Methods and Rules

Serge Abiteboul* Georg Lausen† Heinz Uphoff†‡ Emmanuel Waller*

Abstract

We show how classical datalog semantics can be used directly and very simply to provide semantics to a syntactic extension of datalog with methods, classes, inheritance, overloading and late binding. Several approaches to resolution are considered, implemented in the model, and formally compared. They range from resolution in C^{++} style to original kinds of resolution suggested by the declarative nature of the language. We show connections to view specification and a further extension allowing runtime derivation of the class hierarchy.

1 Introduction

The two actual competing trends in databases are object-oriented and deductive databases. The first lack formal foundations whereas there is (to our knowledge) no real commercial product of the second kind. Deductive object-oriented databases have been proposed as the natural next step to overcome these shortcomings (see for instance the proceedings of the DOOD conferences). We consider an extension of datalog with classes, methods, inheritance and a view mechanism and study *method resolution* for this language. The contribution is in the spirit of [AK89, KL89] a new (we believe important) step towards flexible and formal languages for databases.

A major issue in object-oriented languages is that of method resolution: the same method name may correspond to several implementations (based on the object receiving the method call) and the resolution consists in choosing one such implementation. Consistency of methods has been studied in [AKW90]. We consider here two kinds of resolution: one static in the spirit of the resolution found in procedural languages, say C^{++}; and one more dynamic which we argue is more natural in a deductive context.

In both cases, the semantics is defined by rewriting the programs in datalog with negation . The advantages of this approach are: (i) we can use a standard evaluator of datalog^{neg}; (ii) the translation provides some insight on the essential nature of inheritance and its fundamental nonmonotonicity; and (iii) the various semantics for negation can be used directly. Indeed, for (iii), we argue that the choice of a specific semantics is a somewhat orthogonal issue. (In the paper, we use stratified semantics [ABW88, Prz88] and well-founded [VGRS91].)

We introduce *views* in the language under the form of virtual classes [AB91]. The population of a virtual class is defined intensionally (vs. by extension for real classes). We show that views together with static resolution allow to simulate dynamic resolution. To increase flexibility we consider the case where the hierarchy itself is also defined intensionally as part of the program.

Our approach to inheritance relates to previous work on inheritance in the context of (rule-based) object-oriented languages as follows. Some of these approaches base inheritance on syntactic criteria, either on the class-hierarchy alone [Bre87], on unification of terms denoting objects [AN86, CCCR$^+$90] or on signatures [LO91]. In [BM92], an additional labeling concept of rules affects inheritance. Other approaches consider inheritance as a model-theoretic default mechanism [BL91, LV92, KLW90]. The techniques we study are in some sense between these two directions. We use syntactical criteria to be able to provide rewritings of

*INRIA, 78153 Le Chesnay, France, Serge.Abiteboul@inria.fr, Emmanuel.Waller@inria.fr. Work partially supported by Esprit BRA Project Fide2.

†Fakultät für Mathematik und Informatik, Universität Mannheim, W-6800 Mannheim, Germany, {lausen, uphoff}@pi3.informatik.uni-mannheim.de

‡Work supported by Deutsche Forschungsgemeinschaft, La 598/3-1.

In: Proc. SIGMOD 93

$$
\begin{array}{rcl}
employee(peter). \quad wstudent(paul). & & wstudent(mary). \\
age(peter, 25). \quad age(paul, 28). & & age(mary, 30). \\
employee(X) & \Longleftarrow & wstudent(X). \\
salary(X, Y) & \Longleftarrow & age(X, Z), Y = 20 * Z, employee(X). \\
socins(X, Y) & \Longleftarrow & salary(X, Z), Y = 0.1 * Z, employee(X). \\
socins(X, 50) & \Longleftarrow & wstudent(X).
\end{array}
$$

Figure 1: A datalog program.

the original programs; however the more sophisticated rewritings we introduce capture also semantic aspects such as, for example, the applicability of rules. We apply general evaluation techniques of datalog^{neg}.

The paper is organized as follows. The language is presented in Section 2, static resolution in Section 3 and dynamic in Section 4. In Section 5, we consider virtual classes and in Section 6 the possibility to derive the class hierarchy as part of the program.

2 The datalog^{meth} Language

In this section, after some brief preliminaries on datalog^{neg}, we introduce the syntax of datalog^{meth} which extends datalog^{neg} with methods, classes and inheritance. We also set up the basis for the two semantics considered in the following two sections. At a fundamental level, datalog^{meth} can be seen as a variant of extensions of datalog with data functions proposed in [AG88, AH88].

2.1 Inheritance and datalog^{neg}

We consider countably infinite pairwise disjoint sets of *elements* $\mathcal{D}$, *predicate names* $\mathcal{P}$ and *variables* $\mathcal{V}$. Predicates have a certain arity. A *term* is either an element or a variable.

The *atoms* in datalog^{neg} are expressions of the form $p(t_1, \ldots, t_n)$, where the t_i, $1 \leq i \leq n$, are terms and p is a predicate name. Any atom is also called a *(positive) literal*; a *negative literal* is an expression of the form $\neg A$, where A is an atom. A *rule* is an expression of the form $A \leftarrow L_1, \ldots, L_n$, $n \geq 0$, where the *body* $L_1, \ldots, L_n$ is a sequence of literals and the *head* A is an atom. If $n = 0$, a rule is also called a *fact*. Rules are considered to be $\forall$-quantified. We require that rules are *safe* (cf. [Ull88]). A *program* is a finite set of rules.

