1991

A Declarative View of Inheritance in Logic Programming

M. Bugliesi

Report Number:
91-078
A Declarative View of Inheritance in Logic Programming

M. Bugliesi
Dept. of Computer Science
Purdue University
W-Lafayette, Indiana 47906
e-mail: mb@cs.purdue.edu

November 1991

Abstract

In this paper we discuss a new semantic characterization of inheritance in logic programming. Our approach is inspired both by existing literature on denotational models for inheritance and by earlier work on the semantics of dynamic logic programs. We consider a general form of inheritance which incorporates notions such as overriding between inherited definitions and early/late binding for method invocation.

The framework for our study is a general stateless language in which objects are represented as logic theories. The underlying idea, which we borrowed from previous proposals, is to use functions over Herbrand sets - rather than simple Herbrand sets - to interpret such theories. On this basis, we then develop a new logical model of inheritance which enables us to define the standard notions of operational, model-theoretic and fixpoint semantics and to state the classical result about their confluence.

1 Introduction

The power of horn clause logic as a programming language was pointed out for the first time in [Kow79] and since then it has gained the interest of a still growing research community. The most appealing features of logic programming are to be appreciated in terms of its elegant semantic model as well as of its power as a programming language.

Logic Programming's semantic characterization is obtained as a byproduct of the standard approach to semantics in classical logic. A program is a set of first order formulæ - its clauses - and its meaning can be taken to be any model of these formulæ, that is any assignment for the constant, function and predicate symbols for which all the formulæ of the program are true. Furthermore, the simplified nature of horn clauses as a first order formalism allows one to restrict the choice of candidate models among interpretations built over Herbrand's universes. As such, the meaning of a program can be taken to be its minimal Herbrand model. The minimal herbrand model has two remarkable properties. For any program P, it coincides with the success-set of P i.e. with the set of all the atoms in the Herbrand Base for P which are logical consequences of P under SLD resolution. Secondly, it is also the least fixpoint of of the immediate consequences operator $T_P$ associated with P. The confluence of these model theoretic, operational and fixpoint notions has
greatly contributed to the popularity of this approach to semantics and of horn clauses as a programming language.

The importance of logic in the development of software systems was best summarized by Zaniolo in [Zan90]: “the rule based reasoning of logic, combined with the capability of database systems of managing and efficiently storing and retrieving large amounts of information could provide the basis to build the next-generation of knowledge base systems”. As a matter of fact, its use in the construction of knowledge base applications has promptly disclosed one major weakness of horn clause logic as a programming language. In fact, in spite of its declarativeness, logic programming turns out to not scale very well when it comes to designing practical applications. It’s units of abstraction – clauses and relations – appear to be too fine grained to support the development and the maintenance of large programs.

The need for a more structured approach to software development has motivated a wide research effort in the logic programming community during the last decade. The problem has been approached in (at least) two different ways.

Some authors have focused on conventional notions such as scope rules and structuring constructs and aimed at devising a logic programming paradigm with an embedded scope mechanism over clauses and predicate names.

Others, on the account of the experience gained in related fields, have addressed the issue of abstraction more directly. Motivated by the still increasing popularity of the Object-Oriented programming paradigm, their research is now being directed to the goal of integrating into a logic framework some of the distinguishing ideas of Object Orientation: abstraction, inheritance, message-passing and method overloading (late binding).

Besides its practical implications, the integration poses the interesting theoretical problem of providing a single model for two independent mechanisms: resolution on one side and inheritance on the other. From a logical point of view, this amounts – operationally – to incorporating inheritance into the deductive process of resolution and – declaratively – to finding an interpretation for inheritance systems that fits the standard notions of satisfiability and truth found in classical logic.

Earlier work on possible logical interpretations of inheritance has appeared in the literature. However, it is our belief that none of the existing proposals provides a complete and satisfactory solution to the problem. In fact some of them fail to capture the whole power of the notion of inheritance as defined within the O-O programming community. Others achieve a transformational view whereby inheritance is logically understood only from a strictly operational point of view.

The purpose of this paper is to discuss a new solution to the semantic problem. We consider a general form of inheritance which incorporates the standard notions of overriding between inherited definitions and early/late binding for method invocation. The framework for our study is a general stateless language in which objects are represented as logic theories.

The underlying idea, which we have borrowed from the existing literature on the subject, is to use functions over Herbrand sets – rather than simple Herbrand sets – to interpret our theories. On this basis, we then develop a new logical model of inheritance which enables us to define the standard notions of operational, model-theoretic and fixpoint semantics and to state the classical result about their confluence.

