to develop rule sets and to reuse them.

6 Conclusions

We have proposed an approach to express inheritance in deductive object databases. Deductive object databases are based on deductive objects that can change state. Cooperation among objects is defined by inheritance and message passing. Several types of inheritance have been investigated and a formal operational semantics for the language is given. This semantics models objects with the granularity of theory, updates, methods, message passing, and inheritance as well as transactional behavior. Finally, a prototype has been implemented and a sketch of the interpreter for Obj^{inh}-Datalog is provided.

Our main direction of future work concerns the investigation of the applicability of the proposed approach to other deductive object languages with updates (such as Transaction F-Logic) and to other declarative object models that allows to specify dynamic aspects.

Acknowledgments

We wish to thank the anonymous reviewers for their useful comments that helped us a lot in improving both the technical content and the presentation of the paper. The comments and suggestions emerged during the discussion at the workshop were also very useful.

References

[AK89] S. Abiteboul and P. Kanellakis. Object Identity as a Query Language Primitive. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 159–173, 1989.

[6] Actually, state evolution can be accommodated in F-Logic through Transaction logic [BK94], as discussed in [Kif95]. Also in this case, however, predicate inheritance and overriding work on a per-predicate basis.

[ALUW93] S. Abiteboul, G. Lausen, H. Uphoff, and E. Waller. Methods and Rules. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 32–41, 1993.

[BC96] E. Bertino and B. Catania. Static Analysis of Intensional Databases in U-Datalog. In *Proc. of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 202–212, 1996.

[BGM95] E. Bertino, G. Guerrini, and D. Montesi. Towards Deductive Object Databases. *Theory and Practice of Object Systems*, 1(1):19–39, Spring 1995. Special Issue: Selected Papers from ECOOP '94.

[BGM96] M. Baldoni, L. Giordano, and A. Martelli. Translating a Modal Language with Embedded Implication into Horn Clause Logic. In R. Dyckhoff, H. Herre, and P. Schroeder-Heister, editors, *Proc. Fifth Int'l Workshop on Extensions of Logic Programming*, number 1050 in Lecture Notes in Computer Science, pages 19–33, 1996.

[BJ95] M. Bugliesi and H. M. Jamil. A Stable Model Semantics for Behavioral Inheritance in Deductive Object Oriented Languages. In *Proc. Fifth Int'l Conf. on Database Theory*, pages 222–237, 1995.

[BK94] A. Bonner and M. Kifer. An Overview of Transaction Logic. *Theoretical Computer Science*, 133(2):205–265, 1994.

[CCCR$^+$89] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating Object-Oriented Data Modelling with a Rule-Based Programming Paradigm. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 225–236, 1989.

[CGT90] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, 1990.

[CW88] W. Chen and D. S. Warren. Objects as Intensions. In *Proc. Fifth Int'l Conf. on Logic Programming*, pages 404–419. The MIT Press, 1988.

[DT95] G. Dobbie and R. W. Topor. A Model for Sets and Multiple Inheritance in Deductive Object-Oriented Systems. *Journal of Intelligent Information Systems*, 4(2):193–219, 1995.

[Fre92] B. Freitag. Extending Deductive Database Languages by Embedded Implications. In A. Voronkov, editor, *Proc. Int'l Conf. on Logic Programming and Automated Reasoning*, number 642 in Lecture Notes in Computer Science, pages 84–95, 1992.

[Fre94] B. Freitag. Representing Objects as Modules in Deductive Databases. In U. Geske and D. Seipel, editors, *Proc. Second ICLP-Workshop on Deductive Databases and Logic Programming*, pages 41–56, 1994.

[GLR90] S. Greco, N. Leone, and P. Rullo. COMPLEX: An Object-Oriented Logic Programming System. *IEEE Transactions on Knowledge and Data Engineering*, 4(4):344–359, August 1990.

[JL95] H. M. Jamil and L.V.S. Lakshmanan. A Declarative Semantics of Behavioral Inheritance and Conflict Resolution. In *Proc. Int'l Logic Programming Symposium*, pages 130–144, 1995.

[Kif95] M. Kifer. Deductive and Object Data Languages: A Quest for Integration. In *Proc. Fourth Int'l Conf. on Deductive and Object-Oriented Databases*, number 1013 in Lecture Notes in Computer Science, pages 187–212, 1995.

[KL90] M. Kifer and G. Lausen. F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance, and Schema. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 134–146, 1990.

[LO91] Y. Lou and Z. M. Ozsoyoglu. LLO: An Object-Oriented Deductive Language with Methods and Methods Inheritance. In *Proc. of the ACM SIGMOD Int'l Conf. on Management of Data*, pages 198–207, 1991.

[LW90] G. Leavens and W. Weihl. Reasoning about Object-Oriented Programs that use Subtypes. In *Proc. Fifth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications joint with Fourth European Conference on Object-Oriented Programming*, pages 212–223, 1990.

[MBM97] D. Montesi, E. Bertino, and M. Martelli. Transactions and Updates in Deductive Databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(5):784–797, 1997.

[McC88] F.G. McCabe. *Logic and Objects.* PhD thesis, University of London, November 1988.

[MCH+90] J. Manley, A. Cox, K. Harrison, M. Syrett, and D. Wells. KBMS1 A User Manual. Information System Centre Hewlett-Packard Laboratories, March 1990.

[Mil89] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6(1-2):79–108, 1989.

[MW87] S. Manchanda and D. S. Warren. A Logic-based Language for Database Updates. In J. Minker, editor, *Deductive Databases and Logic Programming*, pages 363–394. Morgan-Kaufmann, 1987.

[NT89] S. Naqvi and S. Tsur. *A Logical Language for Data and Knowledge Bases*, volume 2. Computer Science Press, 1989.

[SLS+94] R. Smith, M. Lentczner, W. Smith, A. Taivalsaari, and D. Ungar. Prototype-Based Languages: Object Lessons from Class-Free Programming (Panel). In *Proc. Ninth Int'l Conf. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 102–112, 1994.