Atoms which do not contain variables are called *ground*. The Herbrand Base of a program P is the set of all its ground atoms. Any subset I of the Herbrand Base is called an *interpretation* of P. A *model* of P is an interpretation, which makes all rules in P true. When there are no negative literals in the rule bodies of P, then P has a unique minimal model, which is called its *canonical* model.

When there are negative literals in the bodies of the rules, there may exist more than one minimal model. In such cases, we choose one of these models as the canonical model. We assume in the following that we use stratified semantics when the program is stratified [ABW88, Prz88] and well-founded semantics otherwise [VGRS91]. In the latter case we in addition require totality of the canonical model. (Thus, not all the programs that we will consider have a semantics. But as we shall see this is already the case because of the functionality requirement of methods.)

A first example of a datalog^{neg} program is given in Figure 1.[1] As usual we distinguish variables from elements by using variable names starting with an upper-case letter. The program in the figure describes a situation, where there are three objects of interest: an employee called *peter* and two w(orking-)students *paul* and *mary*. Moreover, it is stated that each working-student is also an employee. We find one rule defining the salary of all employees, and two rules defining the soc(ial) ins(urance) of employees and working-students, respectively. Observe that all working-students have to pay \$50 for their social insurance.

The program in Figure 1 can be considered a first naive attempt to capture inheritance with datalog^{neg}. The rule $employee(X) \Longleftarrow wstudent(X)$ tries to capture inheritance: the classes with extensions working-students and employees are such that the former class is a subclass of the latter. A closer look at the rules defining the *salary*- and *socins*-predicates reveals that, because the predicates $employee(X)$ and $wstudent(X)$ appear in the body of the respective rules, each rule could be considered as applicable only to the corresponding classes. Now observe that since each working-student is also an employee, the *salary*-rule can also be applied to working-students. On the other hand, the same argument for the *socins*-rules has unintended effects. Since for working-students there exists a more specific implementation of the *socins*-predicate, both rules are applicable and this will in general result in a violation of the intended functional dependency, that for

[1] In this and also in later examples, for the purpose of being realistic, we use arithmetic expressions built out of comparison predicates $=, >, <, \geq, \leq$ and operators $+, -, *, /$.

CLASS :	*employee*		
	METHODS:	$X.salary \rightarrow Y \Longleftarrow age(X, Z),\ Y = 20 * Z.$	
		$X.socins \rightarrow Y \Longleftarrow X.salary \rightarrow Z,\ Y = 0.1 * Z.$	
CLASS :	*wstudent*	SUPER *employee*	
	METHODS:	$X.socins \rightarrow 50 \Longleftarrow .$	
OBJECT:	*peter*	IN CLASS *employee*.	
OBJECT:	*paul*	IN CLASS *wstudent*.	
OBJECT:	*mary*	IN CLASS *wstudent*.	
PREDICATE:	*age*	$(peter, 25),\ (paul, 28),\ (mary, 30).$	

Figure 2: A program in datalog^{meth}.

each student the respective social insurance is uniquely defined. As a consequence, an inconsistency occurs.

From this initial discussion, we see that it is possible to capture some aspects of inheritance in datalog. We call such inheritance *monotonic inheritance*. However, we did not manage to capture what is usually called *overriding*, i.e., the fact that a more specific rule should block the inheritance of a less specific rule. A prime goal of the paper is to show how inheritance with overriding can be incorporated into datalog. To this end we study datalog^{meth}, which extends datalog^{neg} with methods, classes and inheritance.

2.2 Examples

The datalog program of Figure 1 is expressed in datalog^{meth} as shown in Figure 2. The language datalog^{meth} explicitly distinguishes between classes and objects. We assume that the relevant objects can be identified by a unique *object-identifier*, *oid* for short. In the example, the oid's are *peter, paul, mary.* For methods, we use a special (standard) syntax to distinguish them from ordinary predicates (see example). Methods and objects are assigned to classes organized in a hierarchy using SUPER-clause. Note that we allow ordinary predicates in addition to methods. We could also have allowed functions in the style of the data functions of [AH88]. To simplify the presentation, we do not.

The semantics of datalog^{meth} will be given by rewriting datalog^{meth} into datalog^{neg}. The benefit of this approach is the ability to apply already existing evaluation techniques to datalog^{meth}. In Figure 3, we give a rewriting of the datalog^{meth} program of Figure 2 in datalog^{neg}.

Observe how we kept track of the assignment of rules to classes: a rule which was assigned to a class c in datalog^{meth} has a rewriting with an additional atom of the form $c(X)$ in its body. In the rewriting, we have to distinguish method predicates with a functionality constraint from ordinary datalog predicates. More precisely, a model has to satisfy the corresponding functional dependencies. For instance, the relation *salary* must satisfy a functional dependency, namely that a given employee has a unique salary.

The main concern in this paper is inheritance of methods with overriding. Observe that the rewriting in Figure 3 behaves as intended: the method *socins* is inherited only for objects not in class *wstudent*. This is in the spirit of McCarthy's *abnormality*-predicate [McC84]. Instances of class *wstudent* are considered "abnormal" with respect to inheritance from class *employee*. Therefore, the negative literal $\neg wstudent(X)$ is added to the body of the corresponding method to block inheritance from class *employee*. In other terms, the implementation of method *socins* assigned to class *wstudent* overwrites the implementation assigned to class *employee*.

2.3 Syntax of datalog^{meth}

In this section, we define precisely the syntax of datalog^{meth}.