The rest of the paper is organized as follows. In section 2 we first describe the model
of inheritance we use throughout and we introduce some basic terminology and definitions. We then give an introductory overview of our approach. Sections 3, 4 and 5 are dedicated to the study of a series of small languages through which we analyze different aspects of inheritance and their semantic characterization. Finally in section 6 we discuss the relations of our approach with the existing literature and we derive some conclusions.

2 Inheritance

2.1 Preliminaries

The definition of inheritance we assume in this paper is stated in conformance with the one used in most Object-Oriented languages. An intuitive justification for such interpretation is contained in [WP89]. Inheritance is viewed as a mechanism for incremental/differential programming, i.e. “a mechanism for constructing new program components by specifying how they differ from the existing ones”. Differential programming is achieved “by using filters to modify the external behaviour of existing components” and accordingly, “to define a modified version of a function, we define a new function that performs some special operations and possibly calls the original function”. This is best illustrated in terms of the following diagrams, which we borrow from [WP89].

In the above diagram $P$ is a function, $M$ its modification and the arrows represent function invocation. The key to understanding this view of inheritance is to consider the nature of recursive definitions. When $P$ is a recursive function, a naive interpretation of the previous diagram is:

![Diagram](image)

As noted in [WP89], the modification only affects the external clients of the function and it does not modify the function’s recursive calls. To achieve the effect of a true modification of the original component the recursive calls in $P$ must be changed to refer to the modification $M$. Accordingly, diagram (1) should actually be redrawn as follows.

![Diagram](image)

The intuition behind recursion in functions can be rephrased in a more conventional Object-Oriented framework in terms of the use of self-references. Informally, the use of recursion in a function corresponds to the use of self-references in an object. In other words, the use of self-references is the way to achieve a recursive definition of an object. We illustrate this analogy by means of a simple example. Consider the following definitions:
We have two classes, `student` and `cs-student`, and two corresponding instances. Class `cs-student` is a subclass of `student` and redefines one of its superclass methods. The invocation `new class` returns an instance of `class` and the expression `object <- message` denotes the request to `object` to execute the `method` associated to `message`.

We are interested in the answers to the two following messages.

(a) `aStudent <- WhoAreYou`.
(b) `anotherStudent <- WhoAreYou`.

The result of evaluating (a) is straightforward. The message is sent to `aStudent` and the answer is: "I'm a Student".

Case (b) is more interesting. Now the result depends on what `self` refers to. We can restate the problem in terms of the diagrams introduced earlier in this section. Again we have two choices and two corresponding answers. The first is a naive interpretation of `self`.

![Diagram 1](image1)

"I'm a Student"

Notice that the modification has only partially affected the external behaviour of the original object and it has not affected the interpretation of `self`.

What we actually expect here is that `self` refer to the modified object. This is achieved through a different interpretation for `self` which is illustrated in the following diagram.

![Diagram 2](image2)

"I'm a CS-Student"

This diagram also shows the corresponding result of evaluating expression (b) and provides a justification for inheritance as a mechanism for deriving modified versions of recursive structures.

This characterization constitutes the main motivation to Cook's approach in [WP89]. In an independent study, Reddy [Red88] adopts a similar approach although with a weaker
position in favour of this view of inheritance. In [Red88], he develops an incremental study of different forms of inheritance where the interpretation given by diagram (2) is classified as dynamic inheritance — à la Smalltalk [GR83] — as opposed to the static mechanism espoused by languages like Simula-67 and depicted in diagram (1). The technical framework we present here is extensively based on Reddy's denotational definition. Only, we restate those ideas to fit into a logical framework.

2.2 Objects as Theories: a Logic view of Inheritance

Our approach to the declarative semantics of inheritance was actually inspired and first motivated by earlier work on the semantics of dynamic programs in logic programming. In [War84] D.S. Warren developed a modal approach to model updates in Prolog and defined the associated semantic framework in terms of a possible world semantics.

A similar approach was also undertaken in [MP89] to characterize the semantics of Contextual Logic Programming and in [Mi189] in the development of an extensive study of the logic of modules in Logic Programming.

The unifying features of all these approaches is the attempt to model the semantics of a dynamic program, and the choice of functions rather that sets as interpretations. As such, modal operators are functions over possible worlds in [War84], units are functions over situations in [MP89], and interpretations are functions from programs to the interpreting sets in [Mi189].

