A Logic for Encapsulation in Object Oriented Languages

(Extended Abstract)

Michele Bugliesi[1]
Dip. di Matematica Pura ed Applicata, Univ. di Padova
Via Belzoni 7, I-35131 Padova, Italy
e-mail: michele@goedel.math.unipd.it

Hasan M. Jamil[2]
Department of Computer Science, Concordia University
Montréal, Québec, Canada H3G 1M8
e-mail: jamil@cs.concordia.ca

Abstract. We present a logic language with encapsulation based on an object-oriented data model. We give a formal account of the semantics of this language by defining a proof theory, a model theory and a fixed point theory. We also show that the different characterizations of the semantics are equivalent. We then study the logical foundations of encapsulation by introducing a mapping from our language to a corresponding modal language, and we show that the proof theory of the language is sound and complete with respect to the entailment relation of the corresponding modal framework. The proposed semantics represents – to our knowledge – the first attempt to capture a truly logical semantics of encapsulation in deductive object oriented languages.

1 Introduction

There have been several attempts [9, 10, 1, 5, 11, 2, 16] at combining logic programming with object-oriented concepts like object identity, complex objects, methods, encapsulation, signatures, inheritance, etc. in a clean mathematical framework. Most of the approaches either fall short of capturing the essential set of properties of object-orientation or sacrifice declarativity.

The language F-logic proposed by Kifer et al. [15] is a very nice declarative formalism for OODBs and is regarded as one of the best developed proposals so far. The work on F-Logic has provided foundations for a whole suite of research in the field of deductive object-oriented databases. Among others, a fundamental merit of this proposal is that it has a very solid logical foundation with "a sound and complete proof theory" [15].

There are, however, several aspects of the OO programming paradigm that are left outside the logic of F-logic and, instead, are treated as purely *meta*-logical

[1] Partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of C.N.R. grant n. 93.00898.PF69.

[2] Partially supported by grants from the Canadian Commonwealth Scholarship and Fellowship Plan and the University of Dhaka, Bangladesh. The author is on leave from the University of Dhaka, Bangladesh.

features. Encapsulation of properties and inheritance of methods are representative of these missing features. Encapsulation is a major concept in the suite of notions comprising the OO paradigm. It is an essential software-engineering technique which helps define objects with interfaces that restrict the access to the state of an object only through the *public* methods of the object's interface. As such, it frees the users from the burden of knowing the internal structure and state of the objects by hiding all unnecessary details. This mechanism is referred to as *structural encapsulation* as opposed to the notion of *behavioral encapsulation* that refers to the concept of hiding the implementation detail of the methods of objects.

The aim of this paper is to develop a declarative model of encapsulation on top of the data model proposed for F-Logic. Thus the language we employ is similar to that of F-logic but with important differences. In fact, as our present goal is to study only the issue of encapsulation, we opt for a by far simpler language[3]. By restricting the language, we have been able to obtain results that give greater insights into the logical nature of encapsulation, and to build a foundation for studying more complex and expressive languages. We adopt a "complex object" data model in the line of [15] and we represent objects as first-order terms. However, in contrast to [15], we distinguish between objects and their types and we do not allow type variables, and consequently we restrict the schema browsing capability of the language.

We first present the syntax of our language and then define a proof theory, a model theory and a fixed point theory for this language. We also show that the three theories for our language coincide. In contrast to F-logic, we treat encapsulation as a linguistic feature, and its semantics is built into the proof theory as well as into the model theory of the language.

We then move on to study the logical foundations of our language. We discuss a mapping from our language to a corresponding modal language (a subset of S4 modal logic) and show that our proof theory is sound and complete with respect to the entailment relation of the corresponding modal framework. The choice of a modal setting is motivated by the fact that it sheds light on the logical nature of encapsulation and, more precisely, of the visibility rules underlying the functionalities of this mechanism. In fact, when interpreted in the modal framework, the notions of *private* and *public* properties affecting the behaviour of encapsulation have a natural counterpart in the modal notions of, respectively, *possibly true* and *necessarily true* properties.

We organize the rest of our paper as follows. In section 2 we present the syntax and a formal definition of the language. In section 3 we discuss the informal semantics by means of simple examples. In sections 4 and 5 we introduce the proof theory, the model theory and we show their equivalence. Then, in section 6 we study the modal characterization our language and prove the soundness and completeness of our proof theory. We discuss related research in section 7 and finally conclude in section 8 discussing topics of future research.

[3] We do not consider the issue of inheritance in this paper. For a discussion on inheritance in our language, interested readers are referred to [6].

2 Overview of the Language

The syntax of the language is defined along the guidelines of F-Logic. We fix ahead the alphabet: it is a quadruple $A = \langle V, F, T, \Pi \rangle$ where V is a set of variables, F is the set of constant and function symbols, T is the set of *type designators* and Π is the set of predicate symbols. The components of the alphabet are assumed to be pairwise disjoint. We distinguish three sets of terms: (i) *o-terms*, terms built over $F \cup V$, (ii) *p-terms*, terms of the form $p(o_1, \ldots .o_n)$ where $p/n \in \Pi$ and the o_is are o-terms, and (iii) *t-terms*, (ground) terms in the set T. The set of *ground* o-terms is denoted by O, and the variables in V range over this set.

We use the term type rather than class throughout, because there is no inheritance between objects and types. Also there we do not associate any notion of *class extension* with types: the purpose of typing an object is simply to define the signatures of the methods that can be applied to that object. This also explains the reason for not having any type data by disallowing variables to range over the set of t-terms.

Atomic and Complex Formulas. *Molecules*, *type expressions* and *signature expressions* constitute the set of atomic formulas of the language. Their structure is defined below.

Molecules have the form $o[P]$, where o is an o-term and P is a p-term – called respectively the molecule's object identity (*oid*) and p-term. The intended meaning of a molecule in our language is similar to F-logic: $o[P]$ states that property P holds at object o.

Type expressions are of the form $o : \tau$ where o is an o-term and τ is a t-term. The intention of a type expression is to express the typing information for the o-terms: $o : \tau$ states that o is an object of type τ.