In addition to elements $\mathcal{D}$, predicate names $\mathcal{P}$ and variables $\mathcal{V}$ we now consider countably infinite pairwise disjoint sets of *method names* $\mathcal{M}$ and *class names* $\mathcal{C}$, which in addition are pairwise disjoint to $\mathcal{D}$, $\mathcal{P}$ and $\mathcal{V}$. Method names have a certain arity. The set of elements is the union of two disjoint sets: the set of *objects* and the set of the other (normal) elements. Although we will not mention it for the sake of brevity, all the languages that we use are two-sorted.

Terms are as before either variables or elements. Terms may be used to identify objects, i.e., they may appear in the role of *object-identifiers (oids)*. Thus we will also talk about *oid-terms*. This simplistic treatment of oids is sufficient for the purpose of this paper.

Methods and predicates are used to express properties of objects. In contrast to predicates, methods are functions. Each class has an *extension*, which is the set of oids of the objects assigned to that class and an associated *behavior*, which is its set of methods. Classes are structured in a *class-hierarchy*. The class-hierarchy is a binary relation over classes denoted $<$ which is assumed to be a forest. The relation $c_1 < c_2$ denotes that c_1 is subclass of c_2 (resp., c_2 a superclass of c_1).

$$
\begin{array}{rcl}
employee(peter). & & age(peter, 25). \\
wstudent(paul). & & age(paul, 28). \\
wstudent(mary). & & age(mary, 30). \\
employee(X) & \Longleftarrow & wstudent(X). \\
salary(X,Y) & \Longleftarrow & age(X,Z),\ Y = 20 * Z, employee(X). \\
socins(X,Y) & \Longleftarrow & salary(X,Z),\ Y = 0.1 * Z, employee(X), \neg wstudent(X). \\
socins(X, 50) & \Longleftarrow & wstudent(X). \\
fd's : salary : & X \rightarrow Y, & socins : X \rightarrow Y.
\end{array}
$$

Figure 3: A rewriting of a datalog^{meth} program.

From a logical viewpoint, classes are treated as unary predicates.

The various forms of atoms are defined as follows:[2]

method-atom: $O.m@T_1, \ldots, T_n \rightarrow T$.
m is the name of a method, which applies on an object with oid O with arguments $T_1, \ldots, T_n$, gives the result T. The T_i's are terms; O is an oid-term. When appearing in the head of a rule, a method-atom can be considered as a *method-definition*. For methods without arguments the separator '@' is omitted. O is called the receiver object.

predicate-atom: $p(T_1, \ldots, T_n)$.
p is the name of a predicate and the T_i's are terms.

class-atom: $c(O)$.
Here c is the name of a class and O an oid-term.

A sequence of arguments $T_1, \ldots, T_n$ will be abbreviated by $\vec{T}$, respectively $\vec{t}$ in case of ground arguments, in the sequel. We talk about *class-*, *method-* and *predicate-rules*, depending whether a class-, method- or a predicate-atom is in the head of the respective rule. Positive and negative literals are used as usual from these atoms. Class-rules are postponed up to Section 5.

Rules which define methods are attached to classes. This is achieved by a mapping α, which for every method rule r gives the class to which this rule is attached.

A *program* in datalog^{meth} is an expression $\Delta = (C\ , <\ , P\ , \alpha)$, where C is a finite set of class names, $<$ is a forest over[3] C, P is a finite set of rules, and α is a mapping from the method rules in P into the set C of classes. The set of rules attached to a common class c which define a certain method m form the *implementation* of m in that class, abbreviated $P_{m,c}$.

Let a ground atom p, a rule r and an interpretation I be given. We say a rule r is *applicable* with respect to p and I, whenever there exists a (ground) substitution θ of the variables in r such that for the head H of the rule $H\theta = p$, and for all literals L in the rule body: if L is a positive literal, then $L\theta \in I$; otherwise $L = \neg A$ and $A\theta \notin I$. If p is a method atom $o.m@\vec{t} \rightarrow s$, we in addition require the receiver object to be an element of the class $\alpha(r)$, i.e., $\alpha(r)(o)$ is true. We say an implementation $P_{m,c}$ is applicable with respect to a method atom $o.m@\vec{t} \rightarrow s$ and I, if there is a rule in $P_{m,c}$ which is applicable with respect to $o.m@\vec{t} \rightarrow s$ and I.

To conclude this section we impose some restrictions on classes and introduce some further useful notation. First we require that a class-atom is only allowed to appear in a fact, but not in a rule, neither in the head nor in the body. Second we restrict the way classes are populated. In the following, we assume that for each oid o in Δ, there is exactly one ground fact $c(o)$ in Δ. The class c is called the *base class* of an object o, if fact $c(o)$ is in Δ. This is in the spirit of the oid assignment of IQL [AK89].

The following notions to deal with the class-hierarchy will be used. Let $\preceq$ the transitive reflexive closure of $<$. The set of all super- and subclasses of a given class name c is defined as $c^{\uparrow} = \{c' \mid c \preceq c'\}$, and $c^{\downarrow} = \{c' \mid c' \preceq c\}$, respectively. Let $D \subseteq C$ be a set of class names. The subset of minimal (maximal) classes (with respect to $\preceq$) in D is denoted D^{min} (D^{max}).

2.4 Semantics of datalog^{meth} by rewriting

The semantics of a datalog^{meth} program Δ could be defined directly. We prefer to give it by providing a rewriting into a datalog^{neg} program. This allows us to highlight its deep connections with datalog^{neg}.