Our characterization of inheritance in Logic Programming steams from this very first idea of using functions to model interpretations. We assume a logic language that supports some form of modularity, i.e. it allows one to collect set of clauses into theories

and defines a protocol for goal invocation between theories. Theories are viewed in this context as the logical reinterpretation of the Object-Oriented notion of object. This choice is justified in term of the following argument which is due to F. G. McCabe [Cab88] and which we fully subscribe. From a logical point of view the integration of Object-Oriented and logic programming can be attempted along two guidelines. The first one is to incorporate all the O-O features into logic programming by essentially copying them into a logic system. This would be possible but not so interesting, at least theoretically, since some of those features, such as the idea of objects with state and of persistent data are inherently non logical. More interestingly, we can try to reinterpret Object-Orientation in the context of logic programming thus improving some of the Object-Oriented features and possibly ignoring the non logical ones.

From this point of view, an adequate way to look to an object is as a theory. An object is simply a collection of axioms which describe what is true about the object itself. The most natural approach to a model-theoretic semantics for a set of objects is then to associate with each object an interpretation. Inheritance is then viewed as a function which, given the interpretations of a set of objects returns the interpretation corresponding to the composition of the objects via inheritance. From this perspective inheritance is to be understood as a transformation over interpretations.

An alternate approach is to think of the objects themselves as denotations of functions over interpretations. More precisely, the fact that a given object $O_1$ inherits from object $O_2$ can be modeled by associating with $O_1$ a function that takes as input the interpretation for
and returns the interpretation associated with the composition of $O_1$ and $O_2$ through inheritance.

This is the approach we study in this paper. To present our semantics, as in [Red88], we describe a series of small languages: ObjectLog, InheritLog and SelfLog. ObjectLog is a simple logic language that offers support for objects and for method invocation. InheritLog extends ObjectLog to include static inheritance and SelfLog provides a different mechanism that captures the behaviour of dynamic inheritance.

3 ObjectLog

ObjectLog is a simple Logic Programming language extended with few basic mechanisms to support modularity. ObjectLog allows one to declare an object - a unit - as a collection of clauses and to request the evaluation of a goal to a specific object. The syntax is given below:

\[
\begin{align*}
\langle \text{unit def} \rangle &::= \text{unit } \langle \text{unit name} \rangle : \{\langle \text{clause list} \rangle\} \\
\langle \text{clause list} \rangle &::= \langle \text{clause} \rangle \langle \text{clause list} \rangle | \lambda \\
\langle \text{clause} \rangle &::= \langle \text{head} \rangle \langle \text{goal} \rangle \\
\langle \text{goal} \rangle &::= \langle \text{atomic goal} \rangle \langle \text{goal} \rangle | \lambda \\
\langle \text{atomic goal} \rangle &::= \langle \text{atom} \rangle | \langle \text{unit name} \rangle \langle \text{goal} \rangle
\end{align*}
\]

Atomic goals of the form $\langle \text{unit name} \rangle \langle \text{goal} \rangle$ will henceforth be referred to as message-goals.

In the declaration $\langle \text{unit name} \rangle : \{ \ldots \}$ we will denote with $\text{lui}$ the set of the ground instances of the clauses defined by $u$ following the style introduced in [MP89]. From [MP89] we also borrow the notation to define ObjectLog's operational semantics, which we give in terms of figures of the form:

\[
\begin{array}{c}
\text{assumption} \\
\hline
\text{conclusion} \\
\text{(condition)}
\end{array}
\]

ObjectLog's operational semantics is defined in terms of the inference relation $\vdash_\omega$ which is a special case of the more general inference rules used by Monteiro and Porto in [MP89] to define the operational semantics of Contextual Logic Programming. In the following definition we omit the subscript $\omega$ in $\vdash_\omega$ in order to make the rules easier to read. We also use the notation $\Box$ to stand for the empty goal formula and $\epsilon$ for the identity substitution.

\[
\begin{align*}
\langle 1 \rangle & \quad u \vdash_\epsilon \Box \\
\langle 2 \rangle & \quad u \vdash_\sigma G\theta \\
& \quad u \vdash_{\theta\sigma} g \\
& \quad (h : -G \in u \text{ and } \theta = \text{mgu}(g, h)) \\
\langle 3 \rangle & \quad u \vdash_\theta G_1, u \vdash_\sigma G_2 \theta \\
& \quad u \vdash_{\theta\sigma} G_1, G_2 \\
\langle 4 \rangle & \quad \hat{u} \vdash_\theta G \\
& \quad u \vdash_\theta \hat{u}.G
\end{align*}
\]

The interpretation of the above rules is straightforward. Rules $\langle 1 \rangle$, $\langle 2 \rangle$ and $\langle 3 \rangle$ simply model the standard operational semantics of logic programming with the only difference
that our programs are now units. As for rule (4), it simply says that evaluating a message-
goal ‘$u.G$’ in any unit just corresponds to evaluating the goal $G$ in the unit $u$, regardless
of the unit in which the message goal occurred. This behaviour is referred to as context
freeing in [MP89].