Signature expressions have the form $\tau\{p_1 : \Delta_1, \ldots, p_n : \Delta_n\}$ where τ is a t-term and the p_is are predicate symbols from Π. The Δ_is are designators that qualify the status of the associated properties. For every p_i, the designator $\Delta_i \in \{priv, pub\}$ states whether the property is *private* or *public*. The intention of a signature expression at a given type, is to restrict the access to the properties of the object belonging to that type. The private properties of an object are visible (and accessible) only to the object itself, while the public properties are visible to all the objects. Since the properties of an object define everything about the state of the object (attributes and methods) this idea captures precisely the idea of *structural encapsulation* developed and pursued in the OO literature.

Complex formulas are built out of simpler formulas by connecting them through the usual logical connectives. However, our programs will be built over a considerably simpler class of complex formulas, namely the class of definite Horn clauses. The structure of a program clause is defined precisely in the next subsection.

Programs and Queries. The purpose of a program is twofold: firstly, it specifies the type of each object and the signatures for each type and, secondly, it defines the (implementation of) the methods/attributes attached to each object. Thus, every program can be viewed as consisting of two parts, each one dedicated to the specification of one of these aspects.

Definition 1 (PROGRAMS). A program $\boldsymbol{\Pi}$ is a pair $\boldsymbol{\Gamma}:\mathbf{P}$ where:

- $\boldsymbol{\Gamma}$: the *schema declaration*, a (possibly empty) set of *unit* clauses whose head is a type or signature expression.
- $\mathbf{P}$: the *method declaration*, a (possibly empty) set of clauses whose head is a molecule and whose body literals are either type expressions or molecules.

As it is customary in logic programming, we assume that all the variables occurring free in a clause of $\boldsymbol{\Pi}$ be universally quantified. Thus, every clause of $\boldsymbol{\Pi}$ will be of the form $\forall x.(A \leftarrow \exists y.G)$ where A is an atomic formula, y are the variables local to G, and G, in its turn, is a conjunction of atomic formulas (which is empty when A is a type or signature expression). In addition to this, we also require that our programs be well typed, in the sense of the following definition.

Definition 2 (WELL-TYPED PROGRAMS). A program $\boldsymbol{\Gamma}:\mathbf{P}$ is well typed if and only if the following conditions hold:

- every oid is assigned a single type by $\boldsymbol{\Gamma}$;
- every type assigned by $\boldsymbol{\Gamma}$ is closed under instantiation, i.e. if $\forall x.\ o(x) : \tau \in \boldsymbol{\Gamma}$ then, for every instance $o(t)$ of $o(x)$, $o(t) : \tau' \in \boldsymbol{\Gamma} \implies \tau' = \tau$.
- every variable oid occurring in a clause of $\mathbf{P}$ is typed by a type expression occurring in the body of the clause. □

Note that, similarly to the language proposed by Dobbie and Topor in [12], we disallow dynamic typing and dynamic definitions of signatures by requiring that signatures and types be defined by unit clauses. We also require that there be no occurrence of a signature expression as a body literal in the clauses of the component $\mathbf{P}$. Following the same line of reasoning, we disallow the possibility of having signature expressions as queries. A well typed goal G is a conjunction of atomic formulas where the conjuncts are either type expressions or molecules and such that every variable oid occurring in a molecule of G is typed by a type expression occurring in G.

3 Programming Examples

In this section, we describe the informal semantics of the language by means of two simple examples. The first is a classical example of object-oriented programming which we borrow and adapt from [14, 19]. For reasons of notational convenience, we adopt the conventional syntax for clauses and we use logical variables in place of quantified variables.

Example 1. The following program fragment defines the type *point*, with its signatures, methods and values. The structure of each point is defined by the private property $xy/2$ whose arguments denote the coordinates of the point. Furthermore, each point has two public methods: *dist/1* gives the distance of the point from the origin; *closer/1* takes another point as parameter and checks whether the point passed as parameter is closer to the origin than the current point.

$$\boldsymbol{\Gamma} ::= \left| \begin{array}{ll} (1) & point\{dist/1 : pub,\ closer/1 : pub,\ xy/2 : priv\} \\ (2) & origin : point. \\ (3) & pt(X,Y) : point. \end{array} \right.$$

$$\mathbf{P} ::= \left| \begin{array}{ll} (4) & origin[\,xy(0,0)\,]. \\ (5) & pt(X,Y)[\,xy(X,Y)\,]. \\ (6) & P[\,dist(D)\,] \leftarrow P : point, P[\,xy(X,Y)\,],\ D\ is\ sqrt(X^2+Y^2). \\ (7) & P[\,closer(Q)\,] \leftarrow P : point, Q : point,\ Q[\,dist(DQ)\,], \\ & \qquad\qquad\qquad\qquad P[\,dist(DP)\,],\ DP < DQ. \end{array} \right.$$

Note the use of the complex term $pt(X,Y)$ as an object identity in clause (5): this will allow us to create new identities by simply instantiating the arguments of *pt*. Thus, the query "$P : point, P[closer(pt(1,2))]$" will return all the objects with points which are closer to the origin than the point $xy(1,2)$.

It is important to remark that the parameters do *not* disclose the internal representation of the objects of type point: they are just a device for creating new object identity. Counterwise, the structure of a point is defined by the property $xy/2$ which is private and thus encapsulated. This has also the effect of encapsulating the implementation of the method *dist/1* which is the only method that has access to the internal representation. We could, for instance, change this representation and use polar coordinates instead of cartesian coordinates: we would then need to modify the definition of *dist/2*, but users of this method (the method *closer/1*, in this case) would not be affected by the change.

Example 2. As a further example, we put forward a program which will be used to illustrate the properties of the proof theory and of the model theory of the language.

$$\begin{array}{llll} (1) & \tau_1\{p/1 : pub,\ q/1 : priv\} & (5) & X[q(a)] \leftarrow X : \tau_1. \\ (2) & \tau_2\{r/1, s/1 : pub\} & (6) & X[p(Y)] \leftarrow X : \tau_1, X[q(Y)]. \\ (3) & o_1 : \tau_1. & (7) & X[r(Z)] \leftarrow X : \tau_2, Y : \tau_1, Y[q(Z)]. \\ (4) & o_2 : \tau_2. & (8) & X[s(Z)] \leftarrow X : \tau_2, Y : \tau_1, Y[p(Z)]. \end{array}$$

Here we would like to have no answer to the query $o_1[q(X)]$ since $q/1$ is private for τ_1 and o_1 is of type τ_1. Counterwise, the query $o_1[p(X)]$ should produce the answer $\{X/a\}$ since $p/1$ is public for τ_1 and $o_1 : \tau_1$ is allowed to access its own property $q/1$. Following the same line of reasoning, we have no answer for $o_2[r(X)]$: in fact, after head unification with clause (7), we are left with the evaluation of the goal $Y : \tau_1, Y[q(a)]$. Now, substituting o_1 for Y, we have a failure for $o_1[q(a)]$ because $q/1$ is private for o_1.

