Datalog, Units and Information Hiding

Foto Afrati
National Technical
University of Athens

Isambo Karali
University of Athens

Theodoros Mitakos
National Technical
University of Athens

Abstract

In this paper, we propose the introduction of structured units in datalog. This allows modularity and information hiding. Units are divided into a public and a private part. This structure calls for new composition operators which are defined here. These operators express various forms of inheritance including the common ones in the object oriented framework.

1 Introduction

In object oriented databases, encapsulation and inheritance are major issues [ABD$^+$89]. Combining object oriented and deductive databases meets the needs of modern database applications. This combination benefits from the declarativeness and the clean formal model of deductive databases and the rich structural features of the object oriented framework. Datalog is a database query language based on logic programming and consists of function-free Horn clauses. According to [Ull88], it provides the simplest data model and for non-recursive rules is equivalent to positive relational algebra. Several interesting proposals have been made that introduce object oriented features into the deductive framework [KLW94, ALUW93, KA89]. Moreover, inheritance has been treated successfully by introducing principles from modular logic programming [Bug92, BLM94, AKM97]. Modularity is a special form of program composition that allows incremental development and code reusability. One of its forms in logic programming is supplied by units [MP89]. Units consist of named pieces of code and each unit has its own name space.

Encapsulation in the logic programming framework has been treated in a variety of works. In the following, we briefly present some representative approaches.

F-Logic [KLW94] treats encapsulation as type correctness approach. A program in F-Logic is a collection of sets of named clauses. The keywords *private, public* and *export to* are meta-annotations attached to the signature of every rule in the program. This model of type correctness is weaker than desired. A well-typed program may become ill-typed in a later state of its life.

In [BJ94] the logical foundations of encapsulation are studied. A deductive object oriented language with encapsulation is presented. The language is affected by the F-Logic data model. Proof theoretic, model theoretic and fixed point semantics are given and proved equivalent. However, the proposed language lacks the notion of inheritance.

In [BMPT94], the language of program expressions encompass encapsulation as a unary operation over a program expression. A program expression can be a plain logic program or a synthesis of logic programs. Encapsulation substitutes a program expression with its least Herbrand model. This approach is similar to [BLM92] where a program is visible from other programs only through its logical consequences.

In this paper, we introduce structure into units that allows information hiding. Each unit consists of two parts, a public part and a private part. Information from the public part is visible to other units though the private part is invisible. Units can interact invoking goals from other units capturing this way the message-passing mechanism in object-oriented terms. This structure calls for new composition operators as the usual modular programming operators do not apply in our case. The operators were defined along the lines of logic programming composition operators such as the union [MP88] and the overriding union [Bug92] operators. By the union operator, two programs are combined to produce one that consists of the set-theoretic union of the clauses of the original programs. Using the overriding union, though, clauses of the second program that define predicates already defined in the first program are not included in the result. In the paper, we show that the new operators are capable of expressing various forms of inheritance in the object oriented framework, including the common ones, namely monotonic and overriding inheritance.

The paper is organized as follows. In the following section, we present the syntax of the language that we propose. In Section 3, we give a) operational, b) model theoretic and c) fixed point semantics; moreover, we prove that the model theoretic and the fixed point semantics coincide. In Section 4, the composition operators are introduced. The applicability of those operators are illustrated in expressing inheritance when information hiding is an issue.

2 Syntax

In classical logic programming there are no tools for structuring and modularizing programs [Llo87]. In this section, we present the syntax of the language h(ide)-log which extends the traditional notion of unit by defining private and public clauses. Informally, a predicate of a public clause can be called from any unit but a predicate of a private one can be called only from a clause that belongs to the same unit. The syntax and the behavior of the h-log units are illustrated by some examples.

The alphabet of h-log consists of a finite set of unit names U, a finite set of predicate names P, a countable set of variables V and a finite set of constants C. To every $u \in U$ we assign two sets of clauses; u_{priv} is the set of private clauses of unit u and u_{publ} is the set of public clauses of unit u. The clauses that correspond to sets u_{priv} and u_{publ} are called *unit implementation*. The set of ground instances of the clauses of a unit implementation is denoted as $|u|$. Moreover, if p is a predicate name, the set of clauses that have p at their head is called *implementation of* p. If S is a set of clauses then we denote with $\pi(S)$ the set of predicate names defined by the clauses of S. If A is an atom then $pred(A)$ is the predicate of A.

