A Stable Model Semantics for Behavioral Inheritance in Deductive Object Oriented Languages

(Extended Abstract)

Michele Bugliesi[1]
Dip. di Matematica Pura ed Applicata
Università di Padova, Italy
michele@goedel.math.unipd.it

Hasan M. Jamil[2]
Dept. of Computer Science
Concordia University, Canada
jamil@cs.concordia.ca

Abstract. We present a model for deductive object oriented query languages with inheritance and overriding. In this model, we consider a DAG like dynamic *isa* hierarchy and we account for both *value* or *attribute* inheritance and *method* inheritance or *code sharing*. We show that these two types of inheritance can be treated uniformly within an elegant declarative setting. We then propose a novel semantics for the non-monotonic behavior resulting from the combination of overriding, dynamic *self* binding and the dynamic structure of the *isa* hierarchy. This semantics is reminiscent of the stable model semantics of logic programs with negation. We also isolate a syntactic condition that guarantees the existence of a unique stable model for a program. This condition, in its turn, is inspired by the local stratification condition of perfect model semantics for programs with negation. Finally we define a bottom-up procedure that computes the unique stable model of a stratified program.

1 Introduction

There have been several attempts at combining inheritance with deductive programming languages within clean mathematical settings [1, 2, 3, 5, 6, 8, 9, 10, 11, 12, 13, 14]. Inheritance is an essential concept in AI and in object-oriented programming that comprises two main aspects: *structural* and *behavioral* inheritance. Structural inheritance is a mechanism for propagating method declarations and signatures from classes to their subclasses or instances. Behavioral inheritance, on the other hand, propagates method implementations as well as the result of their application.

Logic languages like LOGIN [1] and LIFE [2] incorporate structural inheritance by means of an extended unification algorithm for ψ-terms, complex typed structures that are used for data representation. In [10], Kifer et al. proposed a formalism, called F-Logic, for deductive object oriented database query languages where the semantics of structural inheritance is captured within an elegant model theory and a sound and complete proof theory. F-Logic, together

[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.

with other related formalisms, have also addressed the issue of behavioral inheritance. However, there are several aspects of the resulting models that can be objected to in these approaches: we will discuss some of these aspects later on in this paper, but only after having presented our model.

The object oriented language we consider here is loosely related to F-Logic, but the syntax and semantics are quite different. In particular, we consider only behavioral inheritance in our model and, consequently, disregard method signatures and structural inheritance which are peculiar to F-Logic. Similarly to F-Logic, we allow the *isa* hierarchy to be defined dynamically by allowing rules with schema and method components. Within this setting, we consider both value and method inheritance with overriding, multiple inheritance and we focus our attention to only set-valued methods. This choice is motivated by the fact that it allows us to capture the semantics of multiple inheritance in a quite natural and elegant way. Our syntax as well our semantics for set-valued methods is first order in that our variables range over the elements of a set rather than on (the extension of) that set.

We propose the notion of *inheritance by completion* (*i-completion*) of a program and present an abstract semantics that is based on conventional notions of interpretation and satisfaction. This semantics is reminiscent of the stable model semantics of [7]: as in that case, due to the non-monotonic nature of overriding, a program may have more than one stable model, or no stable model at all. However, we isolate a syntactic condition, that we call *i-stratification*, that guarantees the existence of a unique stable model. This condition is reminiscent of the stratification condition of [15] for logic programs with negation, but in our case it constrains the combination of deduction and inheritance with overriding. Reasoning on the i-completion of our programs, we prove that i-stratification is sufficient to guarantee the existence and uniqueness of the stable models. The definition of i-completion provides also the basis for defining a bottom-up computation of the unique stable model of every i-stratified program.

We organize the rest of our paper as follows. In Section 2 we present our model of inheritance and we discuss the informal semantics by means of simple examples. In Section 3 we introduce the notion of i-completion and present the stable model semantics. Then, in Section 4, we introduce the i-stratification condition and we prove the results of existence and uniqueness of stable models. We then address similarities and differences with related work in Section 5, and finally conclude in Section 6 discussing the extensions of the present model that we plan for our future research.

2 The Inheritance Model

In this section we present the salient features of the inheritance model we consider throughout. We do this by introducing a simple deductive object oriented query language with inheritance: the language doesn't account for a number of important object oriented concepts like signatures, structural inheritance, encapsulation, etc. However, it comprises the essential functionalities related to

behavioral inheritance: multiple inheritance, overriding, dynamic self binding, set-valued methods, etc.

2.1 Syntax

Every program in this language uses symbols from an alphabet $\langle \mathcal{V}, \mathcal{C}, \mathcal{P} \rangle$ where $\mathcal{V}$ is a denumerable set of variables, $\mathcal{C}$ is a set of data constructors and $\mathcal{P}$ is a set of property (attribute and method) symbols. These components are assumed to be pairwise disjoint. We call *o-terms* the terms built over $\mathcal{C} \cup \mathcal{V}$, and *p-terms* the terms $\mathbf{p} = p_{/a}(args)$ where $p \in \mathcal{P}$ is a property name with arity a, and $args$ is a tuple of o-terms. To ease the notation, we will always denote the property symbols from $\mathcal{P}$ using only their names, with the understanding that every name has an associated unique arity. An o-term is a first order entity in the language and denotes the *object identity* (oid) of a class or instance object. The set of oids is denoted by $\mathcal{O}$.