Counterwise, we will have the answer $\{X/a\}$ for $o_2[s(X)]$, for in this case the evaluation of $o_1[q(a)]$ takes place within clause (6), i.e. when the *context of evaluation* is o_1 which, again, is allowed to access its own private property $q/1$.

The proof theory of the language, which is discussed in the following section, captures this behavior by explicitly introducing in the proof rules the notion of evaluation context for a goal.

4 Proof Theory

We define the relation of provability in a sequent-style notation. Provability is defined in two steps: we first define the relation of provability of a closed (existentially quantified or ground) goal starting from a given object. Then we define provability for a closed goal in a program (independently of any object).

Given the universe O of all the ground o-terms, we define $O_\perp$ as the set $O_\perp = O \cup \{o_\perp\}$ where $o_\perp$ is a distinguished oid. It is important to remark that $o_\perp$ is *not* part of the program universe and, accordingly, the variables of the program range over O and not over $O_\perp$. The intention of $o_\perp$ is to model a kind of *null* object, which has no type and, consequently, no (public) property. In practice, the null object can be understood as the *system* object, i.e. the object where from all the computations start off in a program.

Let G be a (well-typed) closed goal, $\boldsymbol{\Pi} = \boldsymbol{\Gamma} : \mathbf{P}$ be a (well-typed) program, $\overline{o} \in O_\perp$ and $o \in O$ be two object identities, and τ be a type designator. Also, let $\widehat{P}$ denote the predicate symbol of the p-term P.

$$(\text{TYPE}) \qquad \frac{o : \tau \in [\boldsymbol{\Gamma}]}{\boldsymbol{\Pi}, \overline{o} \vdash o : \tau}$$

$$(\text{NON-LOCAL}) \qquad \frac{o : \tau, \tau\{\widehat{P} : pub\} \in [\boldsymbol{\Gamma}] \qquad \boldsymbol{\Pi}, o \vdash G}{\boldsymbol{\Pi}, \overline{o} \vdash o[P]} \qquad (o[P] \leftarrow G \in [\mathbf{P}]\,)$$

$$(\text{LOCAL}) \qquad \frac{\boldsymbol{\Pi}, o \vdash G}{\boldsymbol{\Pi}, o \vdash o[P]} \qquad (o[P] \leftarrow G \in [\mathbf{P}]\,)$$

$$(\text{AND}) \qquad \frac{\boldsymbol{\Pi}, \overline{o} \vdash G_1 \qquad \boldsymbol{\Pi}, \overline{o} \vdash G_2}{\boldsymbol{\Pi}, \overline{o} \vdash G_1 \wedge G_2}$$

$$(\text{INSTANCE}) \qquad \frac{\boldsymbol{\Pi}, \overline{o} \vdash [x/o]G}{\boldsymbol{\Pi}, \overline{o} \vdash \exists x.G}$$

We have used the notation $[x/o]G$ to stand for the goal obtained from G by replacing each free occurrence of x in G with the o-term $o \in O$. With $[\mathbf{P}]$ and $[\boldsymbol{\Gamma}]$ we have denoted the universal closure of $\mathbf{P}$ and $\boldsymbol{\Gamma}$, i.e. the minimal sets satisfying the condition[4] that $\forall x.D \in \boldsymbol{\Pi} \implies [x/o]D \in [\boldsymbol{\Pi}]$ for every $o \in O$.

[4] Note that the universal closure is different than ground closure. Let * denote the ground closure operator. Then, while $[\boldsymbol{\Gamma}] = \boldsymbol{\Gamma}^\star$, the same is not true of $[\mathbf{P}]$ and $\mathbf{P}^\star$. In fact, $[\mathbf{P}]$ contains clauses of the form $o[P] \leftarrow \exists x.G$ whereas $\mathbf{P}^\star$ does not.

The object $\overline{o}$ in $\boldsymbol{\Pi}, \overline{o} \vdash G$ represents the context of evaluation of the goal G. Now, according to (NON-LOCAL), if we are to query a property of an object different than the current one, we require that property to be *public* for that object. Counterwise, if we query a local property, then we are allowed to access it independently of its status, i.e. whether it is private or public. Similarly, the proof of a type expression is independent of the context of evaluation.

It is important to remark that the (INSTANCE) rule doe not produce a *witness* substitution, as it is customary in logic programming; instead, it attempts to construct a proof through a potentially infinite non-deterministic or-branching where each branch corresponds to a particular *ground* witness. As a consequence, the application of the three rules (LOCAL), (NON-LOCAL) and (TYPE) will take place only when the corresponding goal is ground.

It should be noted, however, that the hypothesis that $\boldsymbol{\Pi}$ be well typed (in the sense of definition 2) guarantees that deferring instantiation using unification, as it is customary in logic programming, would not break the structure of a proof: in fact, since typing is closed under instantiation, any decision made on the type of a variable – and consequently on whether we are allowed to access a property of a variable oid – will be consistent with the decision we would make on any instance of that variable.

We are now ready to complete the definition of provability for a goal. We anticipated the use of the null object to define provability in a program. We now make this idea precise with the following definition.

$$(\textsc{main}) \qquad \frac{\boldsymbol{\Pi}, o_\perp \vdash G}{\boldsymbol{\Pi} \vdash G}$$

Note that, since the null object is different from any other object $o \in O$, it follows from the definition of the relation $\boldsymbol{\Pi} \vdash G$ that a goal G is provable in the program only if each of the predicate symbols appearing in G is public for at least one of the objects in O (for every G, the proof of $\boldsymbol{\Pi}, o_\perp \vdash G$ uses (NON-LOCAL) as it's first rule).