To this end, any method-atom $O.m@\vec{T} \rightarrow T$ is rewritten to a method-atom (in datalog^{neg}) $m(O, \vec{T}, T)$ which allows to treat m as any ordinary predicate. Observe that oids are moved inside the braces into the first argument position. Predicate- and class-atoms are left unchanged. Classes are now treated as ordinary unary predicates. In analogy to above, a rule is called a *method-rule*, if the predicate-name in its head is a method-name; otherwise it is called a *predicate-rule*.

[2] In the examples, we use syntactic sugaring which should be self-explanatory. For conciseness, we do not formally define this syntactic sugaring.

[3] This assumption excludes multiple inheritance. This is done to simplify the presentation.

In the following, we propose several forms of rewritings of Δ. Let Λ be a rewriting of Δ in datalog^{neg}. We are interested only in "consistent" interpretations of Λ, in which the functional dependencies due to the methods are fulfilled. An interpretation I is *m-functional* if $m(o, \vec{t}, t)$, $m(o, \vec{t}, t') \in I$ implies $t = t'$. Observe that since methods are assigned to classes, m-functionality can only be violated due to some local inconsistency, i.e., this must result from a conflict between two rules defined in the *same* class. This follows because by inheritance we will only apply rules from the same implementation.

The semantics of a program Δ in datalog^{meth} is now defined using two steps. First we provide a rewriting into a datalog^{neg} program Λ. If Λ has a canonical, m-functional model M, then the semantics of Δ is given by the corresponding datalog^{meth} interpretation, denoted with M^{meth}. M^{meth} is the set of method-, predicate-, and class-atoms which can be obtained from M.

2.5 Monotonic Rewriting

Let $\Delta = (C, <, P, \alpha)$ be a program of datalog^{meth}. Let Λ be a datalog^{neg} program, whose set of rules is exactly defined by the following conditions:

1. Let r be a method-rule in Δ with $\alpha(r) = c$ and O the oid-term in the head of r. Then Λ has a rule r' which is obtained from r by rewriting the method-atoms in r and appending the atom $c(O)$ to the body of r'.

2. Let r be a predicate rule in Δ. Then Λ has a rule r' which is obtained from r by rewriting the method-atoms in r.

3. Let c, c' be classes of Δ, where $c' < c$. Then Λ has a rule $c(X) \Longleftarrow c'(X)$.

Λ is called the *monotonic rewriting* of Δ .

For example consider the datalog^{meth} program in Figure 2 and the datalog^{neg} program in Figure 1. It is easy to see that the latter one is the monotonic rewriting of the former. We will refer to the inheritance realized with this particular rewriting as *monotonic inheritance*.

2.6 Resolution

We already mentioned the limitations of monotonic rewriting due to the absence of overriding. The semantics that we consider in the following two sections do allow for overriding. The problem of inheritance is primarily the choice of the "right" implementation of a method when several possible implementations exist. Since implementations are attached to classes, the problem reduces to that of selecting the class where the appropriate implementation can be found, i. e., resolving the overloading. We address this issue next.

Consider a ground method-atom $o.m@\vec{t} \rightarrow t$. We consider the first part of the atom, $o.m@\vec{t}$ as a *method-call*. By the *resolution* of the method call, we mean the task of selecting an implementation of the method being called. In the most general case, the resolution is performed at run-time using all available information, i.e., the resolution is performed by a function *resolve* which takes a method call $o.m@\vec{t}$ as argument and gives a class, $resolve(o.m@\vec{t})$, where the implementation is found. However, it is very common to do resolution at the class-level. More precisely, one then uses a function *resolve* which depends only on the base class c of the receiver o and the name of the method.

Note that for the monotonic rewriting, we did not perform any method resolution. All implementations attached to the superclasses of the receiver were inherited. In the following, we will introduce resolution to bring in more flexibility.

We first discuss a compile-time resolution in the spirit of the control strategy of procedural object-oriented languages we are aware of. We call this inheritance *static inheritance.* We present drawbacks of static inheritance which basically comes from the fact that a rule nonapplicable for a particular object may override an applicable one. This comes from the strict adaptation of design decisions strongly influenced by procedural traditions in a rule context. We propose next as an alternative *dynamic inheritance* which is based on run-time method resolution.

3 Static Inheritance

For static inheritance, we use a resolution which depends only on the base class and the method name. Static inheritance is implemented in our framework by the following resolution:

$$resolve^{static}(m, c) = c',$$

where

$$c' \in \{c'' \in c^{\uparrow} \mid P_{m,c''} \neq \emptyset\}^{min}.$$

When this set is empty, no class provides an implementation. Otherwise, we require that c' is uniquely defined which is always the case for all class-hierarchies considered in this paper.

The function $resolve^{static}$ can be computed at compile time since we know the class hierarchy, and for each method, the classes where there is an implementation of this method.

In the *static* inheritance semantics, method implementations are inherited in subclasses when there is no more specific implementation. If there is a more specific implementation, inheritance from the superclass is blocked by making the body of the rule non-applicable. Let m be a method, c a class name. We define:

$$redef(m, c) = \{c' \in c^{\downarrow} \mid P_{m,c'} \neq \emptyset, c \neq c'\}$$

Let $\Delta = (C, <, P, \alpha)$ be a program of datalog^{meth}. The *static rewriting* Λ of Δ is obtained from its monotonic rewriting by changing the first rewriting-step to:

1. Let r be a method-rule in Δ with $\alpha(r) = c$ and O the oid-term in the head of r. Then Λ has a rule r' which is obtained from r by rewriting the method-atoms in r and appending the atom $c(O)$ to the body of r'. In addition, we append $\neg c_1(O), \ldots, \neg c_k(O)$ to the body of r', where $redef(m, c)^{max} = \{c_1, \ldots, c_k\}$, if $k \geq 1$.