3.1 Model Theory

ObjectLog's model theory can be naturally derived from the model theoretic approach to
the semantics of Logic Programming. The main difference is that in this latter case we
interpret a single program whereas in ObjectLog we'd like to interpret several programs –
our units – simultaneously. An elegant solution to this problem was first shown in [Mil89].
Here we follow essentially the same approach, tailoring it to the case of Objectlog.

We define interpretations as functions $\mathcal{I} : \mathcal{U} \mapsto \mathcal{P} \mathcal{B}$ where $\mathcal{U}$ denotes a set of unit names and $\mathcal{B}$ is the Herbrand base built over a signature $\Sigma$ which is the union of the signatures for all the units in $\mathcal{U}$. Given a unit $u$ and an interpretation $i$ the set $i(u)$ will be referred to as the interpretation-set for $u$.

Using the idea of interpretation sets we can introduce the following notion of satisfiability in an interpretation. Given any interpretation $i \in \mathcal{I}$ and a set $S_i$ in the range of $i$, the satisfiability relation $\models_o$ for ground formulas is defined as follows:

1. $S_i \models_o G \iff G \in S_i$
2. $S_i \models_o h \leftarrow G \iff (S_i \models_o G \Rightarrow S_i \models_o h)$
3. $S_i \models_o G_1, G_2 \iff (S_i \models_o G_1 \text{ and } S_i \models_o G_2)$
4. $S_i \models_o u.G \iff i(u) \models_o G$

Again the idea is quite simple and intuitive. The set $S_i$ can be thought of as the set the interpretation $i$ associates with a given unit. A ground goal formula is true in such set if its conjuncts belong to the set. A (ground) clause is true if the clause's head is true whenever the body is true. A message goal $u.G$ is true if the goal $G$ is true in the interpretation set that $i$ associates with $u$.

We can finally introduce the definition of model. We use here the same idea adopted for interpretations, to distinguish between a model and a model-set.

Models. An interpretation $i$ is a model for a unit $u$ if every ground instance of a clause in $u$ is satisfied in the set $i(u)$. Formally,

$$i \text{ is a model for } u \iff \forall c \in |u| \ i(u) \models_o c.$$  

The set $i(u)$ is called a model-set for $u$.

3.2 Fixpoint Semantics

The definition of satisfiability we have just introduced provides a rather natural way of declaratively characterizing the semantics of ObjectLog. The meaning of a unit system $\mathcal{U}$ can be taken to be any interpretation $i$ which is a model for each of the units $\mathcal{U}$. We can in effect achieve a more precise definition of Objectlog semantics by identifying a special model – the minimal one – as the representative of all the models for $\mathcal{U}$. 

7
We'll next show how this model can be constructively defined in terms of a fixpoint computation. Before doing so, however, we need to introduce some more definitions and notation. Firstly, we enforce a further property for interpretations, by requiring that they be monotone, in the following sense. We define a partial order on $U$ by imposing

\[ u_1 \subseteq u_2 \iff |u_1| \subseteq |u_2| \]

for any two units in $U$. Then, for any given interpretation $i \in \mathcal{I}$ the following property holds true:

\[ \forall u_1, u_2 \ |u_1| \subseteq |u_2| \Rightarrow i(u_1) \subseteq i(u_2). \]

Secondly we introduce a more formal algebraic structure over the set of interpretations.

**Partial Ordering on Interpretations.** The structure $\langle \mathcal{P}(B), \subseteq \rangle$ is a complete partial order (a lattice). In this lattice the empty set $\emptyset$ is the bottom element ($\bot$) and $B$ is the top ($\top$); the join operator $\sqcup$ is defined in terms of set-union ($\cup$) and the meet operator $\sqcap$ is defined as set-intersection ($\cap$). The partial order $\subseteq$ induces a natural partial order $\sqsubseteq$ over interpretations, defined as follows. For any two interpretations $i_1$ and $i_2$,

\[ i_1 \sqsubseteq i_2 \iff \forall u \in U \ i_1(u) \subseteq i_2(u) \]

It easy to show that $\langle \mathcal{I}, \sqsubseteq \rangle$ is again a lattice with:

- (bottom) $\bot = \lambda u \ . \emptyset$
- (top) $\top = \lambda u \ . B$
- (join) $i_1 \sqcup i_2(u) = i_1(u) \cup i_2(u)$
- (meet) $i_1 \sqcap i_2(u) = i_1(u) \cap i_2(u)$