Datalog is a database query language based on logic programming and consists of function symbol free Horn clauses. The clauses in our proposal follow the syntax of positive datalog clauses extended with the operator $>$. e.g. $p :- q, u > r$. The operator $>$ accepts two operands. The first operand is a unit identifier u, the second is a goal r. It means that the goal r must be evaluated using the public predicates, $\pi(u_{publ})$, of unit u.

We assume that $\pi(u_{priv}) \cap \pi(u_{publ}) = \emptyset$.

In the sequel of this section we give examples to illustrate the syntax and functionality of units in h-log.

Example 1

In this example, consider a program that consists of two units, namely unit *bank* and unit *goods*.

unit *bank*
Public
max_draw(X,Y):-amount(X,Z),
    credit(X,K), Y=Z+K.
Private
amount(smith,300).
amount(john,600).
credit(smith,600). credit(john,200).

unit *goods*
Public
can_buy(X,Y):-
    bank > max_draw(X,Z),
    cost(Y,K),lessthan(K,Z).
Private
cost($g1$,100). cost($g2$,1000).

In this example customer X can buy a good Y if the good costs less than or equal to the maximum amount of money that this customer can draw from the bank. Although unit *goods* knows how much money can a customer draw from the bank it does know neither the money that the customer has in his bank account nor the credit that is given to this customer. That is because max_draw/2 is implemented by a public clause but both amount/2 and credit/2 are implemented by private clauses for the unit *bank*. In this way, we achieve information hiding.

Example 2

In the following, we see the implementation of financial policy for unit *employee*.

unit *employee*
Public
salary(X,Y) :- gets(X,Y).
Private
gets(X,Y):- belongs(X,C), climax(C,Y).
climax(a,500). climax(b,750). climax(c,1000).
belongs(smith,a). belongs(john,b). belongs(mary,c).

Imagine that in some future time the financial policy for employees changes. The unit *employee* changes in the following way.

unit *employee*
Public
salary(X,Y) :- earn(X,Y).
Private

rate(10). earn(X,Y):- days(X,D),hours(X,H),rate(R),Y=D*H*R.
days(smith,15). days(john,15). days(mary,20).
hours(smith,4). hours(john,6). hours(mary,6).

This example exhibits some of the engineering benefits we gain by keeping a part of a unit implementation private to this unit only. Other units do not know about the changes that were made to the implementation of predicate salary/2. Thus, the predicate salary/2 acts as an interface that makes possible changes of financial policy transparent to other units. In this way, we achieve information hiding, code reusability, generality and maintainability.

3 Semantics

In this section, we give clean and natural semantics to h-log programs. We consider operational, model theoretic and fixed point semantics. The semantics enables us to express formally the notion of encapsulation in modular logic programming.

3.1 Operational Semantics

The operational semantics of h-log is given as follows. The proof predicate $\vdash$ is defined by the following inference rules, where ϵ is the identity substitution and $\Box$ is the empty formula:

Empty formula $\frac{}{u \vdash_{\epsilon} \Box}$ **Conjunction** $\frac{u \vdash_{\theta} G_1 \;\wedge\; u \vdash_{\sigma} G_2\theta}{u \vdash_{\theta\sigma} G_1, G_2}$

Atomic formula $\frac{u \vdash_{\sigma} G\theta}{u \vdash_{\theta\sigma} g}, \left(\begin{array}{c} h\text{:-}G \in u \\ \theta = mgu(g, h) \end{array} \right)$ **Message passing** $\frac{\hat{u} \vdash_{\theta} G \wedge pred(G) \in \pi(\hat{u}_{publ})}{u \vdash_{\theta} \hat{u} > G}$

The first rule states that the empty formula can be derived in any unit with empty substitution. The **Atomic formula** rule states that an atomic formula can be derived by a unit u if it can be derived by a clause of unit u . It allows for an atomic formula to be derived by a public or a private clause. The **Conjunction** rule states that a conjunction of formulas is derived if each conjunct is derived in turn. The last rule states that the evaluation of a message goal $\hat{u} > G$ is reduced to the evaluation of the goal G in the unit $\hat{u}$ provided that the predicate of G is a public one for $\hat{u}$, no matter where the invocation of this message goal took place.