Atomic and Complex Formulas. *Molecules* and *isa terms* are the atomic formulas of the language. Their structure is defined as follows.
Molecules are statements of the form $o[\mathbf{p}]$, where o is an oid and $\mathbf{p}$ is a p-term (called respectively the molecule's oid and p-term). The intended meaning of a molecule in our language is essentially the same as in F-logic: $o[\mathbf{p}]$ states that property $\mathbf{p}$ holds at object o.
Isa terms are statements of the form $o : c$ or $c :: d$ where o, c and d are o-terms. The intention of an *isa* term is to establish the subclass/membership relation between two objects: $o : c$ states that o is an instance of c, whereas $c :: d$ states that c is a subclass of d. We will often denote with $\sharp$ the type of the *isa* relation between objects: given a class object c, $o\sharp c$ stands for $o : c$ when o is an instance object and for $o :: c$ when o is itself a class object.

Complex formulas are definite clauses of the form $A \leftarrow B_1, \ldots, B_n$ where A and the B_is are molecules and/or *isa* terms. We will call A and the B_is respectively the head and the body literals of the clause and we will assume that all the variables occurring in a clause are universally quantified.

Programs and Queries. As in other object oriented languages, a program in our language specifies which methods/attributes are attached to each object and organizes objects along *isa* hierarchies. Every program can be conceptually viewed as consisting of two parts, each one dedicated to the specification of one of these two components.

Definition 1 (PROGRAMS). A program is a pair $\Gamma : \Pi$ where:

- Γ, the *schema declaration*, is a (possibly empty) set of *isa clauses* whose head is an *isa* term and whose body literals are either *isa* terms or molecules;
- Π, the *property or data definitions*, is a (possibly empty) set of *method clauses* whose head is a molecule and whose body literals are either *isa* terms or molecules.

When the oid in the head molecule of a method clause is a variable, we will assume that there exist an *isa* term $X \sharp o$ in the body to qualify X as an instance or subclass of some object. This assumption does not involve any loss of generality: it serves the only purpose of disallowing clauses like "$X[\mathbf{p}]$." that establish the truth of a property, $\mathbf{p}$ in this case, at *every* object.

The only structural distinction between a method clause and an *isa* clause is that the former has a molecule as its head whereas the latter has an *isa* term. Thus it is possible that the *isa* clauses in the schema and the method clauses in the data definitions of a program depend on each other: the satisfaction of a property at an object may depend on the structure of the *isa* hierarchy (through an *isa* term) and vice versa. Consequently, as in F-Logic, we allow a dynamic structure of the *isa* hierarchy.

2.2 Informal Semantics

The *isa* clauses of the schema organize objects in a DAG-like hierarchy. The interplay between membership and subclassing is subject to the standard condition: every instance of a class is also an instance of all of the super-classes of that class. In other words, where o, c and d are different objects, $o : c$ and $c :: d$ implies that $o : d$. Finally we assume that subclassing and membership are reflexive: an object is always a subclass and an instance of itself.

Each class defines a set of properties (methods and/or attributes) for its instances and subclasses. Every object inherits all the properties that are defined at the objects that are placed higher up in the hierarchy. There are two ways that a property can be inherited, either extensionally or intensionally: we refer to these two types of inheritance respectively as *value inheritance*, and *method inheritance* or *code sharing*.

In the sequel of this section we illustrate the functionalities of inheritance, as well as the interaction of inheritance and overriding by means of a number of simple examples. Later, in section 3, we will formalize these ideas precisely.

Value and Method Inheritance. The difference between value inheritance and method inheritance can be explained as follows. Method inheritance is, in a way, built into our syntax and originates from the interplay between instantiation and the *isa* relations of the schema. Value inheritance, instead, is enforced by our intuitive understanding of the interaction between the *isa* relation and deduction. The following example helps clarify the point.

$$\Gamma_1 ::= \left| \begin{array}{ll} (1) & o : c. \\ (2) & c :: d. \end{array} \right. \qquad \Pi_1 ::= \left| \begin{array}{ll} (3) & c[q(b)]. \\ (4) & X[p(a)] \leftarrow X : d, def_p \end{array} \right.$$

The schema states that o is an instance of c and that c is a subclass of d. Given that every object inherits from its class ancestors in the *isa* hierarchy, here we have that o inherits $q(b)$ from its class c. Similarly, o inherits a definition for p from class d: this is because we can substitute o for X in (4) and, since $o : d$ due to the interplay between ":" and "::", clause (4) can be seen as a definition that o inherits from d. If def_p, succeeds, then we will be able to derive $o[p(a)]$.

Note the difference between the two cases. In the former, o is inheriting the extension of a property from its class c: we will say that o value-inherits q from c. In the latter case, it is the intension of the property (the clause defining it) that gets inherited from d to o: accordingly, we will say that o method-inherits p from d.

Overriding. If there were no overriding, we could account for the two types of inheritance in an elegant and easy way. We would simply need to model the relationship between the *isa* relations and substitution/deduction and have our objects be characterized by all the properties they inherit via instantiation and/or deduction in ways similar to those outlined above. With overriding the picture becomes more complex, because there may be conflicts between the types of inheritance and we may want to reject the inheritance of values (or clauses) along the *isa* hierarchies in case properties are redefined at a subclass or instance. Consider for example the following program.

$$\Gamma_2 ::= \left| \begin{array}{ll} (1) & o : c. \\ (2) & c :: d. \end{array} \right. \qquad \Pi_2 ::= \left| \begin{array}{ll} (3) & c[p(b)]. \\ (4) & X[p(a)] \leftarrow X : d. \end{array} \right.$$

The point is: how should we answer the query $o[p(X)]$? Both the answers $X = a$ and $X = b$ seem reasonable, because $X = b$ follows from (3) being $o : c$, whereas $X = a$ follows from (4) being $o : d$ implied by the schema. If there were no overriding, then we would certainly accept both the answers as legal and, consequently, say that $\{a, b\}$ is the value of p at o. However, if we assume that inheritance is subject to overriding, then clearly we have a conflict. In this case we claim that the only acceptable answer to the above query is $X = b$, because the inheritance of p from c to o overrides the inheritance of the same property from d to o.