The idea to constructively define the model for each of the units of a program is again borrowed from [Mil89]. We attempt to interpret all the units simultaneously by building an interpretation $i$ such that, for each unit $u$ the set $i(u)$ is indeed a model-set for $u$. Such an interpretation will be the result of computing a least fixed point. To this purpose we now define the immediate consequence operator for our language, as the following transformation $\tau_o : \mathcal{I} \mapsto \mathcal{I}$.

\[
\tau_o(i)(u) = \{ A \mid A \in |u| \text{ or the is a clause } A : \neg G \in |u| \text{ and } i(u) \models_o G \}
\]

Notice that the case of unit clauses has been explicitly singled out. We could actually have treated it as a special case of clauses under the assumption that the condition $S \models_o G$ holds true for any set $S$.

It's easy to see that if $i$ is an interpretation, $\tau_o(i)$ is also an interpretation. Following Miller's argument in [Mil89], we can also prove the following properties for interpretation and for the mapping $\tau_o$.

\begin{itemize}
  \item **(P1)** For any formula $G$ and any interpretation $i$, if $G$ is true in $i$ then it is also true in any interpretation more defined than $i$.
    \[ i_1 \sqsubseteq i_2 \Rightarrow (\forall u \in U \ i_1(u) \models_o G \Rightarrow i_2(u) \models_o G) \]
\end{itemize}
Let $i_1 \subseteq i_2 \subseteq \ldots$ be a sequence of interpretations. Then for any unit $u$ and formula $G$ if $\bigcup_{j=1}^{\infty} i_j(u) \models_o G$ then there exists a value $k$ such that $i_k(u) \models_o G$.

$(P3)$ $\tau_o$ is monotonic: $i_1 \subseteq i_2 \Rightarrow \tau_o(i_1) \subseteq \tau_o(i_2)$.

$(P4)$ $\tau_o$ is continuous: $\bigcup_{j=1}^{\infty} \tau_o(i_j) = \tau_o(\bigcup_{j=1}^{\infty} i_j)$.

The proof of the above properties can be obtained by simply generalizing the proofs reported in [Mil89]. The generalization is justified by the fact that those proofs are indeed independent of the semantics of the implication operator $\supset$ in terms of which Miller defines the composition of different program components (see [Mil89] for details). As matter of fact, the fixpoint of $\tau_o$ will denote different functions depending on the semantics of $\supset$, but the choice of different meanings for $\supset$ does not affect the continuity of $\tau$ and the above properties for interpretations.

Existence of a model. The existence of a model for any unit $u$ is derives immediately by observing that the base $B$ is a model-set for any unit (see [Llo84]). Accordingly,

$$i_\tau = \lambda u. B \text{ is a model for any unit } u$$

Existence of a minimal model. Given the set of all the models $\{i_1, \ldots, i_k, \ldots\}$, there exists a minimal model defined as follows:

$$i_M = \bigcap_{n=1}^{\infty} i_n$$

Correspondingly, for any unit $u \in \mathcal{U}$, there is a minimal model-set obtained as $i_M(u)$.

We are now ready to state the two standard equivalence results between the declarative and operational semantics and between the fixpoint construction of the set-theoretic characterization of minimal model.

**Theorem 1** Equivalence of model theory and operational semantics. Let $i_M$ be the minimal model for a unit $u$ and let $M = i_M(u)$ be the associated minimal model-set. Then the set $M$ coincides with the set of the ground atoms provable in $u$. Formally, for any ground atom $G$:

$$M \models_o G \iff u \vdash_o G$$

**Theorem 2** Adequacy of the Fixpoint Semantics versus the Model Theory. Let $M_u$ be the minimal model-set for a unit $u$. Then $M_u$ can be constructively defined in terms of the fixpoint of $\tau_o$. Namely:

$$M_u = \bigcup_{n=1}^{\infty} \tau_o^n(i_\bot)(u)$$

### 3.3 Examples

We conclude the study of ObjectLog with two examples which illustrate the nature of the iterative computation of the fixpoint.
Example 1. Consider the following unit declarations:

\[
\begin{align*}
\text{unit } u_1 & : [a:-u_2.b \ c] \\
\text{unit } u_2 & : [b:-c]
\end{align*}
\]

According to the rules we have defined for \( i_o \), it’s easy to see that the success-sets for \( u_1 \) and \( u_2 \) are respectively the set \( \{ c \} \) and the empty set \( \emptyset \). The expected models for \( u_1 \) and \( u_2 \) are therefore: \( M_{u_1} = \{ c \} \) and \( M_{u_2} = \emptyset \).

