LLO: An Object-Oriented Deductive Language with Methods and Method Inheritance

Yanjun Lou and Z. Meral Ozsoyoglu

Department of Computer Engineering and Science
Case Western Reserve University
Cleveland, OH 44106, USA

Abstract

Integrating value-oriented and object-oriented data models is one of the active research directions in database field. Current Horn clause languages do not have the concept of methods and there is no polymorphism. In this paper, we consider Horn clause programs from an object-oriented perspective. Our observation is that Horn clause languages can be extended and object-oriented features can be integrated with the support of a proper data model. Based on this observation, we present a deductive object-oriented database language, LLO, together with its supporting data model, which uses meta variables as an abstract mechanism to build type/class hierarchies. Methods are defined by rules and method inheritance is achieved through typing and unification mechanisms. Procedural semantics is also presented and several related issues are addressed.

1 Introduction

Recently, there has been a lot of research interest in supporting a declarative language by an object-oriented data model, or designing declarative, object-oriented languages [ChWa 89, KiWu 89, AbKa 89, HuY 90].

While several deductive query models have been developed in the object-oriented paradigm [AbKa 89, KiLa 89], more elaboration is needed on how methods are defined on classes and how method inheritance works. Cardelli and Wegner [CaWe 85, Card 88] have developed a formal semantics for multiple inheritance in a functional paradigm. According to Cardelli [Card 88], even though an object in a subclass is an object in its superclass, the extra information contained in objects of subclasses makes structures of objects in the superclass diversified. Several questions need to be answered in order to have an object-oriented database in a deductive framework. For instance,

1. There are structural mismatches of a class and its subclasses. Deductive languages, such as Horn clauses, lack the ability to deal with inheritance.

2. There is no corresponding concept of method as in procedural languages.

Several approaches have been proposed to solve the first problem. LOGIN [AiNa 86], O-Logic and F-Logic [Maie 86, KiWu 89, KiLa 89] proposed that an object could address its attributes one by one, or access several attributes together (limiting the schema "from below" by specifying what is generally true about the class). Object is emphasized and attributes of an object are methods defined on it. But as pointed out by Cardelli [Card 88], there are some anomalies with this approach and parameterized types cannot be properly handled. Cardelli and Wegner [CaWe 85] suggested a solution by introducing quantifiers $\exists$ and $\forall$ in types.

Progress has been reported on the second problem. Object-oriented algebras [Osbo 89, StOz 90, ScSc 90] and other query languages have been developed using object identity. But the concept of method is ignored. At the same time, F-Logic [KiLa 89] uses nonground id-terms as labels to define methods. In F-Logic, a subclass can also be an instance of the superclass, and classes can be used as objects. Due to this overloading of concepts, it is confusing whether a method defined for a class is defined on its subclass as a whole or on individual objects in the class. There are also restrictions defining methods inside objects and classes. Such operations similar to the join in the relational model do not have intuitive definitions. Invention of object identifiers are addressed in IQL [AbKa 89] and ILOG [HuY 90] in a deductive framework. However, only data inheritance is discussed and there is no clear concept of method. On the other hand, HiLog [CKW 89] proposed the higher order syntax and first order semantics. Even though it did not discuss object-oriented features directly, we find that the mechanisms of defining methods are already developed there and parametric polymorphism is easily implemented. But it did not address how method inheritance can be achieved.

It is commonly expected that object-oriented query models should be compatible with the current database query models in order to utilize the accumulated techniques in this field. COL [AbGr 87] and LDL [Bee+ 87] are examples of upgrading Datalog with complex objects. IQL [AbKa 89] and ILOG [HuY 90] are aiming to build a deductive query model with object-oriented features. Login [AiNa 86] is another example for integrating data inheritance with Prolog. We follow the same route here and try to build a query model that will include current logical models (declarative and conventionally value-oriented) as its subsets.

In this paper, we will discuss a data model proposed in

[LoOz 90a] that will help solving the problem of structural mismatch between a class and its subclasses. Then we extend the Horn clauses and introduce a method concept that is comparable to the one in procedural languages. Our work is based on the following observations on the resolution procedure in Horn clauses [Lloy 84] :

- A program clause is a method. The head of the rule is its interface, and the body is its implementation (here we adopt the method concept given in [AKW 90]).

- Each goal clause (query) is a message.

- Unification algorithm dispatches the message to the proper method.

- The resolution process connects all those methods together and finishes the query evaluation.

Example 1.1 The following program in Datalog computes the transitive closure from relation r and stores the result in p.

$$p(X,Y) : -r(X,Y)$$
$$p(X,Y) : -r(X,Z), p(Z,Y)$$

Query $p(X,Y)$? is a message that will be sent to the database. The method that responds to this message ($p(X,Y)$) will be selected by the unification algorithm, and the transitive closure of relation r will be the answer.

This method (if we call it a method) is defined *solely* for relation r. If we have another relation q and want to get its transitive closure, we need to write another method. There is no sharing in Datalog at all. □

The above paraphrasing views logic programs from the object-oriented perspective. It is obvious that a predicate name may be seen as an object identifier (intension). The extension of each predicate is the value of that object. A relational data model can be seen as an *oversimplified* object-oriented system in which relations correspond to objects. That is, an object can only take set of tuples as its value. In this case, either there is no class or each class has only one object. Or, a relation can be seen as a class, the schema of the relation is the type of the class and each tuple is an object. In this case, there is no explicit object identifier. Compared with the common features of object-oriented database models [Banc 88, ZdMa 90, Kim 90], we have the following problems when relational model is viewed from an object-oriented perspective:

- No data inheritance;

- No polymorphism, both inheritance and parametric [Card 88]; Methods are defined on constant predicates (or objects) and cannot be shared;

- No encapsulation or abstraction;

- Object identifiers can not be used as components of another object; thus its modeling power is limited.

To overcome these difficulties, much work needs to be done.

• New data models to represent objects. The data model should provide support for a deductive query language with object-oriented features.

• Method definition and method polymorphism. This is crucial for an object-oriented query model.

• Higher order syntax. Object identifiers and sets are allowed to be components of objects, and methods may take objects as arguments.

As has been pointed out, method inheritance polymorphism is the major obstacle in turning logic models into object-oriented query models. The structural differences between class and its subclasses prohibit any methods on one class being applied to its subclasses in Horn clause framework. Motivated by this problem, we introduce the concept of meta variables in our data model. Meta variables are used for data abstraction, information hiding and inheritance. Type inheritance hierarchies are built up by meta variable instantiation. Part of the structure of a subtype is hidden by the meta variable in its supertype, which makes all the instances in a type having the uniform structure. We also introduce the concept of named values which can be shared through their identifiers and redundancies can be avoided. Named values, when assigned to classes, become objects.

In this paper, we propose a deductive language named LLO for object-oriented databases (abv. Logic Language for Objects) based on the data model with meta variables. One of the novel features of LLO is that it facilitates defining generic methods as well as concrete ones. That is, in LLO, rules whose left hand side predicate terms start with function id-terms are generic methods, and those start with constant object identifiers are concrete computation. Utilizing meta variables, subclass objects have the same structure as superclass objects. Consequently, a method defined on a superclass can be shared by its subclasses. So method inheritance can be achieved uniformly by checking the types associated with methods. What we need to do is to modify the unification algorithm accordingly to incorporate type/class information. Furthermore, encapsulation and abstract data types are realized by function id-terms which are the driving force in defining methods. LLO is an extension of HiLog [CKW 89] with types and other object-oriented features. COL [AbGr 87] and LDL [Bee+ 87] are subsets of LLO, since data function and groupping operation can be represented in it. By introducing named values and methods, LLO is an extension of IQL [AbKa 89]. Unlike F-Logic, where concepts are overloaded, LLO strictly separates instances and types.