Following the same line of reasoning, if we assume that $w : d$ be part of the schema in the example above, we will interpret the two following clauses:

$$X[t(a)] \leftarrow X : c, def_1.$$
$$X[t(b)] \leftarrow X : d, def_2.$$

as two definitions for t that o and w inherit from their (super)-classes. More precisely, w inherits the second clause from d, whereas o inherits the first clause from c and the second from d. Again, the inheritance of t from c to o overrides the inheritance from d to o. The same arguments apply to the following slightly more complex example.

$$\Gamma_3 ::= \left| \begin{array}{ll} (1) & o : c. \\ (2) & u : c. \\ (3) & c :: d. \end{array} \right. \qquad \Pi_3 ::= \left| \begin{array}{ll} (4) & c[p(b)]. \\ (5) & u[p(e)]. \\ (6) & X[p(a)] \leftarrow X : d. \end{array} \right.$$

Here, the expected answer to the query $u[p(X)]$ is $X = e$ rather than $X = b$. This is because the inheritance of p from c to u is overridden owing to the existence of the local definition (5) for p at u.

Multiple Inheritance. The interaction between inheritance and overriding we have outlined above applies to every path in the *isa* hierarchy: each object inherits a property and/or the clauses defining it from the closest ancestors in the hierarchy that define that property. In order to formalize this notion of "closeness", we assume that no pair of immediate ancestors of any given object be connected by an *isa* link. Accordingly, $\Gamma = \{o : c, o : d\}$ is a valid schema whereas $\Gamma' = \{o : c, c :: d, o : d\}$ violates the assumption because o has two immediate ancestors, c and d, that are connected by the link $c :: d$. In our model, Γ' is interpreted as the linear schema $\{o : c, c :: d\}$ where c is closer to o than d. In other words, the model does not distinguish the cases when $o : d$ is asserted or entailed by the schema.

Dynamic Subclassing. In all of the previous programs, we have seen examples of *static* schema definitions, where the *isa* clauses do not depend on the data definitions. Consider now the more complex case of the following program.

$$\Gamma_4 ::= \left| \begin{array}{ll} (1) & o : c \leftarrow o[p(a)]. \\ (2) & o : d. \end{array} \right. \qquad \Pi_4 ::= \left| \begin{array}{ll} (3) & d[p(a)]. \\ (4) & c[p(b)]. \end{array} \right.$$

Note that the *isa* relation between o and c depends now on the satisfaction of $p(a)$ at the object o. Here, inheritance works as before: from $o : d$ and $d[p(a)]$, we can derive $o[p(a)]$ by inheritance from d to o. Then, by standard deduction we derive $o : c$, and hence, $o[p(b)]$ by inheritance from c to o. Therefore, we conclude that the value of p at o is the set $\{a, b\}$.

As a final example, consider adding the *isa* clause $c :: d \leftarrow o[p(a)]$ to the previous program. $\Gamma_4 : \Pi_4$. We obtain the new program (which we adapt from [10]).

$$\Gamma_5 ::= \left| \begin{array}{ll} (1) & o : c \leftarrow o[p(a)]. \\ (2) & c :: d \leftarrow o[p(a)]. \\ (3) & o : d. \end{array} \right. \qquad \Pi_5 ::= \left| \begin{array}{ll} (4) & d[p(a)]. \\ (5) & c[p(b)]. \end{array} \right.$$

As in the previous example, we can derive $o[p(a)]$ from $o : d$ and $d[p(a)]$. Now, however, by standard deduction, we derive not only $o : c$ but also $c :: d$. But this implies that o should inherit p from c, and consequently, that this inheritance should override the inheritance of the same property from d to o. In other words, we shouldn't have used $o : d$ to deduce $o[p(a)]$. However, if we disregard $o : d$, then we are not even allowed to infer $o : c$, and hence we conclude that the value for p at o is the empty set,

Neither one of the two conclusions seems reasonable: indeed this program doesn't seem to have any sensible (determinate) meaning. As we will show in section 4, our semantics *does* classify programs like $\Gamma_5 : \Pi_5$ as *meaningless* programs because they have no stable models.

3 Stable Model Semantics

As in the classical theory of logic programming, an interpretation of a program in our language is a subset of the Herbrand base over the alphabet of the program. The only additional requirement in our theory is that we assume that

interpretations be *isa* closed: that means that, whenever $o\sharp c$ and $c :: d$ belong to an interpretation I, we require that $o\sharp d$ be also contained in I. The condition of *isa* closedness provides a formal justification for the equivalence of the two schemas $\{o : c, c :: d, o : d\}$ and $\{o : c, c :: d\}$ we have discussed in the previous section.

Satisfaction in an *isa* closed interpretation is defined exactly as in classical Herbrand interpretations in terms of membership. To account for inheritance in this framework, we introduce the notion of i-completion discussed in the next subsection[1].