3.2 Model Theoretic Semantics

In this paragraph, we define model theoretic semantics for h-log programs via a minimal model. In this framework, S(tructured)-interpretations are introduced and T(uple)-interpretations [Bug92] are enhanced. T-interpretations are structures that interpret a set of units. A S-interpretation interprets a single unit. Our approach is similar to [Bug92] but we assign to each unit u a set of public and a set of private ground atoms. A S-interpretation corresponds to a unit $u \in U$ and a tuple of structured interpretations corresponds to a program $P(U)$.

We show that our semantics is solid by proving that the model intersection property holds (Theorem 3.3).

For a unit u we define the structure $H(u) =< H(u)_{publ}, H(u)_{priv} >$ where $H(u)_{publ} = \{Atom|Atom : -G \in |u_{publ}|\}$ and $H(u)_{priv} = \{Atom|Atom : -G \in |u_{priv}|\}$.

Definition 3.1 *A structured interpretation, S-interpretation, for a unit u is defined as* $h =< h_{publ}, h_{priv} >$, *where* $h_{publ} \subseteq H(u)_{publ}$ *and* $h_{priv} \subseteq H(u)_{priv}$.

Definition 3.2 *For a set of units* U *we define the T-interpretation* I *as* $< I(u) : u \in U >$ *and* $I(u)$ *is a S-interpretation that corresponds to unit* u.

In the following we formalize the notion of satisfiability for our framework.

Let h be any S-interpretation and I a T-interpretation then

$h_{priv} \models_I G \Leftrightarrow G \in h_{priv}$

$h_{publ} \models_I G \Leftrightarrow G \in h_{publ}$

$h \models_I G \Leftrightarrow h_{priv} \models_I G \vee h_{publ} \models_I G$

$h \models_I G_1, G_2 \Leftrightarrow (h \models_I G_1 \textit{ and } h \models_I G_2)$

$h \models_I r : -G \Leftrightarrow (h \models_I G \Rightarrow h \models_I r)$

$h \models_I u > G \Leftrightarrow I(u)_{publ} \models_I G$

Proposition 3.1 *Let* U *be a set of units and* $u \in U$. *Let r:-G be a clause of* u_{priv} *(resp.* u_{publ}*). Let* h *be a S-interpretation of* $P(U)$ *and* I *a T-interpretation of* $P(U)$. *Then, if* $h \models_I$ *r:-G and* $h \models_I G$ *then* $h_{priv} \models_I r$ *(resp.* $h_{publ} \models_I r$*).*

According to the previous definition of satisfiability, a message goal $u > G$ is true in the S-interpretation h with respect to the T-interpretation I iff it is satisfied by the public part of the S-interpretation of unit u with respect to the T-interpretation I.

Definition 3.3 *A T-interpretation* I *is a T-model for a program* $P(U)$ *iff for every* $u \in U$ *and each clause* C *of* $|u|$, C *is true in* $I(u)$ *with respect to* I.

Proposition 3.2 *The set of T-models that is associated with a program* $P(U)$, $U = \{u_1, u_2, \ldots, u_n\}$, *is not empty since* $I_H =< H(u_1), H(u_2), ..., H(u_n) >$ *is a T-model.*

A S-interpretation $I(u)$ that is associated with a unit u consists of two parts, i.e. $I(u) =< I(u)_{publ}, I(u)_{priv} >$. The two components are intended to interpret the public and private part of a unit u, accordingly. A set of units U is associated with a T-interpretation I.

We define the operators $\sqcap_{\mathcal{I}}$ and $\sqcup_{\mathcal{I}}$ over the set of T-interpretations that are associated with a program $P(U)$.

$I_1 \sqcap_{\mathcal{I}} I_2 = I$ such that $\forall u \in U,\ I(u) =< I_1(u)_{publ} \cap I_2(u)_{publ}, I_1(u)_{priv} \cap I_2(u)_{priv} >$ and $I_1 \sqcup_{\mathcal{I}} I_2 = I$ such that $\forall u \in U,\ I(u) =< I_1(u)_{publ} \cup I_2(u)_{publ}, I_1(u)_{priv} \cup I_2(u)_{priv} >$

Theorem 3.3 *The intersection,* $\sqcap_{\mathcal{I}}$, *of two T-models for an h-log program* $P(U)$ *is a T-model for* $P(U)$.

Proof
Let I_1 and I_2 be T-models for $P(U)$. If $I_0 = I_1 \sqcap_{\mathcal{I}} I_2$, we will prove that for any $u \in U$ and for each C of $|u|$, $I_0(u) \models_I C$.