The rest of the paper is organized as follows. Section 2 briefly reviews the data model which we claim has overcome the difficulties in using a deductive query language in an object-oriented environment. Section 3 introduces the formal syntax and semantics of the query language, LLO. In section 4, method definition in LLO with function id-terms is presented and the utilization of meta variables in achieving method inheritance is discussed. We also give several illustrative examples and explain how unification algorithm could be modified to achieve method inheritance. The fixpoint semantics is discussed in section 5. Finally, we summarize our presentation and discuss future research directions.

2 Data Model with Meta Variables

Each object has a state and behavior. The complex structure of the state of an object is described by its type. In this section, we briefly review the syntax of our data model and focus on structural inheritance. One of the novel features of the data model is meta variables. A meta variable is a variable which takes type as its value. Type inheritance hierarchies can be build up by instantiating meta variables in types. More detailed discussion of the data model can be

found in [LoOz 90a]. Behavior, or method inheritance, will be the topic of Section 4. The presentation here follows the notation in O_2 [Ban+ 88, LeRi 89] and IQL [AbKa 89].

We assume the existence of the following countably infinite and pairwise disjoint sets of atomic symbols:

1) basic type symbols $\mathcal{B}$ $\{B_1, B_2, \ldots,\}$, specifically $\perp$ is a basic type symbol; for each basic type B_i, there is a constant domain D_i associated with it. We denote $\mathcal{D} = \cup D$;

2) meta-variables $\mathcal{M}$ $\{m_1, m_2, \ldots\}$;

3) types $\mathcal{T}$ $\{\tau_1, \tau_2, \ldots\}$;

4) type names $\mathcal{TN}$ $\{T_1, T_2, \ldots\}$;

5) class names $\mathcal{C}$ $\{C_1, C_2, \ldots\}$;

6) object identifiers (oid's) $\mathcal{O}_{id}$ $\{p_1, p_2, \ldots\}$.

Definition 2.1 The set of *values* $\mathcal{V}$ is the smallest set containing:

1) Each element in $\mathcal{D}$ and each element in $\mathcal{O}_{id}$ is a value.

2) If $v_1, \ldots, v_n (n \geq 0)$ are values, so are $[v_1, \ldots, v_n]$ and $\{v_1, \ldots, v_n\}$. □

Definition 2.2 Let $\mathbf{O}$ be a finite set of object identifiers. A *mapping* δ for $\mathbf{O}$ is a partial function from $\mathbf{O}$ to $\mathcal{V}$ such that every identifier $p \in \mathbf{O}$ is assigned a value $v \in \mathcal{V}$. □

Note that δ is a partial function since there may exist some objects which do not have values.

An object identifier together with the value assigned to it is called *named value*. (In O_2, the term named value is used for persistency). This concept plays the role of integrating value-oriented data and object-oriented data together. Named values can be shared through their identifiers. More importantly, named values provide a mean to address values. IQL introduced a limited form of named values by the concept of relations [AbKa 89]. In our model, a named value can take a set, a tuple, or even another identifier as its value. In particular, a relation in relational model is a named value, whose identifier is the relation name and whose value is the set of tuples.

The mapping δ can be extended to values. Each application of δ, $\delta(v)$ opens up the oid's in v. $\delta^*(v)$ represents the result of repetitive application of δ until no oid's remain in the resulting value*.

Definition 2.3 Let $\mathbf{C}$ be a finite set of class names. An *identifier assignment* π for $\mathbf{C}$ is a function which assigns a finite set of object identifiers to each class name C in $\mathbf{C}$ such that if C_1 is a subclass of class C_2, denoted as $C_1 \prec C_2$, then $\pi(C_1) \subseteq \pi(C_2)$. □

A named value whose identifier is assigned to a class is called *object* of that class. A class is assigned with a set of objects. Since relations are named values, their identifiers (names) may be assigned to classes. In this case, those relations are also objects.

By finiteness, each class is also an object. For each class name C, there is an oid, o_C, $\delta(o_C) = \pi(C)$, such that the corresponding object is an instance of a class of type $\{C\}$†.

*This is always possible if the value assigned to an oid does not include the oid itself.

†o_C, as an object, will not bring the methods defined on class C with it. To this object, only the methods defined on class of type $\{C\}$ are applicable.

If there is no confusion, we also use C as the object identifier for the object corresponding to the class C instead of o_C.

Note: When we say o_C, or C, is an object of another class of type $\{C\}$, we are not talking about meta-classes. Here only the class name and its extension are involved. o_C is an ordinary object. There may be other objects whose value is a set of objects from class C. In other words, the class of type $\{C\}$ is not a class of classes.

Type expressions are represented as follows:

$$\tau = \perp \mid B \mid T \mid m \mid C \mid [\tau_1, \ldots, \tau_n] \mid \{\tau\}$$

where B is a basic type, T is a type name, C is a class name, m is a meta-variable and $\perp$ is the basic type denoting empty set. The set of types defined is denoted by $\mathcal{T}$. A *meta type* is a type which has meta variables as its components.

Each class name (type name) is associated with a type which specifies the structure of the class (type which the type name stands for), σ is a mapping from $\mathbf{C} \cup \mathcal{TN}$ to $\mathcal{T}$, i.e., $\sigma(C) = \tau$, or $\sigma(T) = \tau$, where $\tau \in \mathcal{T}$ is a type.

Class names and type names are used in types to hide structures (there is an essential difference with meta variables, as will be discussed below). The mapping σ can be extended to types, as follows.

$\sigma(\perp) = \perp$;
$\sigma(B) = B$;
$\sigma([\tau_1, \ldots, \tau_n]) = [\sigma(\tau_1), \ldots, \sigma(\tau_n)]$;
$\sigma(\{\tau\}) = \{\sigma(\tau)\}$.

Meta-type is a type in which meta-variables hide part of the structural information from the current type. The instantiation of the meta-variables is the key point in the establishment of type inheritance hierarchies.

Definition 2.4 A *meta-variable instantiation* η is a function from $\mathcal{M}$ to $2^{\mathcal{T}}$ such that every meta-variable is mapped to a set of types. For a meta-variable m, we call its image under η, $\eta(m)$, *domain* of m. □

In this paper, we restrict the number of meta-variables in a type to at most one. Types with multiple meta-variables can be represented by types that have only one meta-variable. Each type in $\eta(m)$ represents the structure that is hidden by m, and is refered as *relevant type* of m. The basic type symbol $\perp$ is used to represent the case in which the meta variable does not hide anything and is omitted if there is no confusion.