3.1 Inheritance by I-completion

We first present the rational behind the idea of i-completion on intuitive grounds. Consider the following program:

$$\Gamma ::= (1) \quad o : c. \qquad\qquad \Pi ::= (2) \quad c[p(a)].$$

In every model of this program we would expect to see both $c[p(a)]$ (of course) and $o[p(a)]$ because it can be inferred by value inheritance. However, the fact that $p(a)$ holds at o is not expressed explicitly in the program: it is our idea of the semantics of inheritance that implies it. This is in fact a general issue: value inheritance is not expressed syntactically in our programs; it is a purely semantic mechanism we are attributing to them. In contrast, method inheritance *does* have a syntactic representation owing to substitutions. So the point is: why not model value inheritance in terms of method inheritance so that we can account for value inheritance syntactically the way we do for method inheritance? It is easy to see how this can be accomplished, at least in the previous program: simply, consider the following *completed* program:

$$\Gamma ::= (1) \quad o : c. \qquad\qquad \Pi ::= \left| \begin{array}{ll} (2) & c[p(a)]. \\ (3) & o[p(a)] \leftarrow o : c, c[p(a)]. \end{array} \right.$$

Note that clause (3) is inherited by o from c. It states that whenever $p(a)$ holds at c, it also holds at o: exactly as in the original program, with the difference that now the value inheritance from c to o is modeled in terms of the inheritance of clause (3) between the two objects. This simple transformation extends naturally to the general case as suggested in the following definition.

Definition 2 (I-COMPLETION). Let P be a set of clauses and let $[P]$ be the ground closure of P. The *i-completion* of P, denoted by $C(P)$ is the minimal set

[1] In [4], we present an alternative semantics based on complex interpretation structures, called β-structures. Using these structures, we are able to capture the functionalities of behavioral inheritance and overriding directly within the definition of satisfaction, without resorting to the notion of i-completion. In [4] we also show the equivalence of the notion of model that results in that framework and the definition of stable model we present in this paper.

of clauses satisfying the following conditions:

1. $[P] \subseteq [C(P)]$
2. $o\sharp c \leftarrow B. \quad c :: d \leftarrow B. \in [C(P)] \implies o\sharp d \leftarrow o\sharp c, c :: d. \in [C(P)]$
3. $c[\mathbf{p}] \leftarrow B. \quad o\sharp c \leftarrow B. \in [C(P)] \implies o[\mathbf{p}] \leftarrow o\sharp c, c[\mathbf{p}]. \in [C(P)].$

The effect of i-completing a program is to expose, syntactically, all the inheritance that is implicitly expressed in the original program. As a consequence, the semantics of an i-completed program can be given simply in terms of deduction as it does not need to make reference to inheritance: what in the original program is inferred by value inheritance can be inferred, in the i-completed program, by standard deduction using the clauses added by the i-completion. In both cases, method inheritance is implicitly entailed by substitution.

Clearly, we still need a formal account for overriding, but the use of i-completion allows us to capture the functionalities of inheritance in terms of a standard notion of satisfaction: we can characterize the semantics of an i-completed program simply in terms of its (classical) minimal Herbrand model.

3.2 Overriding

Before we move on to introduce overriding, we put forward the definitions of *local* and *inherited* clauses that, adapted from [8], help formalize this notion.

Definition 3 (LOCAL METHOD CLAUSES). Let cl be a ground (instance of a) method clause. We say that cl is *local* to $o \in \mathcal{O}$ iff $cl = o[\mathbf{p}] \leftarrow B_1, \ldots, B_n$ and there exists no i such that $B_i = o\sharp c$ with $c \neq o$[2].

Definition 4 (INHERITED CLAUSES). Let $cl = o[\mathbf{p}] \leftarrow B_1, \ldots, B_n$ be a ground (instance of a) method clause and let I be an interpretation. We say that cl is *inherited by o from c in I* iff there exists i such that $B_i = o\sharp c$ and $o\sharp c \in I$

Next, we introduce the concepts of *defined* and *inherited* properties. Again, we denote with $\widehat{\mathbf{p}}$ the property symbol of the p-term $\mathbf{p}$ and call $\widehat{\mathbf{p}}$-clause any clause whose head is $o[\mathbf{p}]$ for some object o. We say that o *defines* a property $p \in \mathcal{P}$ iff there exists a p-clause local to o. Similarly, o *inherits p from d in I* iff o inherits a p-clause from d in I.

Our notion of "overriding" is again inspired by the definition of locality of method clauses proposed in [8]. Overriding comes into play whenever an object o inherits the same property, say p, from different ancestors that are connected by *isa* links in the hierarchy. In every such situation, the conflict is resolved by establishing that o inherits p only from the closest ancestors that define p. This inheritance blocks (overrides) the inheritance of p from all the ancestors of o that are placed higher up in the hierarchy. Note that, since we assume that membership and subclassing are reflexive, it follows that if an object defines

[2] Here, and in Definition 5, with "$\doteq$" we denote syntactic equality.

a property, then the local definition overrides the inheritance of that property from any of the (proper) ancestors of that object.

The natural consequence of this interpretation is that for every object o, only a subset of the clauses that o inherits from its ancestors are actually "relevant" to the definition of the properties that hold at o itself. The set of relevant clauses corresponding to the overriding rule we have just outlined is defined precisely as follows.

Definition 5 (OVERRIDING-FREE INSTANCES). Let $P = \Gamma : \Pi$ be a program and I be an interpretation. All the ground instances of the *isa* clauses in Γ are *overriding free in* I. Let $cl = o[\mathbf{p}] \leftarrow B_1, \ldots, B_n$ be ground instance of a method clause in Π. cl is *overriding free in* I iff:

- either cl is local to o;
- or there exists a class c such that o inherits cl from c and o does not inherit $\widehat{\mathbf{p}}$ from any $d \not\equiv c$ such that $\{o\sharp d, d :: c\} \subseteq I$.