We now show that the fixpoint computation converges and that the resulting interpretation computes \( M_{u_1} \) and \( M_{u_2} \) as the models for \( u_1 \) and \( u_2 \).

From the definition of \( i_o \) it follows that \( i_o(u_1) = i_o(u_2) = \emptyset \). The next steps iteratively compute \( \tau_o^n \) \((n = 1, 2, \ldots)\) until the sequence reaches the fixpoint \( \tau_o^k \) for which \( \tau_o^k = \tau_o^{k+1} \).

\[
\begin{align*}
(n = 1) \quad & \tau_o^1(i_o)(u_1) = \{ c \} ; \tau_o^1(i_o)(u_2) = \emptyset \\
(n = 2) \quad & \tau_o^2(i_o)(u_1) = \{ c \} ; \tau_o^2(i_o)(u_2) = \emptyset
\end{align*}
\]

As such, \( \tau_o^1(i_o) = \text{fix.} \tau_o \) is the fixpoint and, as expected \( \text{fix.} \tau_o(u_1) = M_{u_1} = \{ c \} \) and \( \text{fix.} \tau_o(u_2) = M_{u_2} = \emptyset \).

Example 2. We now make the previous unit definitions a little more complex by letting the two units request the evaluation of a goal to each other. Namely:

\[
\begin{align*}
\text{unit } u_1 & : [a:-u_2.b \ c] \\
\text{unit } u_2 & : [b:-u_1.c]
\end{align*}
\]

The expected models for \( u_1 \) and \( u_2 \) are now \( M_{u_1} = \{ a, c \} \) and \( M_{u_2} = \{ b \} \). Again the computation of the fixpoint converges to the expected interpretation. It uses rule (4) of the satisfiability relation to handle the mutual invocation of the message-goals in \( u_1 \) and \( u_2 \).

Again \( i_o(u_1) = i_o(u_2) = \emptyset \). Then the computation proceeds as follows:

\[
\begin{align*}
(n = 1) \quad & \tau_o(i_o)(u_1) = \{ c \} \quad \text{and} \quad \tau_o(i_o)(u_2) = \emptyset \\
(n = 2) \quad & \text{To compute } \tau_o^2(i_o)(u_1) \text{ we must determine whether } \tau_o(i_o)(u_1) \models_o u_2.b \text{ holds true. Similarly, to compute } \tau_o^2(i_o)(u_2) \text{ we must check whether } \tau_o(i_o)(u_1) \models_o c \text{ holds. Notice that } \tau_o(i_o)(u_1) \models_o u_2.b \text{ if and only if } \tau_o(i_o)(u_2) \models_o b. \text{ Now, since } \tau_o(i_o)(u_2) \not\models_o b, \text{ it follows that } b \not\in \tau_o^2(i_o)(u_1). \text{ Furthermore, since } \tau_o(i_o)(u_1) \models_o c, \text{ we obtain:} \\
& \tau_o^2(i_o)(u_1) = \{ c \} \quad \text{and} \quad \tau_o^2(i_o)(u_2) = \{ b \}
\end{align*}
\]

\[
\begin{align*}
(n = 3) \quad & \text{From the result obtained at the previous step, it is easy to see that} \\
& \tau_o^3(i_o)(u_1) = \{ a, c \} \quad \text{and} \quad \tau_o^3(i_o)(u_2) = \{ b \}.
\end{align*}
\]

Finally, since \( \tau_o^3(i_o) = \tau_o^4(i_o) \), then \( \tau_o^4(i_o) \) is the fixpoint and we obtain the expected result:

\[
\begin{align*}
\text{fix.} \tau_o(u_1) = M_{u_1} = \{ a, c \} \quad \text{and} \quad \text{fix.} \tau_o(u_2) = M_{u_2} = \{ b \}.
\end{align*}
\]

This concludes the study of ObjectLog. It’s semantic characterization is indeed rather smoothly derived as an extension of the fixpoint semantics for logic programming to handle a fairly simple support for modularity. Considering inheritance as a further composition mechanism will result into a more powerful language as well as into a more interesting semantic characterization.
4 InheritLog

InheritLog is the second in the series of languages we study in this paper. It extends ObjectLog to support a simple form of inheritance. Syntactically this amounts to adding a new production for unit declaration:

\[
\text{unit def} ::= \text{unit } (\text{unit name}) \text{ inherit } (\text{unit name}) : [(\text{clause list})]
\]

with the obvious meaning. InheritLog's operational semantics is again borrowed from [MP89]. The language actually espouses the same kind of static inheritance as the one found in Contextual Logic Programming (CxP). The difference from that case is that inheritance in our language is defined in terms of static unit hierarchies whereas in CxP the rules for unit composition are inherently dynamic. Furthermore, message-passing is achieved in InheritLog through context freeing as opposed to the use of context extension found in CxP.