Definition 2.4 can be extended to types in a natural way, as shown below:

If $\tau = \perp$, then $\eta(\tau) = \{\perp\}$.
If $\tau = B$, where B is a basic type, then $\eta(\tau) = \{B\}$.
If $\tau = T$, where T is a type name, then $\eta(\tau) = \eta(\sigma(T))$.
If $\tau = C$, where C is a class name, then $\eta(\tau) = \eta(\sigma(C))$.
If $\tau = [\tau_1, \ldots, \tau_n]$, then $\eta(\tau) = \{[\tau_1', \ldots, \tau_n'] | \tau_1' \in \eta(\tau_1)\ or \ldots or\ \tau_n' \in \eta(\tau_n)\}$.
If $\tau = \{\tau_1\}$, then $\eta(\tau) = \{\{\tau_1'\} | \tau_1' \in \eta(\tau_1)\}$.

The transitive domain of m is denoted by η^*:
$\eta^1(m) = \eta(m)$.
$\eta^k(m) = \cup_{\tau \in \eta^{k-1}(m)} \eta(\tau) \cup \eta^{k-1}(m)$.
$\eta^*(m) = \cup_{k=1}^{\infty} \eta^k(m)$

Let τ_1 and τ_2 be two types from $\mathcal{T}$, m be a meta-variable in τ_1. If τ_2 can be obtained by substituting m in τ_1 by an

element from $\eta(m)$, i.e., $\tau_2 \in \eta(\tau_1)$, then τ_2 is called *subtype* of τ_1, and τ_1 is called *super-type* of τ_2, also denoted by $\tau_2 \prec \tau_1$.

By using meta variables, Cardelli's multiple inheritance [Card 88] and parameterized types [CaWe 85] are modeled in one system. It is straight forward to define parameterized types such as stacks, queues, etc., without incuring those anomalies discussed in [Card 88]. Types with meta variables are actually parameterized types. The importance of meta variables will be further addressed in Section 4. The following example illustrate the role played by η and the representation of multiple inheritance by meta variables. The problem addressed in [AbKa 89], i.e., legal instances with attributes that do not appear in the schema, is virtually eliminated.

In the following presentation, basic types are represented by **boldface** strings and other types and classes start with capital letters.

Example 2.5 (Type hierarchy) Suppose we have the following types:

- [*Name*, **birthyear**, m],
- [*Name*, **birthyear**, [{*Course*}, m_1]],
- [*Name*, **birthyear**, [m_2, **sal**]],
- [*Name*, **birthyear**, [{*Course*},**sal**]],

where

m, m_1 and m_2 are meta-variables, and

Course is a class name of type [**course-name, credit-point,** {*Class-Student-Record*}, **semester**].

$$\eta(m) = \{[\{Course\}, m_1], [m_2, \mathbf{sal}]\},$$
$$\eta(m_1) = \{\mathbf{sal}\}, \text{ and}$$
$$\eta(m_2) = \{\{Course\}\}.$$

The function η can be extended to types, that is,

$\eta([Name, \mathbf{birthyear}, m]) =$
$\{[Name, \mathbf{birthyear}, [\{Course\}, m_1]],$
$[Name, \mathbf{birthyear}, [m_2, \mathbf{sal}]]\},$

$\eta([Name, \mathbf{birthyear}, [\{Course\}, m_1]]) =$
$\{[Name, \mathbf{birthyear}, [\{Course\}, \mathbf{sal}]]\}$

$\eta([Name, \mathbf{birthyear}, [m_2, \mathbf{sal}]]) =$
$\{[Name, \mathbf{birthyear}, [\{Course\}, \mathbf{sal}]]\}.$

We use *Persontype* as a shorthand for [*Name*, **birthyear**, m], *Studenttype* for [*Name*, **birthyear**, [{*Course*}, m_1]], *Stafftype* for [*Name*, **birthyear**, [m_2, **sal**]] and *TAtype* for [*Name*, **birthyear**, [{*Course*}, **sal**]]. Then we have the type inheritance hierarchy as shown in Fig.1. □

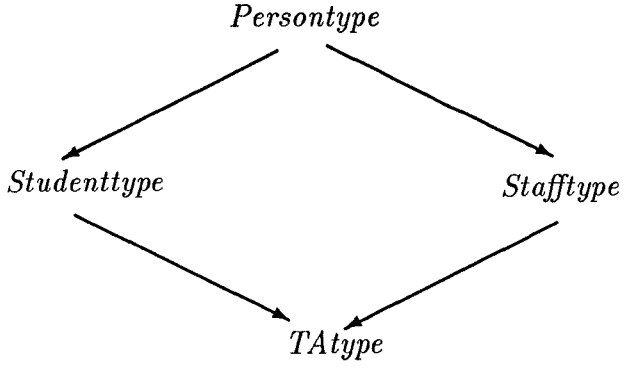

Figure 1: **Personnel Hierarchy**

Both type names, class names and meta variables play the role of data abstraction and information hiding in types. The information hidden by the type name (class name) is opened by σ, which assign each type name (class name) a type. So we may get a more *detailed* type when σ is applied to a type. On the other hand, a meta variable is associated to a *domain* of types through mapping η. The instantiation of a meta variable in a type results in a subtype. In the above example, *Course* is a class name of type [**course-name, credit-point,** {*Class-Student-Record*}, **semester**], while the meta variable m is mapped to a domain $\{[\{Course\}, m_1], [m_2, \mathbf{sal}]\}$.

Given assignment π, each type τ is mapped to a set of values as its *interpretation* $I(\tau)$.

- $I(\perp) = \emptyset$, $I(B) = D_i$, $I(C) = \pi(C)$,
  $I(m) = \cup_{\tau \in \eta^*(m)} I(\tau)$, $I(T) = I(\sigma(T))$.
- $I([\tau_1, \ldots, \tau_n]) = \{[v_1, \ldots, v_n] | v_i \in I(\tau_i), i = 1, \ldots, n\}$.
- $I(\{\tau\}) = \{\{v_1, \ldots, v_k\} \mid k \geq 0, v_i \in I(\tau), i = 1, \ldots, k\}$

A value v is of type τ if $v \in I(\tau)$. Particularly, an oid p is of type C if $p \in \pi(C)$.

Note: There is no clear agreement on whether a set is an object [Kim 90, ZdMa 90]. In our model, a set of objects is not an object[‡]. But a set is a value, which can be assigned to an object. Note that an object must have an oid and a value. First of all, $\{m\}$ is a type (*parameterized type*), where m is a meta-variable. The domain of m includes all the relevant types. All set properties such as membership, subset relationship are defined on $\{m\}$. By the type-subtype relationship defined above, every relevant set type is a subtype of $\{m\}$ and share its methods. In general, the type of a set is determined by the type of the objects that it is allowed to contain [ZdMa 90]. If T_2 is a subtype of type T_1, then every instance of $\{T_2\}$ (a set of values of type T_2) is also an instance of type $\{T_1\}$. For example, a *set of graduate students* is also a *set of students*. Both *set of students* and *set of graduate students* inherit methods from $\{m\}$ if they are in the domain of m. In addition, *set of graduate students* can inherit methods defined on *set of students*. Instead of defining type-subtype relationship by attribute set [Card 88], in our model the type-subtype relationship between set types is determined by meta variables and their domains. This is an alternative solution to the controversy on this topic.