Let $C = h :- g_s$ be a ground instance of a clause of the public part of a unit $u \in U$. Then C is true in $I_1(u)$ with respect to I_1 and C is true in $I_2(u)$ with respect to I_2.

Let g_s be true in $I_0(u)$ with respect to $I_0 \Rightarrow$
g_s true in $I_1(u)$ with respect to $I_1 \Rightarrow$
h true in $I_1(u)$ with respect to $I_1 \Rightarrow$
$h \in I_1(u)_{publ}$. Similarly, we prove that $h \in I_2(u)_{publ}$. Thus, $h \in I_0(u)_{publ}$. Consequently, we proved that if g_s is true in $I_0(u)$ with respect to I_0 then h is true in $I_0(u)$ with respect to I_0.

The proof is similar for the private clauses.

Corollary 3.4 *The minimal T-model, $M_{P(U)}$, exists for any h-log program $P(U)$, as the intersection of all its T-models.*

3.3 Fixed Point Semantics

In definite logic programming, a logic program has a minimal model that coincides with the least fixed point of its immediate consequence operator. In this section, we define the immediate consequence operator for an h-log program and we prove that its least fixed point coincides with the program's minimal T-model (Theorem 3.7).

There is an ordering $\sqsubseteq$ among T-interpretations for an h-log program $P(U)$. This is, $I_1 \sqsubseteq I_2 \Leftrightarrow \forall u \in U, I_1(u)_{priv} \subseteq I_2(u)_{priv}$ and $I_1(u)_{publ} \subseteq I_2(u)_{publ}$. The set of T-interpretations that are associated with $P(U)$, $U = \{u_1, u_2, \ldots, u_n\}$, form a complete lattice *Lat* with bottom element $I_\perp =< \emptyset, \emptyset, \ldots, \emptyset >$ and top element $I_\top =< H(u_1), H(u_2), \ldots H(u_n) >$. For any subset of *Lat*, the operators $\sqcup_{\mathcal{I}}$ and $\sqcap_{\mathcal{I}}$ provide its least upper bound (lub) and its greatest lower bound (glb), respectively.

First, we define the immediate consequence operator for a unit $u \in U$ with respect to a T-interpretation I. That is $T_{u,I(u)} =< T_{u_{publ},I(u)}, T_{u_{priv},I(u)} > = < \{A | A :- G \in |u_{publ}|$ and $I(u) \models_I G\}, \{A | A :- G \in |u_{priv}|$ and $I(u) \models_I G\} >$.

Then, we define the immediate consequence operator for $P(U)$. That is, $T_{P(U)}(I) =< T_{u,I(u)} | u \in U >$. $T_{P(U)}$ is a transformation from T-interpretations to T-interpretations over $P(U)$.

Theorem 3.5 *Let $P(U)$ be a program in h-log. Then the mapping $T_{P(U)}$ is continuous.*

Proposition 3.6 *Let $P(U)$ be a program and I a T-interpretation of $P(U)$. Then I is a T-model for $P(U)$ iff $T_{P(U)}(I) \sqsubseteq I$.*

The transformation $T_{P(U)}$ is also monotonic, as it is continuous. Then, its powers are defined as follows:
$T^0_{P(U)} = I_\perp$
$T^{n+1}_{P(U)} = T_{P(U)}(\ T^n_{P(U)})$
$T^\omega_{P(U)}(= T_{P(U)} \uparrow \omega) = \bigsqcup_{n<\omega} T^n_{P(U)}$

The transformation $T_{P(U)}$ is continuous. Then, it has a least fixed point. The least fixed point is reached if we iterate the transformation repeatedly starting from $I_\perp$. Moreover the minimal T-model of a program coincides with the transformation's least fixed point.

Theorem 3.7 $M_{P(U)} = lfp(T_{P(U)}) = T_{P(U)} \uparrow \omega$.

Proof
$M_{P(U)} = \sqcap_{\mathcal{I}}\{I : I$ is a T-model for $P(U)\}$
$= glb\{I : I$ is a T-model for $P(U)\}$
$= glb\{I :\mathrm{T}_{P(U)}(I) \sqsubseteq I\}$ by Proposition 3.6
$= lfp(T_{P(U)})$ since $T_{P(U)}$ is monotonic
$= T_{P(U)} \uparrow \omega$ since $T_{P(U)}$ is continuous.