In several respects, this approach results in a model theory that is similar to the model theory of Gulog proposed in [6]: as in that case, it is the syntactic structure of the program that determines the set of "relevant" clauses of a program as well as the ways that overriding affects the inheritance of properties. One important difference is that our notion of overriding in a given interpretation is static, as it is based solely on the existence of an overriding definition (regardless of the satisfaction of the body of the definition in the given interpretation). Furthermore, we generalize the definition of model by allowing the *isa* hierarchy to evolve dynamically during the computation. Let $\mathcal{M}(P)$ denote the minimal model of an (i-completed) program P, and let $[P]_I$ denote the set of overriding free instances of P in any given interpretation I.

Definition 6 (STABLE MODELS). Let I be an interpretation and let P be an i-completed program. We say that I is a stable model of P iff $I = \mathcal{M}([P]_I)$.

This definition should be contrasted with the corresponding definition of stable models in [7]. As in that case, given an interpretation I, we isolate the subset of the clauses in P that are "relevant" because they are overriding free in I, and then we check whether the remaining clauses are satisfied by I. Note the recursive flavor of the construction: the set of clauses that must be satisfied in order for I to qualify as a model depend on I itself. Also note that, owing to the dynamic nature of the *isa* definitions in the schema, in the construction of a model I, the set $[P]_I$ may be subject to changes as the interpretation I changes. Hence, this construction may or may not be convergent: the following proposition shows that there exist programs that have no stable models.

Proposition 7. Program $\Gamma_5 : \Pi_5$ of section 2 has no stable model.

Proof. We show that that the i-completion of $\Gamma_5 : \Pi_5$, as shown below, has no stable model.

$$\Gamma ::= \left| \begin{array}{ll} (1) & o : c \leftarrow o[p(a)]. \\ (2) & c :: d \leftarrow o[p(a)]. \\ (3) & o : d. \\ (4) & o : d \leftarrow o : c, c :: d. \end{array} \right. \qquad \Pi ::= \left| \begin{array}{ll} (5) & d[p(a)]. \\ (6) & c[p(b)]. \\ (7) & o[p(a)] \leftarrow o : d, d[p(a)]. \\ (8) & o[p(b)] \leftarrow o : c, c[p(b)]. \\ (9) & c[p(a)] \leftarrow c :: d, d[p(a)]. \end{array} \right.$$

Clauses (4), (7), (8) and (9) have been added by i-completion. Let the above i-completed program be called Q: we show that Q has no stable interpretation. First observe that since (4) is subsumed by (3) we can reason independently of clause (4). Similarly, we can disregard clause (9) since it is "written over" in every interpretation (6) being local to c.

Now observe that clauses (1), (2), (3), (5) and (6) are overriding free in every interpretation. Hence, (3), (5) and (6) being unit clauses, if there exists a stable interpretation I, then I must be a superset of $J = \{o : d, d[p(a)], c[p(b)]\}$. Note also that J itself is not stable: in fact, since clause (7) is overriding free in J, $o[p(a)] \in \mathcal{M}([Q]_J)$. Assume now, by contradiction, that there exists a stable interpretation I for Q. We have two possibilities:

1. If $o[p(a)] \in I$, then $o : c, c :: d \in I$ because I is a model for (1) and (2) that are overriding free. But then clause (6) is not overriding free in I and hence $o[p(a)] \notin I$ being I minimal: a contradiction.
2. If $o[p(a)] \notin I$, then neither $o : c$ nor $c :: d$ belong to I being I minimal. But then, clause (7) is overriding free in I. This, in turn, implies that $o[p(a)] \in I$, being I a model. Again, we have a contradiction. □

It is also not difficult to see that there are programs that have more than one stable model. Consider the following new program.

$$\Gamma ::= \left| \begin{array}{ll} (1) & o : c. \\ (2) & o : d. \\ (3) & c :: d \leftarrow o[p(a)]. \\ (4) & d :: c \leftarrow o[p(b)]. \end{array} \right. \qquad \Pi ::= \left| \begin{array}{ll} (5) & d[p(a)]. \\ (6) & c[p(b)]. \\ (7) & o[p(a)] \leftarrow o : d, d[p(a)]. \\ (8) & o[p(b)] \leftarrow o : c, c[p(b)]. \end{array} \right.$$

This program is not i-complete: to complete it we would need to add the following clauses:

$$\begin{array}{llll} \text{(i)} & o : c \leftarrow o : d, d :: c. & \text{(iii)} & c[p(a)] \leftarrow c :: d, d[p(a)]. \\ \text{(ii)} & o : d \leftarrow o : c, c :: d. & \text{(iv)} & d[p(b)] \leftarrow d :: c, c[p(b)]. \end{array}$$

However, we can disregard these clauses since (i) and (ii) are subsumed respectively by (1) and (2), whereas (iii) and (iv) are written over in every interpretation owing to the presence of the two local definitions (5) and (6) respectively. Now, call P the above program: every model of P is a superset of $\{o : c,\ o : d,\ d[p(a)],\ c[p(b)]\}$ since (1), (2), (5) and (6) are unit clauses that are always overriding free. Furthermore, owing to the presence of clauses (7) and (8), every model must contain either one of $o[p(a)]$ and $o[p(b)]$, but not both

because otherwise we would be led to conclude that the schema of P contains a cycle. Now consider the following two interpretations:

$$I_1 = \{o : c,\ o : d,\ d[p(a)],\ c[p(b)],\ d : c,\ o[p(a)]\}$$
$$I_2 = \{o : c,\ o : d,\ d[p(a)],\ c[p(b)],\ c : d,\ o[p(b)]\}$$

The set of overriding free instances of P in I_1 and I_2 are, respectively, $[P]_{I_1} = [P] \setminus \{(8)\}$ and $[P]_{I_2} = [P] \setminus \{(7)\}$. It is immediate to see that $I_1 = \mathcal{M}([P]_{I_1})$ and $I_2 = \mathcal{M}([P]_{I_2})$ and hence, that I_1 and I_2 are both stable models of P. Furthermore, on the account of the previous observations, we conclude that I_1 and I_2 are actually the only two stable models of this program: yet, neither one is smaller than the other.