A *database schema* S is a tuple $< \mathbf{O}, \mathbf{C}, \sigma, \prec >$, where $\mathbf{O}$ is a set of identifiers[§], $\mathbf{C}$ is a set of class names, σ is a map from $\mathbf{C} \cup \mathbf{O}$ to types, and $\prec$ is a partial order on class names. We will use $\mathcal{O}$ to denote the set of all object identifiers, i.e.,

$$\mathcal{O} = \mathbf{O} \cup (\cup_{C \in \mathbf{C}} \pi(C)).$$

Example 2.6 Consider the types given in the previous example. Let *Personnel*, *Student*, *Staff* and *TA* be the class names of type *Persontype*, *Studenttype*, *Stafftype* and *TAtype* respectively. We have $Student \prec Personnel$, $Staff \prec Personnel$, $TA \prec Student$, and $TA \prec Staff$. □

A *database instance* I_S with schema $S =< \mathbf{O}, \mathbf{C}, \sigma, \prec >$ is a tuple $< \pi, \delta >$, where π is an oid assignment for $\mathbf{C}$, and δ is a partial function from $\mathcal{O}$ to $\mathcal{V}$ satisfying the following condition:

$$\delta^*(\pi(C)) \subseteq I(\sigma(C))$$

for each $C \in \mathbf{C}$, that is, the values assigned to oid's in C must be instances of the type of C.

[‡]This is consistent with Kim [Kim 90].

[§]$\mathbf{O}$ includes the identifiers for named values.

3 The Rule Based Query Language

3.1 The Syntax

In this section, we introduce LLO: a Logic Language for Objects. We follow the generally accepted notations, such as those used in [Lloy 84]. In addition to the parentheses and logical connectives($\vee, \wedge, \neg, \rightarrow, \exists, \forall$), the typed alphabet includes the following disjoint sets of symbols:

1) typed constants and variables;
2) typed function symbols;
3) typed object identifiers (oid's).

Variables start with uppercase letters, constants and function symbols start with lowercase letters.

Id-terms:

1) A constant oid is an id-term of type C which is a class name.

2) If $t_1, t_2, \ldots, t_n$ are terms with type $\tau_1, \tau_2, \ldots, \tau_n$ respectively, f is a function symbol with type $\tau_1 \times \tau_2 \times \ldots \times \tau_n \rightarrow \tau$, where τ is a class name, then $f(t_1, t_2, \ldots, t_n)$ is an id-term of type τ. We call such a term a *function id-term.*

Id-terms are first defined in O-Logic [Maie 86, KiWu 89], and then extended in F-Logic [KiLa 89], where they are used for object identity denotation. In our model, we make extensions and use id-terms as

i. object identifiers inside literals;

ii. method calls to assign values to the oid's they denote and

iii. method definitions when they appear at the predicate symbol position in the head of a rule.

The value assigned to an id-term can be accessed by a special operation $\downarrow$, which corresponds to the δ function in the data model discussed in Section 2.

Terms:

- All id-terms, variables and constants with proper types are terms.
- If t is a term of type τ, then $\downarrow t$ is a term of type τ', which is the result of replacing class names in τ by their types. We call this term *exposed term.*
- If $t_1, t_2, \ldots, t_n$ are terms of type $\tau_1, \tau_2, \ldots, \tau_n$ respectively, then $[t_1, t_2, \ldots, t_n]$ is a term of type $[\tau_1, \tau_2, \ldots, \tau_n]$, called *tuple term.*
- If $t_1, t_2, \ldots, t_n$ are terms of type τ, then $\{t_1, t_2, \ldots, t_n\}$ is a term of type $\{\tau\}$, called *enumeration term* or *set term.*

We call set { }, tuple [ ] and $\downarrow$ *value constructors. Ground terms* are values defined in Section 2.

Note: The introduction of the $\downarrow$ operation brings the object-oriented model and value-oriented model together (cf. δ mapping in Section 2). But it violates the encapsulation. Since all id-terms denote objects, the result of a method call (through id-term) is an object. By applying $\downarrow$, we get the value assigned to the object denoted by the id-term (sounds like a printing procedure). This feature maybe useful for some other applications. For example, the nesting operation in nested relational data model can be easily expressed using a method (grouping related elements together) and a $\downarrow$ operation (get the value returned from the method). In order to achieve encapsulation, however, the application of this operation should be limited. It can only be applied to those "opened up" objects. There are some objects whose states are not accessible directly. In this case, application of $\downarrow$ is not allowed. The states of these encapsulated objects can only be accessed through methods defined on them. We will discuss this problem further in the next section.

Atoms:

1) An id-term is an atom.

2) If t is an id-term of type $\tau = [\tau_1, \tau_2, \ldots, \tau_n]$ or $\tau = \{[\tau_1, \tau_2, \ldots, \tau_n]\}$ [¶], and $t_1, t_2, \ldots, t_n$ are terms of type $\tau_1, \tau_2, \ldots, \tau_n$ respectively, then $t[t_1, t_2, \ldots, t_n]$ and $t(t_1, t_2, \ldots, t_n)$ are atoms; t is called *predicate id-term* of the atom, and $t_1, t_2, \ldots, t_n$ are called *argument terms.*

3) If t_1, t_2 are terms with the same type, then $t_1 = t_2$ is an atom (equality atom).

In LLO, the equality (=) means identical equality.

Literals:

An atom is a literal, called positive literal; if A is a positive literal, then $\neg A$ is a negative literal.

Rules and Programs:

A rule r is a clause of the form:

$$L \leftarrow L_1, L_2, \ldots, L_n,$$

where L is an atom (not an equality atom). L is called the head of the rule, denoted as *head(r).* $L_1, L_2, \ldots, L_n$ is a set of literals called body of the rule, denoted as *body*(r). All the variables that appear in the *argument terms* of the head(r) must also appear in body(r). A *program* is a finite set of rules.

In the rest of the paper, all terms and literals are typed.

Example 3.1 In COL [AbGr 87], the nesting operation is defined by the following rules:

$$s(X, f(X)) : -r(X, Y, Y').$$
$$f(X)(Y, Y') : -r(X, Y, Y')$$

where s and r are relations and f is a data function collecting data into a set. This program can only do the nesting operation for relation r.

In LLO, the nesting operation is implemented by the following program:

$$s(X, f(r, X)) : -r(X, Y, Y').$$
$$f(R, X)(Y, Y') : -R(X, Y, Y').$$

where R is a variable, r is a constant oid and $f(R, X)$ is a function id-term.

Even though this program looks like the one in COL, there are several substantial differences:

- For each constant a, $f(r, a)$ denotes an object identifier instead of a value. A value is assigned to the oid by calling method $f(R, X)$, where R is instantiated with r. In other words, $f(r, a)$ is a call for the method $f(R, X)$ which is defined by the second rule. If such a method does not exist, this oid does not have a value (a kind of null)

- R is an argument in $f(R, X)$. The second rule itself is a method and assigns value to the oid denoted by $f(R, X)$ (after R and X are instantiated) The second rule is independent of the first rule.

[¶] The type of an id-term should be a class name and τ is the type of the class in case t is an object in the class.

- If we replace the first rule by the following rule,

$$nest(R)(X, f(R, X)) : -R(X, Y, Y')$$

then the program becomes a method that can be used to do the nesting for any relation r having a *compatible* type. The result will be an object denoted by $nest(r)$. □

The following is another example illustrating object-oriented features of the language and the roles played by function id-terms.