Example 3
As an example consider the program $P(U)$ where $U = \{$u1,u2$\}$.

```
unit u1                             |          unit u2
Public                              |          Public
 q(X,Y):- u2>r(X,Y).                |           s(g,d).
 q(X,Y):- u2>r(X,Y),q(Y,Z).         |           r(X,Y):- p(X,Y).
Private                             |          Private
 r(X,Y):- u2>s(X,Y).                |           p(a,b).
                                    |           p(b,g).
```

The immediate consequence operator is carried on repeatedly until its least fixed point is reached. That is:
$T_{P(U)}(I_\perp) =<< \emptyset, \emptyset >, < \{s(g,d)\}, \{p(a,b), p(b,g)\} >>= I_1$
$T_{P(U)}(I_1) =<< \emptyset, \{r(g,d)\} >, < \{s(g,d), r(a,b), r(b,g)\}, \{p(a,b), p(b,g)\} >>= I_2$
$T_{P(U)}(I_2) =<< \{q(a,b), q(b,g)\}, \{r(g,d)\} >, < \{s(g,d), r(a,b), r(b,g)\}, \{p(a,b), p(b,g)\} >>= I_3$
$T_{P(U)}(I_3) =<< \{q(a,b), q(b,g), q(a,g)\}, \{r(g,d)\} >, < \{s(g,d), r(a,b), r(b,g)\}, \{p(a,b), p(b,g)\} >>= I_4$
$T_{P(U)}(I_n) = T_{P(U)}(I_{n+1}), n \geq 4$
This is also the minimal T-model of $P(U)$.

4 Composition of Units

In modular logic programming each unit (module) is considered as an autonomous program. From an object-oriented approach, it can be viewed as a class. In software engineering, it is useful to compose new units from existing ones. This composition can be done in various ways. In this section, we define new composition operators that apply to units with public and private clauses. In addition, we examine the hierarchical composition of units. These operators generalize several well known paradigms of inheritance, such as monotonic or non-monotonic inheritance.

4.1 Composition operators

Several composition operators that apply to units (or programs, or modules, or theories) have been consider in the literature [BLM94]. These operators apply whenever a unit is used as a single set of clauses. In our approach, though, a unit consists of two sets of clauses, namely its private and its public clauses. Therefore new operators should be defined to handle composition of units in our framework. We define four new primitive composition operators and we classify them into two categories, namely monotonic and overriding operators. The primitive operators can be combined to provide some other convenient operators.

Let u_p and u_q be two units. In the following, we assume that $\pi(u_{p_{publ}}) \cap \pi(u_{q_{priv}}) = \emptyset$ and $\pi(u_{q_{publ}}) \cap \pi(u_{p_{priv}}) = \emptyset$. This assumption can be made in most cases (e.g. renaming can be applied). We consider cases where renaming destroys the intended meaning as ill-defined.

In the first category, the composition of units takes place considering only the union of their clauses. This approach corresponds to monotonic composition of clauses.

The operator $\sqcup_{publ}$ is defined as:

$u_p \sqcup_{publ} u_q = u_z$
$u_{z_{publ}} = \{A : -G \in u_{p_{publ}}\} \cup \{A : -G \in u_{q_{publ}}\}$
$u_{z_{priv}} = \{A : -G \in u_{p_{priv}}\}$

The operator $\sqcup_{priv}$ is defined as:

$u_p \sqcup_{priv} u_q = u_z$
$u_{z_{publ}} = \{A : -G \in u_{p_{publ}}\}$
$u_{z_{priv}} = \{A : -G \in u_{p_{priv}}\} \cup \{A : -G \in u_{q_{priv}}\}$

Naturally $\sqcup_{publ}$ and $\sqcup_{priv}$ can define operator $\sqcup$. That is $u_p \sqcup u_q = (u_p \sqcup_{publ} u_q) \sqcup_{priv} u_q$

In the second category, the composition of units takes place considering only the overriding union of their clauses. This approach corresponds to non-monotonic composition of clauses.