5 Model Theory and Fixed Point Theory

Programs in the language are interpreted in structures of the form $\mathbf{H} = \langle \mathbf{H}_m, \mathbf{H}_\tau \rangle$ built over the alphabet A. The two components are intended to interpret the corresponding components of a program. $\mathbf{H}_m$ is a set of ground molecules. On the other hand, $\mathbf{H}_\tau$ is the union of two sets: a set of ground type expressions, plus a set of terms of the form $p/a \,@\, \tau$ ($p/a \,\sharp\, \tau$) interpreting the signature expressions. For every predicate p/a and type τ, $p/a \,@\, \tau \in \mathbf{H}_\tau$ means that p/a in all objects of type τ is visible (*public*) to all objects in the universe. Counterwise, $p/a \,\sharp\, \tau \in \mathbf{H}_\tau$ states that p/a is *defined* (*private*) for all the objects of type τ.

Interpretations. Interpretations are built over structures by letting the objects of a given type inherit the visible part of their type's signature. Accordingly, the type component of an interpretation will consist of terms of the form $p/a\,@\,\tau$, $p/a\,\sharp\,\tau$, and $p/a\,@\,o$ with $o \in O$. Given a structure $\mathbf{H} = \langle \mathbf{H}_m, \mathbf{H}_\tau \rangle$, we say that $\mathbf{I} = \langle \mathbf{I}_m, \mathbf{I}_\tau \rangle$ is the *interpretation corresponding to* $\mathbf{H}$ if and only if $\mathbf{I}_m$ and $\mathbf{I}_\tau$ are the minimal sets such that (i) $\mathbf{H}_m \subseteq \mathbf{I}_m$ and $\mathbf{H}_\tau \subseteq \mathbf{I}_\tau$, and (ii) for every predicate symbol p/a, $\;p/a\,@\,\tau \in \mathbf{I}_\tau \;\wedge\; o : \tau \in \mathbf{I}_\tau \implies p/a\,@\,o \in \mathbf{I}_\tau$.

Satisfaction. Satisfaction is defined for closed formulas. Let $\mathbf{I} = \langle \mathbf{I}_m, \mathbf{I}_\tau \rangle$ be an interpretation and let $\widehat{P}$ denote the predicate symbol of any given p-term P. The satisfaction of a closed goal in $\mathbf{I}$ is defined as follows:

$$\begin{array}{ll}(1) & \mathbf{I} \models o : \tau \iff o : \tau \in \mathbf{I}_\tau \\ (2) & \mathbf{I} \models o[P] \iff o[P] \in \mathbf{I}_m \;\text{ and }\; \widehat{P}\,@\,o \in \mathbf{I}_\tau\end{array}$$

The first condition should be obvious. The second condition says that in order for the molecule $o[P]$ to be satisfied, the predicate symbol of P must be exported (made visible) in $\mathbf{I}_\tau$ by (the type of) o. Satisfaction of existentially quantified and conjunctive goal formulas, as well as of type and signature clauses are defined as expected. Satisfaction of method clauses is defined in the line of (2) as

$$(3)\;\; \mathbf{I} \models o[P] \leftarrow G \iff \mathbf{I}^o \models G \implies \mathbf{I}^o \models o[P]$$

where $\mathbf{I}^o$ is obtained from $\mathbf{I}$ by taking $\mathbf{I}^o_m = \mathbf{I}_m$ and defining $\mathbf{I}^o_\tau$ as follows:

$$\mathbf{I}^o_\tau = \mathbf{I}_\tau \cup \{p/a\,@\,o \mid \exists\tau.\; p/a\,\sharp\,\tau \in \mathbf{I}_\tau \text{ and } o : \tau \in \mathbf{I}_\tau\}$$

In other words, if o is of type τ, then we test G for satisfiability in a structure where all the predicates defined for τ are made visible for o. Whenever G is true in this structure, so must be $o[P]$.

Models. An interpretation $\mathbf{I} = \langle \mathbf{I}_m, \mathbf{I}_\tau \rangle$ is a model for a program $\boldsymbol{\Gamma}{:}\mathbf{P}$ iff:

- $\mathbf{I} \models \mathbf{P}$, i.e. every closed instance of all the clauses of $\mathbf{P}$ is satisfied in $\mathbf{I}$;
- $\mathbf{I} \models \boldsymbol{\Gamma}$, i.e. every ground instance of a type expression of $\boldsymbol{\Gamma}$ is contained in the type component of $\mathbf{I}$, and (i) $\tau\{p/a : \mathit{priv}\} \in \boldsymbol{\Gamma} \implies p/a\,\sharp\,\tau \in \mathbf{I}_\tau$ and (ii) $\tau\{p/a : \mathit{pub}\} \in \boldsymbol{\Gamma} \implies p/a\,@\,\tau \in \mathbf{I}_\tau$.

Interpretations can be ordered in the natural way using the ordering on the components (set-inclusion). Namely, if $\mathbf{I}$ and $\mathbf{J}$ are interpretations, then $\mathbf{I} \sqsubseteq \mathbf{J}$ iff $\mathbf{I}_m \subseteq \mathbf{J}_m$ and $\mathbf{I}_\tau \subseteq \mathbf{J}_\tau$.

It is also easy to see that the intersection of two models is again a model, that every program has at least one model and, consequently, that every program has a *least* model (under $\sqsubseteq$), Clearly, the type component of the least model $\mathbf{M}$ of a program will satisfy the condition that $\tau\{p/a : \mathit{pub}\} \in \boldsymbol{\Gamma}$ iff $p/a\,@\,\tau \in \mathbf{M}_\tau$ (and similarly for defined predicates). The least model can be taken as the semantics of the program. Furthermore, it can be computed following the standard recipe, as we show next.

Fixed Points. We define the operator $\mathbf{T}_{\boldsymbol{\Pi}}$ from interpretations to interpretations as follows: $\mathbf{T}_{\boldsymbol{\Pi}}(\langle \mathbf{I}_m, \mathbf{I}_\tau \rangle) = \langle T_{\mathbf{P}}(\mathbf{I}_m, \mathbf{I}_\tau), \mathbf{I}_\tau \rangle$.