4 Existence and Uniqueness of Stable Models

Looking at the previous examples, one notices that the reason why we fail to construct a model is that we have a conflict between the deduction of a property at a given object and the deduction of an *isa* relation for that object. More precisely, the problem is that we use an *isa* relation $o\sharp c$ to derive a molecule $o[\mathbf{p}]$ by value inheritance from c but, having done this, we immediately find out that there exists an intervening object mid such that $o\sharp mid$ and $mid :: c$ and that the existence of mid causes $o\sharp c$ to be overridden for $\widehat{\mathbf{p}}$. In both the previous examples this is the actual reason why we fail to define a (unique) stable model.

4.1 I-stratification

To obtain a stable model, we will need to constrain the dependency of an *isa* term on a molecule such that if the *isa* term $o\sharp c$ is used to derive a molecule $o[\mathbf{p}]$ by inheritance from c, then we will not, at later stages, derive a new *isa* term $o\sharp d$ that blocks the inheritance of $\widehat{\mathbf{p}}$ from c to o. The following definition of stratification ensures this property. Again, let $[P]$ denote the ground extension of P.

Definition 8. Let P be an i-completed program. We say that P is *i-stratified* iff there exists a mapping μ from ground atoms to positive integers such that, for every pair of atoms A and B, the following conditions are satisfied:

1. $\mu(A) \geq \mu(B)$ iff A is the head of a clause of $[P]$ and B is a body literal of that clause;
2. $\mu(A) > \mu(B)$ iff $A = o[\mathbf{p}]$ is the head of a clause of $[P]$ and $B = o\sharp c$ is a body literal of that clause.

The i-stratification mapping μ aims at decomposing a program P in different strata $P^1, \ldots, P^n$ such that $[P]$ can be obtained as the disjoint union of these strata. The intention of condition (2) is to separate clauses defining *isa* relations between objects from clauses defining properties at these objects by placing them

at different strata of the program. If there exists an i-stratification $P^1 \cup \cdots \cup P^n$ of P, then it will satisfy the following property. Assume that P^i contains a clause $o[\mathbf{p}] \leftarrow B_1, \ldots, B_n$ and that there exists B_k such that $B_k = o\sharp c$: then all the clauses whose head is $o\sharp c$ are placed at strata P^j with j strictly lower than i.

The notion of i-stratification, suggests also a way to compute a model of an i-completed program. Let T_P be the following immediate-consequence operator:

$$T_P(I) = \{A \mid A \leftarrow B_1, \ldots, B_n \in P \text{ and } \{B_1, \ldots, B_n\} \subseteq I\} \cup I$$

The intention is to construct a model for a program by repeatedly iterating the T_P operator at each stratum of the program: owing to i-stratification, the set of overriding free instances of each stratum will not be subject to changes as the construction of the model proceeds with iterations at higher strata.

The following theorem shows that the iterated fixed point computation we have just outlined leads indeed to the construction of stable models.

Theorem 9 (EXISTENCE). Let P be an i-complete and i-stratified program and let $P^1 \cup \cdots \cup P^n$ be an i-stratification of P. For every interpretation I, denote with P^j_I the subset of the j-th stratum of $[P]$ consisting of the clauses that are overriding free in I. Finally, let $M^\star_P$ be the interpretation resulting from the following iterated fixed-point computation:

$$\begin{aligned} M_1 &= T^\omega_{P^1}(\emptyset) \\ M_i &= T^\omega_{P^i_{M_{i-1}}}(M_{i-1}) \quad 1 < i \leq n \\ M^\star_P &= M_n \end{aligned}$$

$M^\star_P$ is a stable model for P.

Proof. We use the following two properties:

1. for every interpretation I, $T^\omega_{P^i \cup P^{i+1}}(I) = T^\omega_{P^{i+1}}(T^\omega_{P^i}(I))$ for every $i = 1, .., n-1$.
2. for every $i = 1, \ldots, n$, $P^i_{M_{i-1}} = P^i_{M_n}$ where we take $M_0 = \emptyset$ by definition.

The first property is a well-known property of stratified programs that carries over directly to i-stratified programs. The proof of the second is omitted for the lack of space and can be found in [4]. To show that $M^\star_P = M_n$ is a stable model, we need to show that $\mathcal{M}([P]_{M_n}) = M_n$. From (1) and (2) above, we can proceed as follows:

$$\begin{aligned} \mathcal{M}([P]_{M_n}) = T^\omega_{P_{M_n}}(\emptyset) &= T^\omega_{P^1_{M_n} \cup \cdots \cup P^n_{M_n}}(\emptyset) \\ \text{by (1)} \quad &= T^\omega_{P^n_{M_n}}(T^\omega_{P^{n-1}_{M_n}}(\cdots(T^\omega_{P^1_{M_n}}(\emptyset))\cdots)) \\ \text{by (2)} \quad &= T^\omega_{P^n_{M_{n-1}}}(T^\omega_{P^{n-1}_{M_{n-2}}}(\cdots(T^\omega_{P^1}(\emptyset))\cdots)) = M_n \end{aligned}$$

We conclude the section with the proof that every i-complete and i-stratified program has exactly one stable model. The proof of this result also shows that the construction of $M^\star_P$ is independent of the choice of the i-stratification of P.