For the datalog^{meth} program in Figure 2 the corresponding static rewriting is shown in Figure 3. The structure of the rewriting is very simple: the rule provided by the class *employee* to compute the social insurance tax is used only for those employees which do not belong to the class of working students. Note that there may be other subclasses of *employee*, for which this method applies.

The following theorem states that the semantics of methods indeed agrees with the static resolution described above.

Theorem 1 Let Δ be a datalog^{meth} program, and let Λ be the static rewriting of Δ. Assume Λ has a canonical model M. Then a method atom $m(o, \vec{t}, s)$ is present in M if and only if some method implementation $P_{m,c'}$ is applicable with respect to M^{meth} and $o.m@\vec{t} \rightarrow s$, where $c' = resolve^{static}(m, c)$ and c is the base class of o. $\Box$

Note that for positive datalog^{meth} programs, i.e., involving no negative literal, the static rewriting is stratified.

Static inheritance is a nice and simple concept closely capturing inheritance as used in procedural languages. However, a closer observation reveals unintended effects. In procedural languages, an implementation of a method is supposed to provide a total function. In contrast, rules provide a more flexible programming style: if our knowledge increases, we reflect this by adding new rules, respectively, modifying rules. Thus according to the standard rule-based programming paradigm, a rule definition of a method in a class is clearly *not* a total function. Therefore, in the context of rules, it is not natural to tie completely resolution to compile-time decisions. When the rules selected in the most specialized definition are not applicable, one would certainly like to try to use a more general definition assuming one applicable exists.

This is illustrated next in an example which introduces "dynamic" inheritance.

Assume we change in our running example the rule to determine *socins* in class *wstudent* such that this rule only holds for students with a salary smaller than 500:

CLASS *wstudent*
METHODS:$X.socins \rightarrow 50 \Longleftarrow X.salary \rightarrow S, S \leq 500.$

Assume we have the following informations given for employees and for students:

employee	salary
peter	8000

wstudent	salary
paul	300
mary	2000

With static inheritance, *mary* does not have a social insurance tax defined. This does not seem to be the intended meaning of this specification – using for *mary* the implementation of *socins* attached to class *employee* seems more appropriate. We will provide rewritings which will achieve this desired effect.

To summarize the difference: In a procedural language, method implementations in subclasses are assumed to be total (although this results in lack of flexibility). In a deductive language, when one implementation is not applicable, it makes sense to try a more general one which plays the role of a default.

4 Dynamic Inheritance

As a consequence of the previous discussion, in a deductive context, method inheritance should be defined with respect to applicability of rules and not only to class membership. The corresponding kind of inheritance is called *dynamic inheritance*. The associated resolution function is denoted $resolve^{dyn}$.

Let a ground method-call $o.m@\vec{t}$ be given. Static semantics was defined using the set of classes where there is an implementation of this method. Now we have to take into account whether an implementation is applicable with respect to the method-call. Let I be an interpretation. We already defined when an implementation $P_{m,c}$ is applicable with respect to a method-*atom*. We now say an implementation is applicable with respect to a *method-call* $o.m@\vec{t}$ (and I), if $P_{m,c}$ is applicable with respect to a method-atom $o.m@\vec{t} \rightarrow s$ (and I) for some $s \in \mathcal{D}$.

Dynamic resolution selects the minimal class providing an applicable implementation. For a method-call $o.m@\vec{t}$ and an interpretation I we define:

$$resolve^{dyn}(o.m@\vec{t}, I) = c$$

where c is the class given by

$$c \in \{c' \in C \mid P_{m,c'} \text{ applicable w.r.t. } o.m@\vec{t}, I\}^{min}.$$

This class c is uniquely defined, in case one such class exists.

We now introduce knowledge about applicability of rules into rewritings. We are doing this by introducing predicates $appl_{c,m}$, indicating that there is an applicable method implementation for method m in class c. The arity of $appl_{c,m}$ equals the arity of m minus 1, because we are not interested in the result position.

$$
\begin{array}{rcl}
employee(X) & \Leftarrow & wstudent(X). \\
salary(X,Y) & \Leftarrow & age(X,A), Y = 20 * A, employee(X). \\
socins(X,Y) & \Leftarrow & salary(X,S), Y = 0.1 * S, employee(X),\ \neg appl_{wstudent,socins}(X). \\
appl_{wstudent,socins}(X),\ socins(X,50) & \Leftarrow & salary(X,S), S \leq 500, wstudent(X).
\end{array}
$$

Figure 4: Example of a dynamic rewriting.

This predicate may be used in other classes, blocking application of rules from superclasses. For the example in Figure 2, the corresponding *dynamic rewriting* is shown in Figure 4. More than one atom in the head of one rule is interpreted as a conjunction. (This can be viewed as an abbreviation for a set of rules, all having the same body, but with a single atom in the head.) In Figure 4 the students for which the predicate $appl_{wstudent,socins}$ holds are exactly those for which the method implementation in *wstudent* is applicable.