$T_{\mathbf{P}}$ constitutes the counterpart of the immediate-consequence operator used in logic programming: $T_{\mathbf{P}}(\mathbf{I}) = \{o[P] \mid o[P] \leftarrow G \in [\mathbf{P}] \text{ and } \mathbf{I}^o \models G\}$ where $\mathbf{I}^o$ is defined in terms of $\mathbf{I}$ as in the definition of satisfaction of a clause.

Lemma 3. *Let $\boldsymbol{\Gamma} : \mathbf{P}$ be a program, and let $\mathbf{I}$ be an interpretation. Then $\mathbf{I} \models \mathbf{P}$ iff $T_{\mathbf{P}}(\mathbf{I}) \subseteq \mathbf{I}_m$.* $\square$

Using the result of this lemma, we can show that the least model of every program can be computed as the interpretation obtained by iterating the corresponding $\mathbf{T}_{\boldsymbol{\Pi}}$ operator up to ω.

Theorem 4. *Let $\boldsymbol{\Pi} = \boldsymbol{\Gamma} : \mathbf{P}$ be a program, and let $\mathbf{M}_{\boldsymbol{\Pi}}$ be the least model of the program. Then, given $\langle \emptyset, \mathbf{M}_\tau \rangle$, the least (under $\sqsubseteq$) interpretation that satisfies $\boldsymbol{\Gamma}$, we have that $\mathbf{M}_{\boldsymbol{\Pi}} = \mathbf{T}^{\omega}_{\boldsymbol{\Pi}}(\langle \emptyset, \mathbf{M}_\tau \rangle)$.* $\square$

The fixed point construction of the least model has the same importance as in logic programming. It allows us to prove the equivalence between the model theory and the proof theory. As for lemma 3, the proof is similar to the standard proof of the corresponding result in logic programming [3].

Theorem 5. *Let $\boldsymbol{\Pi}$ be a program and let $\mathbf{M}_{\boldsymbol{\Pi}}$ be the least model of the program. Then for every closed goal formula G, $\boldsymbol{\Pi} \vdash G$ iff $\mathbf{M}_{\boldsymbol{\Pi}} \models G$.* $\square$

We exemplify the last result by revisiting the program of example 2 (section 3).

(1)	$\tau_1\{p/1 : pub,\ q/1 : priv\}$	(5)	$X[q(a)] \leftarrow X : \tau_1.$
(2)	$\tau_2\{r/1, s/1 : pub\}$	(6)	$X[p(Y)] \leftarrow X : \tau_1, X[q(Y)].$
(3)	$o_1 : \tau_1.$	(7)	$X[r(Z)] \leftarrow X : \tau_2, Y : \tau_1, Y[q(Z)].$
(4)	$o_2 : \tau_2.$	(8)	$X[s(Z)] \leftarrow X : \tau_2, Y : \tau_1, Y[p(Z)].$

The model $\mathbf{M}$ we want to compute for this program has the following structure:

$$\mathbf{M}_m = \{o_1[q(a)],\ o_1[p(a)],\ o_2[s(a)]\}$$
$$\mathbf{M}_\tau = \left\{ \begin{array}{l} o_1 : \tau_1,\ o_2 : \tau_2,\ p/1 @ o_1,\ r/1 @ o_2,\ s/1 @ o_2, \\ p/1 @ \tau_1,\ r/1 @ \tau_2,\ s/1 @ \tau_2, p/1 \sharp \tau_1, q/1 \sharp \tau_1\ r/1 \sharp \tau_2,\ s/1 \sharp \tau_2 \end{array} \right\}$$

It can be easily verified that $\mathbf{M}$ is a model and that it is minimal. It is also interesting to verify that $\mathbf{M}$ captures the expected behavior of encapsulation. In fact:

(i) $o_1[q(a)] \in \mathbf{M}_m$ but $\mathbf{M} \not\models o_1[q(a)]$ since $q/1 @ o_1 \notin \mathbf{M}_\tau$. On the contrary, $\mathbf{M}$ satisfies clause (5). In fact, consider substituting o_1 for X in (5). Then $\mathbf{M} \models o_1[q(a)] \leftarrow o_1 : \tau_1$ being $o_1 : \tau_1 \in \mathbf{M}_\tau$ and $\mathbf{M}^{o_1} \models o_1[q(a)]$ since, by definition, $\mathbf{M}^{o_1}_\tau = \mathbf{M}_\tau \cup \{q/1 @ o_1\}$.

(ii) $o_2[r(a)] \notin \mathbf{M}_m$ and yet $\mathbf{M}$ satisfies clause (7). The argument is similar to that used in (i). Consider the ground instance of (7) obtained by substituting

o_2 for X, o_1 for Y and a for Z. Then, following the definition of satisfaction, we have that $\mathbf{M} \models o_2[r(a)] \leftarrow o_2 : \tau_2, o_1 : \tau_1, o_1[q(a)]$ holds if and only if

$$\mathbf{M}^{o_2} \models o_2 : \tau_2, o_1 : \tau_1, o_1[q(a)] \implies \mathbf{M}^{o_2} \models o_2[r(a)]$$

where $\mathbf{M}^{o_2} = \mathbf{M}$ in this case since $r/1$ is already exported by τ_2 in $\mathbf{M}$. But the last implication holds trivially since $\mathbf{M} \not\models o_1[q(a)]$.

(iii) $\mathbf{M} \models o_2[s(a)]$ since in this case the access to the clause which defines the private predicate $q/1$ in τ_1, is performed indirectly via $p/1$ which is public for τ_1.

What follows from (i), (ii) and (iii) is that the definition of $q/1$ is *structurally encapsulated* in o_1. In particular, (i) says that the property $q(a)$ is part of the structure of o_1 (being $o_1[q(a)] \in \mathbf{M}_m$), but access to that part of the structure cannot be performed directly as shown by the fact that $\mathbf{M} \not\models o_1[q(a)]$. Thus the only way to access the definition of $q/1$ is through the visible predicate $p/1$, defined by clause (6).

6 Logical Foundations

In this section we study the logical foundations of our language. One of the standard ways that this can be accomplished is to define a mapping from programs in our language to corresponding (and equivalent) programs whose operational behavior can be stated in terms of the inference rules of a known logical system. It is indeed not difficult to see that we could obtain a logical characterization of the proof theory in terms of provability in classical logic. In fact, there is an easy way to model the workings of encapsulation in terms of predicate renaming. However, we argue that an encoding based on predicate renaming has several unwanted consequences: most importantly it does not help understand and clarify the logical meaning of encapsulation.