The operator $\oplus_{publ}$ is defined as:

$u_p \oplus_{publ} u_q = u_z$
$u_{z_{publ}} = \{A : -G \in u_{p_{publ}}\} \cup \{A : -G \in u_{q_{publ}} : pred(A) \notin \pi(u_{p_{publ}})\}$
$u_{z_{priv}} = \{A : -G \in u_{p_{priv}}\}$

The operator $\oplus_{priv}$ is defined as:

$u_p \oplus_{priv} u_q = u_z$
$u_{z_{publ}} = \{A : -G \in u_{p_{publ}}\}$
$u_{z_{priv}} = \{A : -G \in u_{p_{priv}}\} \cup \{A : -G \in u_{q_{priv}} : pred(A) \notin \pi(u_{p_{priv}})\}$

The operators $\oplus_{publ}$ and $\oplus_{priv}$ can be combined to produce the operator $\oplus$. That is $u_p \oplus u_q = (u_p \oplus_{publ} u_q) \oplus_{priv} u_q$.

Combining the above operators, we can define another two operators. These operators express a mixed, monotonic and non-monotonic, composition of units.

The operator $\diamond_{publ}$ is defined as $u_p \diamond_{publ} u_q = (u_p \sqcup_{priv} u_q) \oplus_{publ} u_q$. That is:

$u_p \diamond_{publ} u_q = u_z$
$u_{z_{publ}} = \{A : -G \in u_{p_{publ}}\} \cup \{A : -G \in u_{q_{publ}} : pred(A) \notin \pi(u_{p_{publ}})\}$

$u_{z_{priv}} = \{A:-G \in u_{p_{priv}}\} \cup \{A:-G \in u_{q_{priv}}\}$
The operator $\diamond_{priv}$ is defined as $u_p \diamond_{priv} u_q = (u_p \sqcup_{publ} u_q) \oplus_{priv} u_q$. That is:
$u_p \diamond_{priv} u_q = u_z$
$u_{z_{publ}} = \{A:-G \in u_{p_{publ}}\} \cup \{A:-G \in u_{q_{publ}}\}$
$u_{z_{priv}} = \{A:-G \in u_{p_{priv}}\} \cup \{A:-G \in u_{q_{priv}} : pred(A) \notin \pi(u_{p_{priv}})\}$

To illustrate the above operators giving an example of their functionality, we shall use concepts from deductive databases. Consider that the private part of a unit implements the EDB relations and the public part the IDBs. Moreover, consider that the IDBs implement a secure interface to the EDBs, e.g. a projection to some attributes. Then, it may be convenient to merge such databases in one of the following ways:

1. Take the union of the EDBs, i.e. the union of the actual data, and the overriding union of the IDBs, i.e. the interfaces. This means that, in case of conflict, we desire to keep only the security interfaces of the first database. This is implemented by $\diamond_{publ}$.

2. Consider that the database administrator is interested only in the actual data (EDBs) of the second database. In this case, $\sqcup_{priv}$ can be used.

3. Consider that only the security interface of the second database is of interest, and not the actual data, but no definition conflict with the one of the first database is desired. In this case, $\oplus_{publ}$ can be used.

What is more is that, another area of application of these operators, which is illustrated in the next section in more detail, is the object oriented paradigm. More precisely, the inheritance property of object oriented systems —in both of its forms, monotonic and non-monotonic— can be expressed using the proposed operators. Forms of monotonic inheritance can be expressed using the $\sqcup$ family of operators. Forms of non-monotonic inheritance can be expressed using the $\oplus$ and $\diamond$ families of operators. The operators express not only the common forms of monotonic and non-monotonic inheritance but even innovative, more interesting approaches. In addition, the operators can be used to express some forms of inheritance of C++. More precisely, the inheritance of public methods of C++ can be simulated by the $\sqcup_{publ}$ operator and the inheritance of the protected methods of C++ by the $\sqcup_{priv}$ operator.

4.2 Inheritance and Information Hiding

The operators defined so far can be exploited in supporting different structuring policies for units. Hierarchical composition of units can be achieved. Hierarchies of units exhibit object oriented features such as code reuse, code redefinition and information hiding that are achieved by inheritance and encapsulation in the object oriented framework.

A program in h-log is a set of units that are not related with each other but they can exchange messages. We can extend the alphabet of h-log with a composition operator c and with an is-a relation $\prec$. The relation $\prec$ defines a tree over the elements of a set of units U. The operator c can be a primitive or a composed one. When $u_1 \prec u_2$ then u_1 inherits information from u_2. The information that is inherited depends on the kind of the composition. In order to give the semantics of the inheritance, we assume that an h-log program $P(U')$ is produced from an extended h-log program $P(U)$ in the following way:

If $u_r \in U$ and u_r is the root of the tree then there is a unit u'_r in U' that is identical to u_r.