InheritLog's operational semantics is defined in terms of the inference relation \( \vdash_i \). We first introduce some further notation. Again, in order to make the rules easier to read, in the following definition we omit the subscript \( i \) in \( \vdash_i \).

\[
\begin{align*}
\langle 1 \rangle & \quad c \vdash_c \Box \\
\langle 2.a \rangle & \quad \frac{[u | c] \vdash_\sigma G \theta}{[u | c] \vdash_{\theta \sigma} g} \quad (h: G \text{ is a clause of } u \text{ and } \theta = \text{mgu}(g, h)) \\
\langle 2.b \rangle & \quad \frac{c \vdash_\sigma G \theta}{[u | c] \vdash_{\theta \sigma} G} \quad (\text{pred}(G) \not\in \| u \|) \\
\langle 3 \rangle & \quad \frac{c \vdash_\theta G_1 \quad c \vdash_\theta G_2 \theta}{c \vdash_{\theta \sigma} G_1, G_2} \\
\langle 4 \rangle & \quad \frac{\hat{c} \vdash_\theta G}{c \vdash_\theta \hat{u}.G} \quad (\text{closure}(\hat{u}) = \hat{c})
\end{align*}
\]

A few remarks about our notation are worthwhile here. For any atomic formula of the form \( G = p(t_1, \ldots, t_n) \), the predicate symbol \( p \) is referred to as the name of \( G \). The notation \( \text{pred}(G) \) is also meant to refer to the name of \( G \). We say that a unit defines a predicate name \( p \) if it contains (at least) a clause whose head's name is \( p \). We denote with \( \| u \| \) the set of predicate names defined by \( u \).

Also, we represent unit hierarchies as lists (contexts according to Monteiro and Porto's terminology). As such, the hierarchy

\[ u_n \text{ inherit } u_{n-1} \ldots \text{ inherit } u_1 \]

is represented as the context \( [u_n, \ldots, u_1] \), where \( u_n \) is the top unit. The closure operator occurring in the above rules can be then interpreted as follows: for any unit \( u \), closure(\( u \)) denotes the context \( [u | c] \) where \( c \) is the context associated with the sequence of \( u \)'s ancestors.

The above rules can now be interpreted as follows. Rule \( \langle 4 \rangle \) embeds the standard semantics of message passing for method invocation. Evaluating the goal \( u.G \) corresponds to evaluating \( G \) in the context of the unit hierarchy whose tip mode is \( u \).
Rules (2.a) and (2.b) define the standard look-up semantics for method evaluation. A goal $G$ is evaluated in a unit $u$ by first attempting all the local definitions, if any. If no definition for $G$ is found in $u$, and only in this case, the inherited definitions are eventually used. Notice that this implies that any definition for $G$ found in the ancestors of $u$ is overridden by $u$’s local definition (if any). Notice also that the search for a matching clause for $G$ always starts from the unit where $G$ is invoked. In other words, the lookup for a matching clause is performed independently of the the unit on behalf of which the current method is being executed (i.e. the original receiver of the message).

This kind of static inheritance is precisely the one defined in [MP89] and addressed elsewhere ([BLM]) as eager (early) binding for method determination.

4.1 Model Theory

The approach to the model-theoretic semantics of InheritLog needs reconsidering the nature of interpretations and of the partial order $\langle \mathcal{S}, \subseteq \rangle$ we defined for ObjectLog. The point is that now the set of provable atoms in a given unit does not depend only on the unit itself but also on the unit’s ancestors. A degree of interdependency between units was also to be found ObjectLog embedded, in that case, only in what we called message goals of the form $u.G$. The case of InheritLog is different and subtler since such interdependencies are now also established implicitly via inheritance. As such, an atom might be provable in a unit just because the unit inherits its proof, or part of it, from its ancestors.

This brings up the following problem. Consider the declarations:

```
unit u : [ q(a) q(b) ]
unit u_1 inherit u : [ p(x):-q(x) ]
unit u_2 inherit u : [ p(x):-q(x) q(a) ]
```

According to the definition of $\Gamma_i$, the success-set for $u_1$ is $S_1 = \{ p(a), q(a), q(b), p(b) \}$ and the corresponding set for $u_2$ is $S_2 = \{ p(a), q(a) \}$. Any reasonable semantics will then have to consistently associate models with $u_1$ and $u_2$. As a matter of fact, we expect that $M_{u_1}$ coincides with $S_1$ and $M_{u_2}$ with $S_2$.