For this reason, we discuss a different type of encoding whereby a program is interpreted as a set of modal formulas. The interest in this solution is that it sheds light on the logical nature of the visibility rules underlying the functionalities of encapsulation. The following section explores this issue in full details.

6.1 Programs as S4 Modal Programs

We assume readers' familiarity with the basic notions of modal logics. We refer the interested reader to [13, 8] for a full description of the underlying theory. The modal framework we refer to is S4 with the two modal operators of necessity $\Box$, and of possibility $\Diamond$. Given a set of type and signature declarations $\boldsymbol{\Gamma}$, we define a mapping $\mu_{\boldsymbol{\Gamma}}$ from ground clauses to corresponding modal clauses as follows:

$$\begin{array}{lcl}
\mu_{\boldsymbol{\Gamma}}(o:\tau) & = & \Box\, o:\tau \;\; \text{if } o:\tau \in [\boldsymbol{\Gamma}] \\
\mu_{\boldsymbol{\Gamma}}(o[P]) & = & \left|\begin{array}{ll} \Diamond\, o[P] & \text{if } o:\tau, \tau\{\widehat{P}:\mathit{priv}\} \in [\boldsymbol{\Gamma}] \\ \Box\, o[P] & \text{if } o:\tau, \tau\{\widehat{P}:\mathit{pub}\} \in [\boldsymbol{\Gamma}] \end{array}\right. \\
\mu_{\boldsymbol{\Gamma}}(o[P] \leftarrow G) & = & \mu_{\boldsymbol{\Gamma}}(o[P]) \leftarrow \eta_{\boldsymbol{\Gamma}}(G, o)
\end{array}$$

The first two cases should be intuitively clear. Every type expression is interpreted as a necessary true statement. Counterwise, a molecule is interpreted as a possibly true or necessarily true formula depending on the status of the predicate symbol of the molecule's p-term. The same intuition justifies the encoding of the head of a clause. On the other hand, the mapping for the body of the clause is dependent on the oid of the molecule which constitutes the head. The mapping $\eta_{\boldsymbol{\Gamma}}$ takes two arguments – a ground goal formula and an object $o \in O_{\perp}$ – and returns a modal formula according to the following rules:

$$\begin{array}{lcl}
\eta_{\boldsymbol{\Gamma}}(o:\tau,\, o') & = & \mu_{\boldsymbol{\Gamma}}(o:\tau) \\
\eta_{\boldsymbol{\Gamma}}(o[P],\, o') & = & \left|\begin{array}{ll} \Box\, o[P] & \text{if } o \neq o' \\ \mu_{\boldsymbol{\Gamma}}(o[P])) & \text{if } o = o' \end{array}\right. \\
\eta_{\boldsymbol{\Gamma}}((G_1, G_2),\, o) & = & \eta_{\boldsymbol{\Gamma}}(G_1, o),\, \eta_{\boldsymbol{\Gamma}}(G_2, o)
\end{array}$$

Here the first and third cases should be obvious. Again every type expression is encoded as a necessary true statement; on the other hand, the encoding of a conjunctive formula is given, as expected, inductively on the encoding of the conjuncts. The case of atomic molecules is more interesting and should be contrasted with the proof rules (LOCAL) and (NON-LOCAL) defined in section 4. The second argument of the mapping $\eta_{\boldsymbol{\Gamma}}$ is meant to model the context of evaluation used in the proof rules: the proof that a property holds at an object different than the current one corresponds, in the modal framework, to a proof that the property is not simply true but rather *necessarily* true. If, instead, we are to prove that a property holds at the current object, then the encoding is consistent with the encoding which applies to the current object.

We can finally define the encoding of a program as a set of modal clauses. For any program $\boldsymbol{\Pi} = \boldsymbol{\Gamma}:\mathbf{P}$ let $\boldsymbol{\Gamma}^\star$ and $\mathbf{P}^\star$ denote the ground closure respectively of $\boldsymbol{\Gamma}$ and of $\mathbf{P}$. Then the corresponding modal program is defined as the set $\mu(\boldsymbol{\Pi}) \;=\; \{\, \mu_{\boldsymbol{\Gamma}}(cl) \mid cl \in \mathbf{P}^\star \,\} \cup \{\, \Box\, o:\tau \mid o:\tau \in \boldsymbol{\Gamma}^\star \,\}$.

6.2 S4 Satisfiability and Entailment

The notions of possible and necessary truth have a very elegant interpretation in the possible-world semantics of modal logic [13]. A necessarily true proposition in a given world is a proposition that is true not only in that world, but also in every possible world accessible from the present one. In contrast, a possibly true proposition is one which might turn out to be true in at least one of the possible worlds accessible from the present one. Within this setting, an S4 Kripke interpretation can be defined as a triple $K = \langle W, \mapsto, \phi \rangle$ where W is the set of worlds (or *substates*), $\mapsto$ is a reflexive and transitive relation over W and the *valuation* function ϕ is defined over W and ranges over the power-set of the atomic formulas of the language. The truth of a formula α in an S4 interpretation K at substate w is formalized by the following definition [13]:

$$\begin{array}{lll}
K, w \models_{S4} \alpha & \text{iff} \;\; \alpha \in \phi(w) \quad (\alpha \text{ atomic}) & \\
K, w \models_{S4} \alpha \wedge \beta & \text{iff} \;\; K, w \models_{S4} \alpha \;\text{ and }\; K, w \models_{S4} \beta & \\
K, w \models_{S4} \alpha \leftarrow \beta & \text{iff} \;\; K, w \models_{S4} \beta \implies K, w \models_{S4} \alpha & \\
K, w \models_{S4} \Box\, \alpha & \text{iff} \;\; K, w' \models_{S4} \alpha & \text{for every } w' \text{ s.t. } w \mapsto w' \\
K, w \models_{S4} \Diamond\, \alpha & \text{iff} \;\; K, w' \models_{S4} \alpha & \text{for at least one } w' \text{ s.t. } w \mapsto w'
\end{array}$$

A formula α is said to be true in K iff $K, w \models_{S4} \alpha$ for all the substates of K; α is S4 valid iff it is true in every S4 interpretation.