Example 3.2 [AbKa 89] Let r be a set of edges with type [D, D] representing the graph G, where D is the type of the node of the graph. The following program transforms r into an object p whose value is a set of objects of class C. Class C is of type [D, { C}], where { C } represents all those objects for those nodes to which D has an edge. An instance of [D, {C}] is actually an adjacency list for a node of type D.

First, for each node d appearing in r, there is one object identifier denoted by the function id-term $adj(r, d)$, and these oid's are populated by calling method $adj(R, X)$[‖].

(1) $p(adj(r, X)) : -r(X, Y).$

(2) $p(adj(r, Y)) : -r(X, Y).$

The first attribute of the resulting object is the node X, the second attribute is the set of objects whose first attribute share edges with X. We use $g(R, X)$ to collect all those objects. $\downarrow g(R, X)$ will get the value assigned to $g(R, X)$.

(3) $adj(R, X)[X, \downarrow g(R, X)] : -R(X, Y)$

Finally, we have the following rule to collect all those oid's related to node X through their first attribute (node Y) into $g(R, X)$.

(4) $g(R, X)(adj(R, Y)) : -R(X, Y).$

Since $adj(R, Y)$ denotes oid, this rule simply collects all those oid's into a set and assigns it to $g(R, X)$.

For example, let r be a graph with four nodes, a,b,c and d. The set of edges is given below:

r: [a, b], [b, c], [a, d], [c, d]

First, for each node in r, there is one oid denoted by a ground id-term. Thus p is assigned a set of objects $\{adj(r, a), adj(r, b), adj(r, c), adj(r, d)\}$ (rule (1), (2)). $adj(r, a)$ is assigned the value $[a, \downarrow g(r, a)]$. By the last rule, $g(r, a)$ is assigned the value of $\{adj(r, b), adj(r, d)\}$. Since the function $\downarrow$ is to get to the value, the value assigned to $adj(r, a)$ is $[a, \{adj(r, b), adj(r, d)\}]$. Similarly, the value assigned to $adj(r, b)$ is $[b, \{adj(r, c)\}]$, the value assigned to $adj(r, c)$ is $[c, \{adj(r, d)\}]$, and the value assigned to $adj(r, d)$ is $[d, \{\ \}]$. □

Note that in a similar example in F-Logic, every node in the graph is an object already and has a descendants attribute. So one rule similar to the nesting operation is enough to transform a graph represented by a set of edges into a set of nodes, each one of which has a descendants attribute. The program in LLO, as well as in IQL, actually does much more work, transforming value-oriented data into object-oriented ones. This is because in LLO and IQL, types and values are strictly separated, which corresponds to strong typing in programming languages. This, in turn [AbKa 89], facilitates having a data model which generalizes the relational data model, most complex-object models and logical data models.

[‖]Method signature which specifies the type of the output of a method will be discussed in Section 4.

3.2 Semantics

As has been pointed out in [CKW 89], deductive languages extended with nested structures have higher order semantics even though some of them have first order syntax. LLO without $\downarrow$ is actually typed HiLog, so it has higher order syntax and first order semantics. Set terms and tuple terms can be seen as function id-terms, in the same way as suggested in [CKW 89].

The *universe*, U_L, for LLO is the set of all ground terms that can be formed out of the constants, id-terms and terms. U_L corresponds to the set of values, denoted by $\mathcal{V}$, in the data model discussed in Section 2. For a program P, the subset B_P of U_L which consists of ground id-terms appearing in P is called the *base* of P. We call P, *program under schema* $< \mathbf{O}, \mathbf{C}, \sigma, \prec >$, where $\mathbf{O}$ is a *finite* subset of B_P, $\mathbf{C}$ is a *finite* set of class names, σ is a mapping from class names and type names to types and $\prec$ is a partial order as given in Section 2. A data base instance $I =< \pi, \delta >$ is an interpretation of the program P, where π is an oid assignment for class names and δ is a mapping from identifiers to values (ref. Section 2). A database instance represents a set of ground facts similar to a Herbrand interpretation [AbKa 89].

$$\begin{aligned} ground\text{-}facts(I) = &\{C(o) | C \in \mathbf{C}, o \in \pi(C)\} \\ &\cup \{o(v) | \delta(o) = V, V \text{ is a set value}, v \in V, o \in \mathcal{O}\} \\ &\cup \{o[v] | \delta(o) = v, v \text{ is not a set value}, o \in \mathcal{O}\} \end{aligned}$$

Given an interpretation I, a *valuation* θ is a function from variables to U_L satisfying type constraints. θ can be extended to terms and literals. We have the notion of satisfaction similar to that in COL [AbGr 87].

Definition 3.3 The notion of satisfaction ($\models$) is defined by:

1. For each ground id-term t of type C, where C is a class name, $I \models t$ if and only if $C(t) \in ground\text{-}facts(I)$.
2. For each ground positive literal, $I \models p(b_1, \ldots, b_n)$ ($I \models p[b_1, \ldots, b_n]$) if and only if $p(b_1, \ldots, b_n) \in ground\text{-}facts(I)$ ($I \models p[b_1, \ldots, b_n] \in ground\text{-}facts(I)$) and $I \models b_i$ if b_i is of type C, where C is a class name; $I \models b_1 = b_2$ if only if $b_1 = b_2$ is a tautology.
3. For each ground negative literal $\neg L$, $I \models \neg L$ iff $I \not\models L$.
4. Let $r = A \leftarrow L_1, L_2, \ldots, L_m$. Then $I \models r$ if and only if for each valuation θ such that for each i, $I \models \theta L_i$ implies $I \models \theta A$.
5. For each program P, $I \models P$ iff for each rule $r \in P$, $I \models r$.

A model M of P is an interpretation which satisfies P. A model M of P is *minimal* iff for each model N of P, $ground\text{-}facts(N) \subseteq ground\text{-}facts(M) \Rightarrow ground\text{-}facts(N) = ground\text{-}facts(M)$. □

From the above definition (rule 3), it can be seen that the *closed world assumption (CWA)* is adopted in handling negation.

4 Methods and Method Inheritance

In Section 2, a data model is established to for the representational aspects of complex objects in a database. In this section, we focus on the behavioral aspect of objects.

4.1 Rules as Methods

In F-Logic, attributes are methods. The state of an object (set of attributes) is accessible by using the methods applicable to the object. In order to achieve encapsulation, a *view* concept similar to the one in relational data model is proposed to expose parts of the state to authorized users. But the concept of view, in our opinion, is not equivalent to the concept of encapsulation.

In LLO, the structural part of an object is composed of an oid and a complex state. There are two diferent ways to get access to the state depending on whether the object is encapsulated or not:

1. If the object is encapsulated, then a group of methods should be defined as interface.

2. If the object is not required to be encapsulated, the state of object can be accessed directly. For example, $\downarrow$ operation can be applied and attribute values are available to users.

Methods can also be defined on several classes/types. Sometimes methods are used mainly to achieve abstraction of computation and sharing. A rule in LLO is a method. And methods are used to get attribute values from objects or derive new information from the states of those objects. By *generic method*, we mean a method defined on several classes/types and has the properties of polymorphisms, both parametric and inheritance [Card 88, CaWe 85]. As has been pointed out in Section 1, a Datalog rule is not a generic method. More specifically, in Datalog, predicate names are *constants* which prohibit any attempt of sharing. Hence rules in LLO whose predicate terms on the left hand side are constant oid's can be considered as *concrete methods*. Example 1.1 in Section 1 defines a concrete method to get transitive closure for relation r.