Theorem 10 (UNIQUENESS). Let P be an i-complete and i-stratified program. Let I be a stable model of P. Then $I = M^{\star}_{P}$.

Proof. Let I be a stable model and let $P^1 \cup \cdots \cup P^n$ be an i-stratification of P. Then consider the set of overriding free instances $[P]_I = P^1_I \cup \cdots \cup P^n_I$. Clearly $\mathcal{M}([P]_I) = \mathcal{M}(P^1_I \cup \cdots \cup P^n_I)$. Let then N_i be the following sequence of sets:

$$\begin{aligned} N_1 &= T^{\omega}_{P^1_I}(\emptyset) \\ N_i &= T^{\omega}_{P^i_I}(N_{i-1}) \quad 1 < i \leq n \\ N^{\star} &= N_n \end{aligned}$$

Since P is i-stratified, clearly $\mathcal{M}([P]_I) = N^{\star}$. Furthermore, since the minimal model of every i-stratified set of clauses is independent of the i-stratification mapping, the above construction of $\mathcal{M}([P]_I)$ is also independent of the chosen i-stratification of P. Now consider the sequence of sets M_i that result in the construction of $M^{\star}_{P}$ using the stratification $P^1 \cup \cdots \cup P^n$. We show, by induction on i, that for every $i = 1, \ldots, n$, $M_i = N_i$.

Base case. First note that for every interpretation I, $P^1_I \subseteq P^1$. If $P^1_I \subset P^1$, then P^1 must contain an inherited clause $cl = o[\mathbf{p}] \leftarrow o\sharp c, B$ that is overridden in I (for these are the only clauses that can be written over). But then, since P is i-stratified, neither $[P]$ nor P^1 contain any *isa* clause whose head is $o\sharp c$. Hence, by iterating T_P on P^1, cl will never produce $o[\mathbf{p}]$ and, consequently, P^1 and $P^1 \setminus \{cl\}$ have the same minimal model. Since this argument applies to all the clauses of P^1 that are not overriding free in I, we have that: $N_1 = \mathcal{M}(P^1_I) = \mathcal{M}(P^1) = M_1$

Inductive step. First notice that, being I stable, $I = \mathcal{M}([P]_I)$ and consequently, $I = N_n$. Then observe that, by construction, $N_i \subseteq I$ for every i: from this we have that $M_{i-1} \subseteq I$ because, by the inductive hypothesis $N_{i-1} = M_{i-1}$. Now we can show that $P^i_I = P^i_{M_{i-1}}$ using the properties of i-completion and i-stratification. Further details can be found in [4]. □

4.2 Remarks

A natural question that arises at this point is how large is the class of programs that have a unique minimal model. A subclass of the programs that enjoy this property is the class of programs that are simple, in the sense of the following definition.

Definition 11 (SIMPLE PROGRAMS). A program $\Gamma : \Pi$ is *simple* if and only if the body of every clause in the schema Γ is constituted solely of *isa* terms.

That these programs have a unique stable model follows as corollary of the results of the previous section. The proof is immediate since the i-completion of every simple program $\Gamma : \Pi$ can be seen as a two-stratum program $P^1 \cup P^2$ where P^1 and P^2 are the i-completions of respectively Γ and Π.

The case of non-simple programs, where the schema and data definitions may depend mutually on each other, is more complex. In this regard, it is interesting

to note that the condition of i-stratification is precise enough to distinguish the two programs $\Gamma_4 : \Pi_4$ and $\Gamma_5 : \Pi_5$ of section 2. We already showed that the latter has no stable model: it can now be easily verified that the i-completion of this program, introduced in proposition 7, is not stratified. On the other hand, it is easy to see that the sets of clauses displayed below, define a stratification of (the completion of) $\Gamma_4 : \Pi_4$.

$$P^1 ::= \left| \begin{array}{l} o : d. \\ d[p(a)]. \\ c[p(b)]. \end{array} \right.$$

$$P^2 ::= \left| \begin{array}{l} o : c \leftarrow o[p(a)]. \\ o[p(a)] \leftarrow o : d, d[p(a)]. \end{array} \right.$$

$$P^3 ::= \left| \; o[p(b)] \leftarrow o : c, c[p(b)]. \right.$$

In general, it is hard to give a precise characterization of the class of i-stratified programs. However, our contention is that i-stratification is interesting in itself as a structuring principle: it simply requires that the *isa* relation between two objects be independent of the properties whose satisfaction depends itself on that *isa* relation. As such, i-stratification seems indeed to offer a reasonable principle for writing programs that exploit the power of inheritance in meaningful and practical ways.

5 Discussion on Related Work

In this section we take a very brief look at other proposals that are related to our present work. Readers are referred to [10] for a lucid and comprehensive discussion on the contemporary approaches to inheritance in the literature.

In *L&O* [12], the semantics of inheritance and overriding is given indirectly by translating *L&O* program to logic programs and, hence, it provides little insight into the relationships between inheritance, overriding and deduction.

In F-Logic [10], only structural inheritance is captured semantically within the model theory and the proof theory of the formalism. Counterwise, for behavioral inheritance, the non-monotonic aspects introduced by the combination of overriding and dynamic binding are modeled only indirectly by means of an iterated fixed point construction. Another weakness of F-Logic is that it accommodates only value inheritance: in F-Logic, what gets inherited along the *isa* hierarchy is ground data expressions – values resulting from the application of a method at a superclass – and not method implementations. Method inheritance and overriding, in their turn, are accounted for only indirectly by means of an ad-hoc technique that relies on the higher order features of this formalism. Finally, in F-Logic the problems introduced by the dynamic structure of the schema are solved resorting to a highly non-deterministic semantics: in F-Logic a program might have more than one model and no mechanism is provided so that one can systematically identify an intended or preferred model.