We are now ready to prove the main result of this section. The soundness and completeness of the proof theory $\vdash$ with respect to entailment in S4 modal logics is established in terms of the following theorem. In order to simplify the discussion we restrict the analysis to the case of ground programs. The extension to the non-ground case is simple and can be handled in ways similars to what we do below.

Theorem 6. *Let* $\boldsymbol{\Pi}$ *be a* ground *program. For every* ground *goal* G, $\boldsymbol{\Pi} \vdash G$ *iff* $\mu(\boldsymbol{\Pi}) \models_{S4} \Box\, G$.

Note that, when G is ground, as in the statement of the theorem above theorem, we can write $\Box\, G$ to stand for $\Box\, G_1, \cdots, \Box\, G_n$ when G is the conjunctive goal $G_1, \cdots, G_n$. In fact, it is easy to see that for every S4 interpretation K, $K \models_{S4} \Box\,(G_1 \wedge G_2)$ iff $K \models_{S4} \Box\, G_1 \wedge \Box\, G_2$.

The rest of this section is dedicated to the proof of theorem 6. We will use the term program throughout to refer to ground programs (or, equivalently, to ground closures of non-ground programs).

Soundness. The soundness of $\vdash$ can be proved easily following an indirect argument. We can show that $\vdash$ is sound with respect to the proof relation of S4. Then the proof follows directly by observing that the proof theory of S4 is sound with respect to $\models_{S4}$. The following lemma can be proved by a straightforward induction on the structure of a goal. Let $\vdash_{S4}$ denote the proof predicate of S4.

Lemma 7. *Let* $\boldsymbol{\Pi} = \boldsymbol{\Gamma} : \mathbf{P}$ *be a program,* G *be a ground goal. Then for every* $o \in O_\perp$, $\boldsymbol{\Pi}, o \vdash G \implies \mu(\boldsymbol{\Pi}) \vdash_{S4} \eta_{\boldsymbol{\Gamma}}(G, o)$. $\Box$

Theorem 8 (SOUNDNESS). *Let* $\boldsymbol{\Pi} = \boldsymbol{\Gamma} : \mathbf{P}$ *be a program. For every groud goal* G, $\boldsymbol{\Pi} \vdash G \implies \mu(\boldsymbol{\Pi}) \vdash_{S4} \Box\, G$

Proof. Apply lemma 7 with $o = o_\perp$ and observe that $\eta_{\boldsymbol{\Gamma}}(G, o_\perp) = \Box\, G$ for every ground G. $\Box$

Completeness. The proof of the opposite direction of the implication established in theorem 8 can be carried out using a technique inspired by the Henking construction of modal logic [17, 13]. This construction not only allows us to obtain a completeness proof, but it also provides us with a constructive method for defining a canonical S4 model for every program.

Definition 9 (CANONICAL MODEL). For every program $\boldsymbol{\Pi} = \boldsymbol{\Gamma} : \mathbf{P}$, we define the *canonical* model $K_{\boldsymbol{\Pi}}$ of the program as the structure $\langle W, \mapsto, \phi \rangle$ such that:

- the set W coincides with the universe $O_\perp$;
- the valuation function ϕ associates with each substate the set of atomic formulas provable in the program from the object corresponding to the substate. Formally $\phi(o) = \{A \mid \boldsymbol{\Pi}, o \vdash A\}$ where A is a ground molecule or type expression;
- the accessibility relation $\mapsto$ is the relation that satisfies the condition that $o \mapsto o'$ for every $o, o' \in O_\perp$.

The set of the worlds coincides with the universe of objects and should not sound surprising: it simply models the fact that the possible worlds where we interpret our programs are nothing else than the objects themselves, which coincide with the individuals of the universe of discourse. It is interesting to remark that this is indeed the very idea behind the "complex object" approach to data modeling where values and objects are undistinguished.

The canonical model enjoys two important properties, which are stated in the following lemma. The proof of the following result is omitted due to the lack of space and can be found in [7].

Lemma 10. *Let $\boldsymbol{\Pi} = \boldsymbol{\Gamma} : \mathbf{P}$ be a program. Then,*

1. $K_{\boldsymbol{\Pi}} \models_{S4} \eta_{\boldsymbol{\Gamma}}(G, o) \implies \boldsymbol{\Pi}, o \vdash G$ *for every ground goal G and$o \in O_\perp$,*
2. $K_{\boldsymbol{\Pi}} \models_{S4} \mu(\boldsymbol{\Pi})$ *i.e. $K_{\boldsymbol{\Pi}}$ is an S4 model of for $\boldsymbol{\Pi}$.*

Completeness follows almost immediately from these results.

Theorem 11 (COMPLETENESS). *Let $\boldsymbol{\Pi} = \boldsymbol{\Gamma} : \mathbf{P}$ be a program. Then, for every ground goal G, $\mu(\boldsymbol{\Pi}) \models_{S4} \Box\, G \implies \boldsymbol{\Pi} \vdash G$.*

Proof.

$$\begin{aligned}
\mu(\boldsymbol{\Pi}) \models_{S4} \Box\, G &\iff K \models_{S4} \mu(\boldsymbol{\Pi}) \implies K \models_{S4} \Box\, G \quad \text{for every S4 int. } K \\
&\implies K_{\boldsymbol{\Pi}} \models_{S4} \Box\, G \quad \text{by lemma 10.2 being } K_{\boldsymbol{\Pi}} \models_{S4} \mu(\boldsymbol{\Pi}) \\
&\iff \exists o \in O_\perp \text{ s.t. } \eta_{\boldsymbol{\Gamma}}(G, o) = \Box\, G \ \wedge\ K_{\boldsymbol{\Pi}} \models_{S4} \eta_{\boldsymbol{\Gamma}}(G, o) \\
&\implies K_{\boldsymbol{\Pi}} \models_{S4} \eta_{\boldsymbol{\Gamma}}(G, o_\perp) \ \text{ choosing } o = o_\perp \\
&\implies \boldsymbol{\Pi}, o_\perp \vdash G \ \text{ by lemma 10.1} \\
&\iff \boldsymbol{\Pi} \vdash G \ \text{ by definition}
\end{aligned}$$

7 Related Research

We refer the readers to [15] for a comprehensive review of contemporary research in the area of deductive object-oriented query languages, and we will not review them here for the want of space.