If $u \in U$ and the hierarchy of units above u is $H(u) = u \prec u_1 \prec u_2 ... \prec u_r$ then there is a unit $u' \in U'$ such that $u' = (u\ c\ (u_1\ c\ (...\ c\ (u_r)))$ —note that, for all operators, the associativity law holds—. According to the desired functionality of inheritance, the appropriate operator can be used to instantiate c.

The semantics of the extended h-log program $P(U)$ is given by the minimal model of the corresponding h-log program $P(U')$, that is $M_{P(U')} = lfp(\mathrm{T}_{P(U')}) = \mathrm{T}_{P(U')} \uparrow \omega$.

The operational semantics for a hierarchy of units that is built by the composition operator $c = \sqcup_{publ}$ is given below:

$(H1_c)$ $\quad \dfrac{}{H(u) \vdash_\epsilon \Box}$

$(H2_c)$ $\quad \dfrac{u \sqcup_{publ} u_1 \sqcup_{publ} ... \sqcup_{publ} u_r \vdash_\sigma G\theta}{H(u) \vdash_{\theta\sigma} g} \quad \left(\begin{array}{c} h\text{:-}G \in u' \\ \theta = mgu(g,h) \\ H(u) = u \prec u_1 \prec u_2 ... \prec u_r \end{array} \right)$

$(H3_c)$ $\quad \dfrac{H(u) \vdash_\theta G_1 \ \wedge\ H(u) \vdash_\sigma G_2\theta}{H(u) \vdash_{\theta\sigma} G_1, G_2}$

$(H4_c)$ $\quad \dfrac{H(\hat{u}) \vdash_\theta G \wedge pred(G) \in \pi(\hat{u}'_{publ})}{H(u) \vdash_\theta \hat{u'} > G} \quad (\hat{u'} = \hat{u} \sqcup_{publ} \hat{u_1} \sqcup_{publ} \dots \sqcup_{publ} u_r)$

The operational semantics for the other operators are given in a similar way.

5 Discussion and Conclusions

Unlike the traditional notion of units or modules, we propose a novel syntactic and semantic approach with applicability in deductive databases. The units in our framework are composed of two kind of clauses that is public and private clauses. Clean semantics are given based on the notions of S-interpretations and T-interpretations instead of the classical logic interpretation. This is natural because classical logic lacks the notion of modularity. We also contribute defining new composition operators that apply to units with public and private clauses. Our framework is a generalization of several modular logic programming approaches. It also models several object oriented features such as objects, classes, message passing, public and private methods, inheritance and encapsulation.

The basic theory of units with public and private clauses can be extended. More precisely, we can introduce non-uniform composition of units. This means that different composition operators can take place in the construction of an is-a hierarchy. Moreover, a unit can be combined with more than one immediate ancestors. In this case, the hierarchy is a graph and we achieve multiple inheritance. So far we have examined the case where the private clauses of a unit are invisible to other units. If we want some other unit to have access to these clauses we have to introduce the notion of friendly units. Additionally, we can further extend our basic theory with parameterized units.

References

[ABD+89] M. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *First International DOOD Conference*, pages 223–240, 1989.

[AKM97] F. Afrati, I. Karali, and T. Mitakos. Inheritance in object oriented datalog: A modular logic programming approach. Technical report, National Technical University of Athens, 1997.

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

[BJ94] M. Bugliesi and H. Jamil. A logic for encapsulation in object oriented languages. In *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming*, pages 215–229, 1994.

[BLM92] A. Brogi, E. Lamma, and P. Mello. Compositional model–theoretic semantics for logic programs. *New Generation Computing*, 11:1–21, 1992.

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

[BMPT94] A. Brogi, P. Mancrella, D. Pedreschi, and F. Turini. Modular logic programming. *ACM Transactions on Programming Languages and Systems*, 16:1361–1398, 1994.

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

[KA89] P. Kanellakis and S. Abiteboul. A logical database query language with object identity and strong typing. In *6th International Conference in Logic Programming*, pages 675–692, 1989.

[KLW94] M. Kifer, G. Lausen, and J. Wu. Logical foundations for object-oriented and frame-based languages. Technical report tr - 93/06, SUNY at Stony Brook, Department of Computer Science, 1994.

[Llo87] John W. Lloyd. *Foundations of Logic Programming*. Springer–Verlag, 1987.

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

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

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