In Gulog [5, 6], Dobbie and Topor develop an elegant semantics for inheritance with overriding that addresses some of the unresolved problems in F-Logic. However, the elegance of their solution is achieved at the expense of a number of restrictions on the inheritance model. In particular, Gulog does not account for value inheritance and, more importantly, it separates the schema declarations from the data definitions thus avoiding the problems introduced by the dynamic subclassing capabilities of F-Logic.

In Orlog [9], Jamil and Lakshmanan developed a model for inheritance based on the notion of inheritance *withdrawal* to capture the idea of *user defined* inheritance and *conflict resolution* in multiple inheritance networks. One of the major shortcomings of this model is that overriding is captured via specification and hence is not deducible. However, by introducing the idea of *locality* of method clauses and the notion of *inheritability* in [8], the above handicap in Orlog is eliminated. However, the proposal in [8] achieved this functionality at the expense of the loss of dynamic subclassing capability.

Behavioral inheritance has been studied also in deductive formalisms like the *Ordered Theories* of [11], in modular languages such as Contextual Logic Programming [13, 14], *SelfLog* [3] and several others. In these proposals, an object is viewed as a set of rules (clauses) that represent the properties that hold at that object. Hence, although the functionalities of inheritance are the same as in object oriented systems, the resulting languages are essentially modular languages that retain the relational flavor of data peculiar to logic programming and, as such, differ from conventional object oriented languages, both syntactically and semantically.

6 Conclusion and Future Research

A desirable extension of the inheritance model we have presented would be to include inheritance with dynamic overriding in ways similar to those proposed for Gulog (and F-Logic, to that matter).

In Gulog, this feature is accounted for by resorting to interpretation structures that carry extra information needed (i) to identify the objects from which a value is inherited and (ii) to resolve the possible conflicts between the inheritance from different ancestors. Our current solution, based on static overriding, simplifies the treatment of overriding for set-valued methods and has also the potential benefit of allowing room for some form of static type checking. However, the extension to dynamic overriding appears to be necessary for several applications, notably for reasoning about inheritance hierarchies in artificial intelligence [16]. Our current work shows that the generalization of the framework we have presented in this paper should be smoothly accomplished by integrating our definition of i-stratification with the i-stratification condition proposed by Dobbie and Topor in [6].

As a further extension, we are currently studying the integration of our model with a corresponding model of structural inheritance. One of the challenges, in this extended framework, is to isolate and define an adequate relation between

method inheritance and overriding, as we have defined them here, with the properties of covariance and contravariance for the types of these methods' arguments and results.

References

1. H. Aït-Kaci and R. Nasr. Login: a logic programming language with built-in inheritance. *Journal of Logic Programming*, 3:182–215, 1986.
2. H. Aït-Kaci and A. Podelski. Towards a Meaning of LIFE. Technical Report 11, Digital Paris Research Labs, 1991.
3. M. Bugliesi. A declarative view of inheritance in logic programming. In K. Apt, editor, *Proc. Joint Int. Conference and Symposium on Logic Programming*, pages 113–130. The MIT Press, 1992.
4. M. Bugliesi and M. H. Jamil. A Stable Model Semantics for Behavioral Inheritance in Deductive Object Oriented Languages. Technical Report 6, Dip. di Matematica Pura ed Applicata, Univ. di Padova, 1994.
5. G. Dobbie and R. Topor. A Model for Inheritance and Overriding in Deductive Object-Oriented Systems. In *Sixteen Australian Computer Science Conference*, January 1988.
6. G. Dobbie and R. Topor. A Model for Sets and Multiple Inheritance in Deductive Object-Oriented Systems. Technical report, School of Computing and Information Technology, Griffith University, Nathan Qld 4111, Australia, January 1993.
7. Michael Gelfond and Vladimir Lifschitz. The Stable Model Semantics for Logic Programming. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Conference on Logic Programming*, pages 1081–1086. The MIT Press, 1988.
8. H. M. Jamil. *Semantics of Behavioral Inheritance in Deductive Object-Oriented Databases*. PhD Thesis (in preparation), Department of Computer Science, Concordia University, Canada, 1994.
9. H. M. Jamil and L. V. S. Lakshmanan. Orlog: A Logic for Semantic Object-Oriented Models. In *Proc. of the International Conference on Information and Knowledge Management, Baltimore, Maryland*, pages 584–592, November 1992.
10. 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).
11. E. Laesen and D. Vermeir. A Fixpoint Semantics for Ordered Logic. *Journal of Logic and Computation*, 1(2):159–185, 1990.
12. F.G. McCabe. *Logic and Objects*. Prentice Hall International, London, 1992.
13. L. Monteiro and A. Porto. A transformational view of inheritance in Logic Programming. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7th Int. Conference on Logic Programming*, pages 481–494. The MIT Press, 1990.
14. L. Monteiro and A. Porto. Syntactic and Semantic Inheritance in Logic Programming. In J. Darlington and R. Dietrich, editors, *Workshop on Declarative Programming*. Workshops in Computing, Springer-Verlag, 1991.
15. Teodor Przymusinski. Perfect Model Semantics. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Conference on Logic Programming*, pages 1081–1096. The MIT Press, 1988.
16. D. S. Touretzky. *The Mathematics of Inheritance Systems*. Morgan Kaufmann, Los Altos, CA, 1986.