We argue that with function symbols used in id-terms, we can define generic methods on classes/types. Variables in the function id-term in the head of a rule play the role of passing information from the head to the body. The types of these variables specify input argument types of the method.

4.2 Generic Methods

Each function symbol has an arity and a type associated with it. The type of a function symbol f takes the form:

$$f : \tau_1 \times \tau_2 \times \ldots \times \tau_n \rightarrow \tau$$

where n is its arity, $\tau_1, \tau_2, \ldots, \tau_n$ are the types of its arguments and τ is the type of the output object. In particular, each object identity (oid) o with type τ can be seen as a constant function of type

$$o : \ \rightarrow \tau.$$

When the head of a rule starts with a function id-term, this rule is defining a method. The type of the function is the signature of the method as in [AKW 90]. When a message (method call) is received by a method, a value is computed and assigned to the resulting object. The identifier of the resulting object is denoted by the function id-term (with all variables instantiated) at the head.

Example 4.1 It is straightforward to define a method for the transitive closure example in Section 1 **:

$$\textit{trans-closure}(R)(X,Y) : -R(X,Y).$$
$$\textit{trans-closure}(R)(X,Y) : -R(X,Z), \ \textit{trans-closure}(R)(Z,Y).$$

with signature

$$\textit{trans-closure}: \ \tau \rightarrow \tau,$$

where τ is a class name of type {[D, D]}. Type D may be a meta-variable whose domain includes all relevant types. In this example, R is a variable of type τ, *trans-closure*(R) is the predicate id-term denoting the interface of the method. If we send a message *trans-closure*(r) to the database, method *trans-closure*(R) will respond to this message. The oid of the resulting object is *trans-closure*(r) and a value is assigned to this oid by the method (in this case, the transitive closure of r). □

Definition 4.2 A *generic method by function id-term* is a rule (set of rules) with the literal at left-hand-side starting with a function id-term $f(X_1, X_2, \ldots, X_n)$ of type $\tau_1, \ldots, \tau_n \rightarrow \tau$. The body of the rule (rules) is the *implementation* of the method, the function id-term $f(X_1, X_2, \ldots, X_n)$ is the *interface* of the method, and $\tau_1, \ldots, \tau_n \rightarrow \tau$ is the *signature* of the method. □

Since predicate names can be considered as constant function id-terms, all rules in LLO are methods.

Example 4.3 The program in Example 3.2 can be turned into a generic method, *tr*, which calls other methods *adj* and *g*, to finish the transformation.

The first two rules in Example 3.2 specify a concrete transformation, since p is an object identifier, not a function id-term. In order to have a generic method, we replace p at the beginning of both rules by $tr(R)$, and all other appearances of r by R. $tr(R)$ together with its type is the interface of the method.

$$tr(R)(adj(R,X)) : -R(X,Y).$$
$$tr(R)(adj(R,Y)) : -R(X,Y).$$

Methods $adj(R,X)$ and $g(R,X)$ are as defined in Example 3.2. □

A method f with signature

$$f : \tau \rightarrow \tau'$$

is a method defined on class τ. A set of methods can be defined for each class this way. To achieve encapsulation, all access to the state of an object must use the methods defined on its class. On the other hand, methods can also be defined to achieve computation abstraction or modularity of programs. We believe that not all objects in a database need to be encapsulated. For some classes, objects are grouped together because they have common properties. In these cases, methods are used for code sharing.

In many situations n objects need to be retrieved. To accomplish this, a method can be defined on one class, using the rest of the classes as its arguments [KiLa 89]. In case those classes are symmetric, this approach is not natural [ZdMa 90]. n methods need to be defined, one for each class. The generic method with arity n discussed here may be a solution to this problem.

The advantages of using id-term in method definition are summarized in the following:

- Methods are rules.

**The same observation is made in HiLog [CKW 89] from pure logic point of view.

- Identity denotation for the resulting object and method call are combined together.

- Functional programming, object-oriented features and logic programming are integrated, since method interfaces are actually consistent with functional paradigm.

Another important feature of our approach is *lazy evaluation*. Since a ground function id-term denotes an object identifier, a method call through function id-term inside a literal need not be evaluated until the value assigned to the oid is required.

Example 4.4 [AbGr 87] Let p be an object with type [A, C], where C is a class name of type {[D, D]}. By using the trans-closure method in Example 4.1, we can transform the second attribute into its transitive closure and hide its detail by an oid.

$$q(X, \textit{trans-closure}(Z)) : -p(X, Z).$$

The program copies X over and get the transitive-closure of Z by a method call to *trans-closure*. □

4.3 Method Inheritance

Let $f(t_1, \ldots, t_n)$ be a method interface with signature

$$\tau_1 \times \ldots \tau_n \rightarrow \tau,$$

where τ is a class name of type $[\tau_{u,1}, \ldots, \tau_{u,m}]$. Since constant oid is a special case of function id-term, the method interface given above is general.

Suppose $f(t'_1, \ldots, t'_n)$ is a message (query) that is sent to the database, where $t'_1, \ldots, t'_n$ are terms of types $\tau'_1, \ldots \tau'_n$ respectively and $f(t'_1, \ldots, t'_n)$ is of type τ'. This message is calling for a method $f(t'_1, \ldots, t'_n)$ with signature

$$\tau'_1 \times \ldots \times \tau'_n \rightarrow \tau'.$$

Should method $f(t_1, \ldots, t_n)$ respond to this message?

To dispatch this message to method $f(t_1, \ldots, t_n)$, the following typing requirements should be fulfilled[††]:

1. $\tau'_1 \preceq \tau_1, \ldots, \tau'_n \preceq \tau_n$;
2. $\tau \preceq \tau'$.

Note that, the argument types of messages should be subtypes of the method, while the resulting type of the method should be a subtype of the message type.

The first requirement states that methods defined on supertypes are applicable to subtypes. The second requirement simply says that the resulting object can only be used as a supertype object. Both requirements are supported by meta variables in the data model described in Section 2. The unification algorithm in [Lloy 84] needs to be extended to check these requirements in order to dispatch a message to the proper method. Thus, in LLO, methods on one class and on several classes/types have the uniform representation and also can be handled in a uniform way.

Each meta variable has a domain, indicating the values (i.e. types) the meta variable may take. Consequently, if a variable in a literal is of type m which is a meta variable, this variable can take values of type τ if $\tau \in \eta(m)$. So from the view point of m, there is no structural differences among types in its domain. Meta variable functions as a cushion between a type and its subtypes and makes objects in a class have a uniform structure. On the other hand, Some information in the resulting object could be hidden by a meta variable in the supertype. For the example discussed in Section 2, if there exists a method with output type *Studenttype*, then the resulting object of this method can be used as an object of *Persontype* by igoring $[\{Course\}, m_1]$, but may not be used as an object of *TAtype*.