We are now ready to formalize dynamic rewriting. Let $\Delta = (C, <, P, \alpha)$ be a program of datalog^{meth}. The *dynamic rewriting* Λ of Δ is obtained from its monotonic rewriting by changing the first rewriting-step to:

1. Let r be a method-rule in Δ with $\alpha(r) = c$ and $m(O, \vec{T}, T)$ the head of r. Then Λ has a rule r' which is obtained from r by:

   (a) rewriting the method-atoms of r,

   (b) appending the atom $c(O)$ to the body and the atom $appl_{c,m}(O, \vec{T})$ to the head,[4]

   (c) appending $\neg appl_{c_1,m}(O, \vec{T}), \ldots, \neg appl_{c_k,m}(O, \vec{T})$ to the body, where $redef(m, c) = \{c_1, \ldots c_k\}$, $k \geq 1$.

This leads to the following

Theorem 2 Let Δ be a datalog^{meth} program, and let Λ be the dynamic rewriting of Δ. Assume Λ has a canonical model M. 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 $o.m@\vec{t} \rightarrow s$ such that $c = resolve^{dyn}(o.m@\vec{t}, M^{meth})$. □

Note that for positive datalog^{meth} programs the dynamic rewriting is well-founded. Indeed, we conjecture that Fitting semantics [Fit85] is even sufficient.

5 Virtual Classes

The notion of virtual classes was proposed in [AB91]. The instances of a virtual class are defined intensionally instead of being explicitly given. Intuitively, a view is therefore just a query returning a set of objects. A similar approach can be found in F-logic [KLW90]. In this section, we introduce virtual classes into datalog^{meth}.

The purpose of a class is to group data, based on properties they share, structure or behavior. This decision is done when designing the general organization of an application, and is very rigid. In many cases, it is desirable to treat as one class all data satisfying a given run-time predicate. Such a class is introduced in the hierarchy at the same time as any other, but its population is selected at run-time. The objects in a virtual class are equipped with the methods associated to this class. We refer to [AB91] for more motivations; Figure 5 illustrates virtual classes by an example. The virtual class *poorstudent* groups some *w(orking)students* based on the value of property *salary* (cf. the INSTANCE-clause). Method *socins* is refined in the virtual class and other methods are introduced.

A datalog^{meth} program $(C, <, P, \alpha)$ is said to provide virtual classes if class names from C are allowed to occur as defined symbols in P. In other words, we now also allow class-rules.

A class c in C is a *virtual class* if c is the defined symbol of a rule r in P. In this case, c is not in the "input", i.e., there is no fact with symbol c in the program. The set of rules with defined symbol c is the *definition* of c. If a class is not virtual, it is called *real*. A class-rule populates a virtual class with objects from other (possibly virtual) classes. There is no restriction on the use of virtual classes within the hierarchy (as long as the hierarchy is a forest).

We can now use the static and dynamic rewriting as above [ALUW93]. Our next result rather surprisingly relates virtual classes and static inheritance to the dynamic inheritance from the previous sections. This demonstrates that the dynamicity achieved using virtual classes is at least as powerful as having dynamic resolution.

Theorem 3 For each datalog^{meth} program under the dynamic inheritance, there is an equivalent datalog^{meth} program (with virtual classes) under static inheritance.

To provide an intuition of the proof, we give an example of the translation of a program with dynamic

[4] In Figure 4 we left out some superfluous *appl*-predicates in the heads to improve readability, as well as class- and predicate-atoms.

$$
\begin{array}{llll}
\text{CLASS}: & wstudent & & \\
 & \text{METHODS}: & X.salary \rightarrow Y \Longleftarrow X.age \rightarrow Z,\ Y = 20 * Z. & \\
 & & X.socins \rightarrow Y \Longleftarrow X.salary \rightarrow Z,\ Y = 0.1 * Z. & \\
\text{CLASS}: & poorstudent & \text{SUPER } wstudent & \\
 & \text{INSTANCES}: & poorstudent(X) \Longleftarrow X.salary \rightarrow S, S \leq 200. & \\
 & \text{METHODS}: & X.socins \rightarrow Y \Longleftarrow X.salary \rightarrow Z,\ Y = 0.01 * Z. & \\
 & & X.registrationfees \rightarrow Y \Longleftarrow X.age \rightarrow Z,\ Y = 0.1 * Z. & \\
 & & X.taxes \rightarrow 0 \Longleftarrow . &
\end{array}
$$

Figure 5: Example of a virtual class.

inheritance into one with virtual classes and static inheritance.

Consider the following program (with dynamic inheritance):

$$
\begin{array}{ll}
c' < c, & \\
X.m \rightarrow Y \Longleftarrow r(X,Y). & \%\ \textit{attached to class } c \\
X.m \rightarrow Y \Longleftarrow s(X,Y). & \%\ \textit{attached to class } c'
\end{array}
$$

The domain of method m at class c' is simulated by a virtual subclass c'_1 of c'. The corresponding program (with static inheritance) is:

$$
\begin{array}{ll}
c'_1 < c' < c, & \\
X.m \rightarrow Y \Longleftarrow r(X,Y). & \%\ \textit{attached to class } c \\
c'_1(X) \Longleftarrow s(X,Y), c'(X). & \%\ \textit{population of class } c'_1 \\
X.m \rightarrow Y \Longleftarrow s(X,Y). & \%\ \textit{attached to class } c'_1
\end{array}
$$

It is now easy to see that the first program with dynamic inheritance and the second program with static inheritance are equivalent.

Note that these methods have no argument. In the general case, virtual classes may involve cartesian products and resolving inheritance involve several arguments. □

6 Deriving also the Hierarchy

We allow more flexibility in the design of applications in the following way. Data organization is now also specified by rules and thus may depend on the properties of the objects involved. Reasoning about the hierarchy becomes part of the program. This issue is discussed in [KLW90] emphasizing semantic aspects; our main concern here are algorithms, i.e., a rewriting in datalog^{neg} in the line of the previous sections.