Although there are quite a number of interesting proposals for deductive object-oriented languages, most of them are mainly concerned with the issues related to inheritance, object identity, methods, signatures, etc. but none, to our knowledge, have addressed the issue of encapsulation in its true logical sense. For example, encapsulation in F-logic [15] is viewed as type-correctness mechanism based on the idea of *modules*, and consequently is a *meta-logical* notion.

The logical theories of modules and program composition also have some form of built-in encapsulation. Miller [18] proposes a way to represent modules in logic programming via the intuitionistic embedded implications. Baldoni et al. [4] uses modal logic to capture visibility rules in modules. In these approaches, however, the resulting languages retain the relational flavor of data peculiar to logic programming and, as such, differ from conventional object oriented languages, both syntactically and semantically. Furthermore, they do not reveal the exact nature and strength of encapsulation.

In contrast to all these proposals, we have shown that our language has a direct first-order semantics and enjoys a sound and complete proof theory. Our proposal is also general enough and can be used for other languages, specially languages similar to F-logic that have schema definition capability.

8 Conclusions and Future Research

We have defined a declarative object-oriented language to model encapsulation and have studied several semantics for this language. We have defined a direct semantics where the notion of encapsulation is built into the proof theory and the model theory[5] of the language. We have also shown that the proof theory is sound and complete with respect to the notion of entailment in S4 modal logic. In particular, we have shown that the visibility rules affecting encapsulation can be characterized elegantly in terms of the modal notions of possible and necessary truth. The notion of satisfaction introduced in section 5 and the modal characterization of the language discussed in section 6 are equally important: the former gives direct insights into the computational properties of encapsulation as well as a formal account of its semantics, and the latter provides an elegant logical foundation for our mechanism.

As we have mentioned at the outset, we choose to deal with a rather simpler language in order to isolate the functionalities of encapsulation. However, it is

[5] In this paper we have presented a model theory based on Herbrand interpretations only for the sake of simplicity and brevity. It is, however, easy to see that developing a general interpretation for our language is straightforward. Interested readers are referred to [7] for a discussion on the subject. The proof for most of our lemmas and theorems can also be found in [7] which we have omitted for the want of space.

certainly desirable to bring other important object-oriented concepts into the language so as to increase its expressive power and flexibility. Some extensions, like the ability to define recursive type definitions of the form $T \leftarrow T_1, \ldots, T_n$ are already at hand. In fact, all the results mentioned in the paper extend immediately to this case provided that the extension of the type definitions preserves the well-typedness of the program. Other extensions are the subject of our current research. We are working on the definition of an object-oriented language in the line of F-Logic which combines encapsulation with the (non-monotonic) inheritance of properties, methods and signatures. As in the approach we have described here, the semantics of the extended language is captured by a notion of satisfaction that models a proof system affected by several inter-related forms of inference: classical deduction, deduction by inheritance, overriding, and encapsulation.

References

1. A. Abiteboul. Towards a deductive object-oriented language. *Data and Knowledge Engineering*, (5):263–287, 1990.
2. A. Abiteboul and P. C. Kanelakis. Object identity as a query language primitive. In *Proceedings of the ACM SOGMOD Int. Conference on the Management of Data*, pages 159–173, 1989.
3. K. R. Apt and M.H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.
4. M. Baldoni, Giordano, and A. Martelli. A Multimodal Logic to Define Modules in Logic Programming. In D. Miller, editor, *Proceedings of the International Logic Programming Symposium ILPS'93*, pages 473–487. The MIT Press, 1993.
5. C. Beeri. A formal approach to object-oriented databases. *Data and Knowledge Engineering*, (5):353–382, 1990.
6. M. Bugliesi and H. M. Jamil. A Stable Model Semantics for Behavioral Inheritance in Deductive Object Oriented Languages. (Manuscript).
7. M. Bugliesi and H. M. Jamil. A Logic for Encapsulation in Object Oriented Languages. Technical Report 5, Dip. Di Matematica Pura ed Applicata, Universitá di Padova, 1994.
8. B. F. Chellas. *Modal Logic: an Introduction.* Cambridge University Press, 1980.
9. W. Chen and D. S. Warren. C-Logic for Complex Objects. In *ACM SIGMOD Conference on Management of Data*, 1989.
10. J.S. Conery. Logical objects. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Conference on Logic Programming*, pages 420–434. The MIT Press, 1988.
11. C. Delobel, M. Kifer, and Y. Masunaga, editors. *Proceedings of the 2nd International Conference of Deductive Object-Oriented Databases.* Springer-Verlag, 1991. Appeared as LNCS 566.
12. G. Dobbie and R. Topor. A Model for Inheritance and Overriding in Deductive Object-Oriented Systems. In *Sixteen Australian Computer Science Conference*, Jannuary 1988.
13. G. E. Hughes and M. J. Cresswell. *An Introduction to Modal Logic.* Methuen & Co. Ltd, 1968.

14. S. Kamin. Inheritance in SMALLTALK-80: a denotational definition. In *Proceedings of the ACM Int. Conf. on Principles of Programming Languages*, pages 80–87, 1988.
15. M. Kifer, G. Lausen, and J. Wu. Logical Foundations for Object-Oriented and Frame-Based Languages. Technical Report TR-93/06, Department of Computer Science, SUNY at Stony Brook, 1993. (accepted to Journal of ACM).
16. W. Kim, J-M. Nicola, and S. Nishio, editors. *Proceedings of the 1st International Conference of Deductive Object-Oriented Databases*. 1989.
17. S. A. Kripke. Semantical Analysis of Intuitionistic Logic. In J. N. Crossley and M. A. E. Dummett, editors, *Formal Systems and Recursive Functions*, pages 92–130. North-Holland Pub. Co., 1965.
18. D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, (6):79–108, 1989.
19. U. Reddy. Objects as Closures: Abstarct Semantics of Object Oriented Languages. In *Proceedings of the ACM Int. Conf. on Lisp and Functional Programming*, pages 289–297, 1988.