Meta variables also provide a flexible way to model parameterized types [Card 88]. The set type $\{m\}$ discussed in Section 2 is one example. All methods defined on class of type $\{m\}$ will be inherited by classes of type $\{T\}$ if $T \in \eta(m)$. More importantly, the domain of a meta variable put a constraint on the parameter. For example, instead of modeling one parameterized stack for all types, we may set up several generic stacks $stack(m_1)$, ..., $stack(m_k)$. Each meta variable has a disjoint domain. Classes of stack objects may inherit methods from a specific stack. This feature is useful for implementation and efficiency purposes.

The following example illustrates how meta variables support method inheritance.

Example 4.5 Let *Persons* be a class name of type *Persontype* with attributes Name, Birth-year and a meta-variable m. Let P be an object in *Persons*. The following is a method calculating the age of a person:

$$age(P)(N) : -P(Name, Birth\text{-}year, X), \quad N = Curr\text{-}year - Birth\text{-}year.$$

where *Curr-year* is a system variable.

Now, we want to get the age of a TA john, whose type is a subtype of *Persontype*. Suppose the value of john is $["john", 1965, [c\text{-}john, 1000]]$ where *c-john* is a set value representing all the courses john has been taken. Now we pose the following query:

$$age(john)(N)?$$

Since the type of *john* is a subtype of Persons in the method $age(P)$, the unification succeeds, therefore, turns the method into a concrete computation:

$$age(john)(N) : -john("john", 1965, (c\text{-}john, 1000)), \quad N = Curr\text{-}year - 1965.$$

The variable X in literal $P(Name, \mathbf{Birthyear}, X)$ in method $age(P)$ corresponds to the meta variable m in Persontype. Since $[Courses, \mathbf{sal}]$ is in transitive domain of m, X will absorb $[c\text{-}john, 1000]$. □

In LLO, method names can be overloaded. Different signatures, when associated with a method name (function symbol), specify different methods. When a message is sent to the database, the signature (type) of the message will determine which method should be used to respond to it. Even though two methods have the same name, different argument types will result in different computation. For example, in a department store database, we may apply a *size* method to shoe objects and slack objects. Since shoe and slack are objects of different types, two different methods will respond to the calls, with the sizes returned in different measurements.

Method overriding can also be achieved in LLO. A method defined on supertype can be overriden by a method with same name in the subtype. When the unification dispatches a message, it will search the class inheritance hierarchy bottom up to find the method with the "best" matching type.

[††]In the following presentation we use $\preceq$ to denote $\prec$ or $=$.

From the above observations, it is clear that

1) Typing plays an important role in method inheritance. Method inheritance is achieved following the type/class hierarchies. The unification algorithm should do the type checking to guarantee the correct inheritance.

2) Unlike the first order logic, where predicate symbols must be the same for a successful unification, predicate id-terms play an important role in the unification process.

Note: In LLO, as well as in HiLog, variables are allowed to be used as arguments in the predicate term of a message. This may cause some problems. Although the function symbol provides information about which method should respond to this message, variables in the predicate id-term may not have finite domain. That is, if the type of a variable is a class name, it ranges over the objects of the class. However, if the type of a variable is a type (not a class name), its domain maybe infinite (for example, integer). In both cases, many objects may be generated for the message due to different binding of variables. Ambiguity may arise as what is the response to the message, the whole set of objects or only one of them. The same analysis can be applied to the case where variables are used as predicate terms. More research is needed to ensure termination and safety.

5 Fixpoint Semantics

In this section, we give a brief analysis of the fixpoint semantics similar to that given in LDL and COL. While the $\downarrow$ operator brings the object-oriented database system and value-oriented system together, it also makes the semantics of the language complicated. Thus our discussion here will be focused on the $\downarrow$ operation.

In LLO, ground id-terms are playing different roles, as predicate terms and values. The $\downarrow$ operator, when applied to an ground id-term, will get the value assigned to the oid the id-term denotes. However, the id-term must be populated first. For an id-term $f(X,Y)$, $\downarrow f(X,Y)$ is equivalent to a data function in COL. The grouping (or nesting) operation in LDL can also be simulated by $\downarrow$ operation and a function id-term. Due to this connection, the stratification of LLO program bears some similarity to that of COL [AbGr 87] and LDL [Bee+ 87]. On the other hand, LLO without the $\downarrow$ operation is typed HiLog. So LLO without the $\downarrow$ has higher order syntax and first order semantics [CKW 89]. Unlike data functions in COL or grouping terms in LDL, where they must be computed prior to the predicate terms in which they appear, ground id-terms in literals are themself *values*. The oid's denoted by ground id-terms are not required to be populated at the time those literals are computed.

In order to define stratified programs in LLO, we use the concept of *determinant*[‡‡]. Let P be a LLO program. All predicate id-terms and exposed terms (remember an exposed term is a term with a $\downarrow$ preceding it) in P are *determinant* terms. The predicate id-term in the head of a rule is also called *defined id-term of the rule.*

We say a determinant term is a *total determinant* if one of the following conditions hold true:

1. the determinant is an exposed term;

2. the determinant term is the predicate id-term of a negative literal.

A determinant term is called *partial determinant* if it is not a total determinant.

Consider, for instance, the following LLO rule:

$$f(X)(X, \downarrow g(Y)) : -X(Y,Z), \neg s(Y)$$

$f(X)$ is the defined id-term of the rule, $g(Y)$ and $s(Y)$ are total determinants and X is a partial determinant(note the roles played by X on the left hand side and on the right hand side are different).

A term with a $\downarrow$ operator preceding it should be computed prior to the literals in which it appears and the defined id-term of the rule.

Let X be the defined id-term of a rule r. A total determinant Y in r is also total determinant of X, denoted as $X < Y$. That is, Y must be populated before X. Otherwise, Y is a partial determinant of X, denoted as $X \leq Y$, meaning Y must be populated no later than X.

Program P is *stratified* if there is no sequence of determinant terms in the rules of P of the form:

$$t_1 \Delta_1 t_2 \ldots \Delta_{k-1} t_k$$

such that t_1 is the same as t_k, $\Delta_i \in \{<, \leq\}$ for $i = 1, 2, \ldots, k-1$ and at least one Δ_i is $<$.

The following proposition [ABW 87] is also applicable to LLO programs:

Proposition 5.1 Let P be a program and $\mathbf{S}$ be the set of determinant terms appearing in P. Then P is *stratified* iff there is a partition $S_1, S_2, \ldots, S_l$ of $\mathbf{S}$ such that

$$t_1 \leq t_2, t_1 \in S_i \Rightarrow \exists j (i \leq j, t_2 \in S_j),$$
$$t_1 < t_2, t_1 \in S_i \Rightarrow \exists j (i < j, t_2 \in S_j)$$

For each partition $S_1, S_2, \ldots, S_l$, program P is partitioned into $P_1, \ldots, P_l$ [ABW 87]. Each P_i $(1 \leq i \leq l)$ is called a *stratum* or *layer*. If $l = 1$ then P is a *monostratum program.* Otherwise, it is a *multistrata program.*

The program in Example 3.2 is stratifiable, even though id-term $g(R,X)$ and $adj(R,X)$ are mutually recursive. The ground-term $g(R,X)$ and $adj(R,X)$ denote oid's when they appear inside literals. The interesting part is the intension of the object. Method $g(R,X)$ simply groups together all those oid's. This benefit comes from the separation between intension and extension part of an object.