Consider the example given in Figure 6. We have a base class *items* with subclasses *bigItems* and *cheapItems*. There is a method *price* defined in the classes *items* and *cheapItems*. If the rent for the store is too high, we want to get rid of the big items. In this case, we turn them into cheap items to reduce their price. This is expressed by defining the hierarchy using the program. We add a rule saying that big items are cheap items if the mentioned conditions apply.

We have to extend the definition of datalog^{meth} programs in order to represent the hierarchy in the program. To capture this new view, class symbols are not treated as predicate symbols any more, they are treated as elements in $\mathcal{D}$. (To keep the formalism small we do not introduce an extra sort for such elements.) Thus we are able to define so called *super*-atoms, to make the definition of the class-hierarchy part of the program. In addition, we change the definition of class atoms: whenever we wrote $c(o)$ we now introduce an extra predicate in and write $in(c, o)$. This approach allows us to quantify over class names.

super-atom: $C_1 \prec C_2$.
: C_1, C_2 are terms denoting a sub- and superclass, respectively.

class-atom: $in(O, C)$.
: O is a term denoting an object and C is a term denoting a class name.

A super-atom is only allowed to occur in the head of a rule.

The modification of datalog^{meth} resulting from allowing super-atoms and the modified class atoms is called datalog^{meth,flex}. A datalog^{meth,flex} program is an expression (C, P, α), where C is the set of class symbols, P is a program and α an attachment to class symbols in C as usual. The hierarchy has become part of the program P and doesn't appear any more in the expression. We give semantics to a datalog^{meth,flex} program by presenting a rewriting realizing dynamic resolution. In principle, the hierarchy has to be known when giving the dynamic rewriting for datalog^{meth} programs. Here we have to rewrite a rule in a class without knowledge of the subclasses of this class. The rewriting presented below implements dynamic resolution in this framework.

Consider the rewriting of the datalog^{meth,flex} program in Figure 6, which is given in Figure 7. The hierarchy definition has become part of the program. In addition, there is a rule indicating transitivity of the

$$\begin{array}{lll}
\text{CLASS}: & items & \\
& \text{METHODS}: & X.price \rightarrow Y \Longleftarrow r(X,Y). \\
\text{CLASS}: & bigItem & \text{SUPER } items \\
\text{CLASS}: & cheapItem & \text{SUPER } items \\
& \text{METHODS}: & X.price \rightarrow Y \Longleftarrow s(X,Y). \\
\text{HIERARCHY}: & bigItem \prec cheapItem \Longleftarrow rent(X), X > 2000. &
\end{array}$$

Figure 6: Example program defining the hierarchy

$$\begin{array}{rcl}
bigItem \prec item. & & cheapItem \prec item. \\
bigItem \prec cheapItem & \Longleftarrow & rent(X), X > 2000. \\
X \prec Z & \Longleftarrow & X \prec Y, Y \prec Z. \\
in(O, C') & \Longleftarrow & in(O, C), C \prec C'. \\
price_{possible}(item, X, Y) & \Longleftarrow & r(X, Y),\ in(X, item). \\
price_{possible}(cheapItem, X, Y) & \Longleftarrow & s(X, Y),\ in(X, cheapItem). \\
price(X, Y) & \Longleftarrow & price_{possible}(C, X, Y),\ \neg price_{overwrite}(C, X). \\
price_{overwrite}(C1, X) & \Longleftarrow & price_{possible}(C1, X, R1),\ C2 \prec C1, \\
& & price_{possible}(C2, X, R2),\ \neg C1 \prec C2.
\end{array}$$

Figure 7: Example of a dynamic rewriting for a datalog^{meth,flex} program

hierachy. The rule $c(X) \Longleftarrow c'(X)$ known from the previous rewritings is adapted to the new *in* predicate.

One method rule from the datalog^{meth,flex} program corresponds to several rules in the rewritten program: First, the heads of the rules defining the method *price* are modified: instead of directly deriving $price(X, Y)$ there is introduced a predicate $price_{possible}$, indicating a *possible* method definition. The first parameters denotes the class where the rule was attached to: *item* or *cheapItem*, respectively.

The method *price* is defined only when there is a possible method definition which is not overwritten. Overwriting of method definitions is indicated by the predicate $price_{overwrite}$: if this predicate holds for a tuple (c, o, r), then there is a definition for method *price* in class c, but this should be overwritten, since this is not the minimal class providing a result. Observe that in this simple example the resulting program is stratified, since no method is defined recursively. In general, it is possible to have methods and the hierachy defined recursively.

Let $\Delta = (C, P, \alpha)$ be a program of datalog^{meth,flex}. The *flexible rewriting* Λ of Δ is a datalog^{neg} program, which contains exactly the following rules:

1. For each rule r in Δ, which is not a method-rule a rule r' which is obtained from r by rewriting the method-atoms in r.

2. The following rules to realize transitivity of class-membership: $X \prec Z \Longleftarrow X \prec Y, Y \prec Z.$
   $in(O, C') \Longleftarrow in(O, C), C \prec C'.$

3. For each method m the rules:

$$m_{overwrite}(C1, X, \vec{T}) \Longleftarrow$$
$$m_{possible}(C1, X, \vec{T}, R1),\ C2 \prec C1,$$
$$m_{possible}(C2, X, \vec{T}, R2),\ \neg C1 \prec C2.$$
$$m(X, \vec{T}, Y) \Longleftarrow$$
$$m_{possible}(C, X, \vec{T}, Y), \neg m_{overwrite}(C, X, \vec{T}).$$

4. For each method-rule r with head $O.m@\vec{T} \rightarrow T$ attached to class c a rule r' which is obtained from r by rewriting the method-atoms in the body and replacing the head by $m_{possible}(c, O, \vec{T}, T)$.

Note that the resolution functions in the previous sections were defined with respect to a given interpretation and a fixed hierarchy $<$. In the case of a datalog^{meth,flex} program, to apply dynamic resolution, we first have to extract the hierarchy from the respective interpretation. Let I be an interpretation. Then denote $<_I$ the hierarchy defined by the set of super-atoms in I. For simplicity let us accept only such interpretations, which imply $<_I$ to be a forest.[5] In this slightly adapted framework we now can relate a flexible rewriting to dynamic inheritance as follows:

Theorem 4 Let Δ be a datalog^{meth,flex} program, and let Λ be the flexible rewriting of Δ. Assume Λ has a canonical model M. Then a method atom $m(o, \vec{t}, s)$

[5] In fact, the flexible rewriting behaves well also in cases where $<_I$ is a partial order, not necessarily a forest. A discussion of such situations is beyond the scope of this paper.

is present in M if and only if there is an applicable method implementation $P_{m,c}$ with respect to M^{meth} and $o.m@\vec{t} \rightarrow s$, such that $c = resolve^{dyn}(o.m@\vec{t}, M^{meth})$. □

7 Conclusion

In this paper, methods and object-oriented views are provided within a simple extension of datalog. Several approaches to inheritance are introduced and compared.

This work can be extended in several directions. One may introduce very naturally multiple inheritance in the model. Indeed, because of the declarative nature of the language, multiple inheritance seems to open wide possibilities in terms of modeling comfort. (It is much less so in procedural languages.) Another possible extension is to consider resolution based on *all* the arguments of method calls as in [CLOS]. This can be achieved using techniques of [AKW90]. Another issue is the development of specific optimization techniques for datalog^{meth} computations.

References

[AB91] S. Abiteboul and A. Bonner. Objects and views. In *Proc. SIGMOD*, 1991.

[ABW88] K. R. Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, ed., *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, 1988.

[AG88] S. Abiteboul and S. Grumbach, A Rule-Based Language with Functions and Sets, In *ACM Tods*, 16(1), pp. 1–30, 1991.

[AH88] S. Abiteboul and R. Hull. Data functions, datalog and negation. In *Proc. SIGMOD*, 1988.

[AK89] S. Abiteboul and P. C. Kanellakis, Object Identity as a Query Language Primitive", *Proc. SIGMOD*, 1989, to appear in *J. ACM.*

[AKW90] S. Abiteboul, P. C. Kanellakis and E. Waller. Methods Schemas. In *Proc. PODS*, 1990.

[ALUW93] S. Abiteboul, G. Lausen, H. Uphoff and E. Waller. Methods and Rules. Full version.

[AN86] H. Aït-Kaci and R. Nasr. LOGIN: A Logic Programming Language with Built-in Inheritance. In *J. Logic Programming,* 3, 1986.

[BL91] S. Brass and U. W. Lipeck. Semantics of inheritance in logical object specifications. In *Proc. DOOD*, 1991.

[BM92] E. Bertino and D. Montesi. Towards a logical-object oriented programming language for databases. In *Proc. EDBT*, 1992.

[Bre87] G. Brewka. The logic of inheritance in frame systems. In *Intl. Joint Conference on Artificial Intelligence*, 1987.

[CCCR$^+$90] F. Cacace, S. Ceri, S. Crespi-Reghizzi, L. Tanca, and R. Zicari. Integrating object-oriented data modeling with a rule-based programming paradigm. In *Proc. SIGMOD*, 1990.

[CLOS] D. G. Bobrow, L. Demichael, R. P. Gabriel, S. Keene, G. Kiczales, D. Moon. Common Lisp Object System Specification. In *SIGPLAN Notice*, vol. 23 Special issue, sept. 1988.

[Fit85] M. Fitting. A Kripke-Kleene semantics for logic programs. In *J. Logic Programming,* 2(4):295–312, 1985.

[KL89] M. Kifer and G. Lausen. F-logic: A higher-order language for reasoning about objects, inheritance and scheme. In *Proc. SIGMOD*, 1989.

[KLW90] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object oriented and frame-based languages. Technical report, State University of New York at Stony Brook, 1990.

[LO91] Y. Lou and Z. Meral Ozsoyoglu. LLO: An object oriented deductive language with methods and method inheritance. In *Proc. SIGMOD*, 1991.

[LV92] E. Laenens and D. Vermeir. Assumption-free semantics for ordered logic programs: On the relationship between well-founded and stable partial models. *J. Logic Computat.*, 2(2), 1992.

[McC84] J. McCarthy. Applications of circumscription to formalizing common sense knowledge. In *Proc. Nonmonotonic Reasoning Workshop*, 1984.

[Prz88] T. C. Przymusinski. On the declarative semantics of deductive databases and logic programs. In J. Minker, ed., *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufman, 1988.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume I.* Computer Science Press, New York, 1988.

[VGRS91] A. Van Gelder, K. Ross, and J. S. Schlipf The well-founded semantics for general logic programs. *JACM*, 38(3), 7 1991.