Example 5.2 Consider rule from [AbGr 87]:

$$f(Y)(X) : -s(Y, f(X)).$$

This rule as a program in COL is not stratified, since $f(X)$ is a total determinant and $f < f$. But the same program in LLO is stratified. Unlike the data function in COL, the id-term $f(X)$ in the body, if instantiated, denotes an object identifier. And we have $f \leq s$, which is the only relationship between determinants.

Let s be assigned the value

$$s : \{[1, f(2)], [2, f(3)], [1, f(3)]\}.$$

The result will be: $f(1) : \{2, 3\}$, $f(2) : \{3\}$. □

For a monostratum LLO program, its fixpoint semantics can be defined similar to those defined in COL and LDL. The fixpoint operator is applied to the extensional input iteratively until no more facts can be derived. For a multistrata

[‡‡]Note that the term "determinant" is also used in COL [AbGr 87]. Here we extend it for LLO.

program, the fixpoint operator is applied stratum by stratum, the result of a lower stratum is the extensional input to the next stratum.

More semantic analysis of LLO programs can be found in [LoOz 90c].

6 Conclusion

In this paper, we established a data model which we believe will provide proper support for deductive object-oriented languages. Then we presented a typed language LLO, integrated with object-oriented features, such as object identity, methods and method inheritance. Most importantly, we focused on extending current Horn clause languages so that the accumulated techniques could be inherited.

Meta variables are introduced as a mean of data abstraction and information hiding. Instead of building inheritance hierarchies by attribute set of types, as in [Card 88], inheritance hierarchies are established by instantiation of meta variables. This approach solves some problems raised in [Card 88]. The concept of named values plays the role of bridging value-oriented data models and object-oriented data models together. A relation can be a named value and nothing is encapsulated, or assigned to a class for method sharing. In the later case, relation can also be encapsulated.

In LLO, rules are methods. Generic methods can be defined using function id-terms. The variables in function id-terms provide a vehicle to pass information from the head of a rule to its body. With the support of the data model we developed, method inheritance is made possible. The unification algorithm dispatches a message to a proper method using the signature information of the method and the message. The semantics of the language is no more complex than LDL or COL. However, in LLO, since the unification will also handle types with meta variables in addition to set types, one of the drawbacks is efficiency.

More research is needed on problems such as variables as id-terms and variables as arguments of id-terms. Another important topic is inference on meta information such as types of objects and class-subclass relationships. This is accomplished in F-Logic by overloading concepts.

Acknowledgements

The authors gratefully acknowledge the comments provided by the reviewers which helped improving the readability of the paper.

References

[AbGr 87] Abiteboul S., and S. Grumbach, *COL: a Language for Complex Objects based on Recursive Rules*, abstract in Proc. Workshop on Database Programming Languages, Rodkoff(1987)pp271-293.

[AbKa 89] Abiteboul S., and P. Kanellakis, *Object Identity as a Query Language Primitive*. In Proc. ACM SIGMOD, pp. 159-173(1989).

[ABW 87] Apt, K. R., H. A. Blair and A. Walker, *Towards a Theory of Declarative Knowledge*, Foundation of Deductive Databases and Logic Programming, Ed. by J. Minker

[AKW 90] Abiteboul S., P. C. Kanellakis and E. Waller, *Method Schemas*. To appear in PODS'90.

[AiNa 86] Ait-Kaci, H. and R. Nasr,*LOGIN: A Logic Programming Language With Built-in Inheritance*, J. Logic Programming, 3:185-215, 1986

[Banc 88] Bancilhon, F.,*Object Oriented Database Systems*. In Proc. ACM PODS, pp.240-250(1988).

[Ban+ 88] Bancilhon, F., et al.,*The Design and Implementation of O_2, an Object-Oriented Database System*. In Proc. OODBS2 Workshop, Badmunster RFA, 1988

[Bee+ 87] Beeri, C., et al.,*Sets and Negation in a Logic Database Language(LDL1)*, ACM PODS, pp. 21-37 (1987).

[Card 88] Cardelli, L.,*A Semantics of Multiple Inheritance*. Information and Computation, 76:138-164,1988.

[CaWe 85] Cardelli, L., and P. Wegner, *On Understanding Types, Data Abstraction and Polymorphism*. In Computing Surveys, Vol.17, No.4, Dec. 1985.

[ChWa 89] Chen, W. and D. S. Warren, *C-logic for Complex Objects*, Proc. of PODS, March 1989.

[CKW 89] Chen,W., M. Kifer, D. S. Warren, *HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs*. In Proc. of the North American Conference on Logic Programming, Vol.2, pp.1090-1114, 1989.

[HuY 90] Hull, R., M. Yoshikawa, *ILOG: Declarative Creation and Manipulation of Object Identifiers (Extended Abstract)*, VLDB'90.

[KiLa 89] Kifer, M., and G. Lausen, *F-Logic: A Higher-Order Language for Reasoning about Objects, Inheritance and Scheme*. In Proc. ACM PODS, pp. 134-146(1989).

[KiWu 89] Kifer, M. and J. Wu, *A Logic for Object-Oriented Logic Programming(Maier's O-logic: Revisited)*. In Proc. ACM PODS, pp.379-393 (1989).

[Kim 90] Kim, W., *Object-Oriented Databases: Definition an Research Directions* , IEEE Trans. on Knowledge and Data Engineering, Vol. 2, No. 3, 1990

[LeRi 89] Lecluse, C., P. Richard, *Modeling Complex Structures in Object-Oriented Databases*. ACM PODS, pp. 360-368. (1989)

[Lloy 84] Lloyd, J. W., *Foundations of Logic Programming*, Springer-Verlag, Berlin, 1984.

[LoOz 90a] Lou, Y. , and M. Ozsoyoglu. *Types, Objects and Inheritance in an Object-Oriented Data Model*, TR-90-10, Department of Computer Engineering and Science, Case Western Reserve University, March 1990.

[LoOz 90b] Lou, Y. , and M. Ozsoyoglu. *Meta Variables and Inheritance in an Object-Oriented Data Model*, Submited to Conference.

[LoOz 90c] Lou, Y. , and M. Ozsoyoglu. *LLO: A Deductive Language with Methods and Method Inheritance*, TR-90-42, Department of Computer Engineering and Science, Case Western Reserve University, Dec. 1990.

[Maie 86] Maier, D., *A Logic for Objects, Proc. of the Workshop on Foundations of Deductive Databases and Logic Programming*, Washington D.C., pp. 6-26, August 1986.

[Osbo 89] Osborn, S. L., *The Role of Polymorphism in Schema Evolution in an Object-Oriented Database*. IEEE Trans. on Knowledge and Data Engineering, Vol. 1, No. 3, Sept. 1989.

[ScSc 90] Scholl, M. H. and H.-J. Schek, *A Relational Object Model*, ICDT'90, Paris.

[StOz 90] Straube, D. S. and M. T. Ozsu, *A Model for Queries and Query Processing in Object-Oriented Databases*. Manuscript, University of Alberta, 1990.

[ZdMa 90] Zdonik, S. B. and D. Maier, *Foundamentals of Object-Oriented Databases*, in Readings in Object-Oriented Database Systems, S. B. Zdonik and D. Maier (eds), pp. 1-32, 1990