Now, if we assume our previous definition of the partial order over units, any function which associates $M_{u_1}$ and $M_{u_2}$ with $u_1$ and $u_2$ is not monotonic. In fact, from their definitions, it follows that $u_1 \subset u_2$ whereas $M_{u_1} \not\supset M_{u_2}$.

We can tailor our previous framework to conform to the extended case of InheritLog by assuming a different ordering over units. A flat ordering with a bottom element $u_\bot$ will serve our purposes. The new order relation, which we denote again with $\subseteq$, is such that for any two units $u_1$ and $u_2$, $u_1 \subseteq u_2$ if and only if $u_1 = u_\bot$. The unit $u_\bot$ is the least defined unit and can be thought of as the unit denoting an empty set of clauses. As such, $\forall i \in \mathcal{S} : i(u_\bot) = \emptyset$. We also introduce a function $p : \mathcal{U} \mapsto \mathcal{U}$ which, given a unit name, returns its direct ancestor (possibly $u_\bot$).

Given an interpretation $i$, InheritLog’s satisfiability relation for ground formulas is defined as follows.
The key to understanding InheritLog’s declarative semantics is provided by rules (1) and (2).

An atomic formula $G$ is true in the interpretation-set $i(u)$ if $u$ defines the predicate name of $G$, and $G$ belongs to $i(u)$. Conversely, if $u$ does not define the name of $G$, $i(u) |=_i G$ holds if $G$ belongs to interpretation-set of the closest ancestor of $u$ which defines its name.

A ground clause $h: -G$ is true in $i(u)$ if the predicative name of $h$ is among those defined by $u$, and $h$ is true in $i(u)$ whenever $G$ is true in $i(u)$. Conversely, if $u$ does not define the name of $h$, then $h: -G$ is true in $i(u)$ if and only if it is true in the interpretation-set of the immediate ancestor of $u$, namely $i(p(u))$.

There are two important remarks that are worth mentioning here. According to rule (1), the truth value of an atomic formula $G$ in a unit $u$ is determined by the value of $G$ in the closest of $u$’s ancestors in which $G$’s name is defined. In other words, the definition of $G$ in $u$ overrides any other definition found in any of $u$’s ancestors for the same predicate name.

As for rule (2) the important thing to mention is that, if $u$ does not contain any definition for the clause head ($\text{pred}(h)$), then any clause for the body $G$ which might occur in $u$ does not contribute to establish the truth of $h: -G$. Notice in fact that $h: -G$ is to be satisfied in $i(p(u))$. This provides the declarative counterpart of the eager binding mechanism introduced in the definition of InheritLog’s operational semantics.

4.2 Fixpoint Semantics

We now turn to the definition of the transformation $\tau_i : \mathcal{S} \mapsto \mathcal{S}$ for InheritLog. For this purpose we’ll assume that each unit is defined as the heir of a parent unit. Accordingly, a unit definition of the form unit $u : [...]$ will be simply considered as a shorthand for the extended notation:

$$\text{unit } u \text{ inherit } u_1 : [...]$$

We also introduce a further operator, $\Diamond_{\Sigma} : \mathcal{P}(B) \mapsto \mathcal{P}(B)$ which provides the formal device for modeling, in a set-theoretic sense, the overriding of predicates in the fixpoint computation of the model associated to a unit.

**Definition 4.1** Let $\Sigma$ denote any set of the predicate names of the basis $B$. Let $S_1$ and $S_2$ be two sets in $\mathcal{P}(B)$. Then:

$$S_1 \Diamond_{\Sigma} S_2 = S_1 \cup \{ t \in S_2 \mid \text{pred}(t) \notin \Sigma \}$$

As mentioned before, $\Diamond$ models on interpretations, the type of overriding between inherited definitions embedded in both the relations $\models_i$ and $|=_i$. This is achieved through a suitable
choice of the signature \( \Sigma \). For any given \( i \), if \( S_1 = i(u_1) \) and \( S_2 = i(u_2) \), then by choosing \( \Sigma = \| u_1 \| \), the set \( S_1 \diamond \Sigma S_2 \) denotes the set whose elements are the elements of \( S_1 \) united with all the elements of \( S_2 \) which are not overridden by those in \( S_1 \). Accordingly, if \( S_2 = \{ p(a), q(b) \} \) and \( S_1 = \{ q(a) \} \), then \( S_1 \diamond \| u_1 \| S_2 \) denotes the set \( \{ p(a), q(a) \} \).

We can now introduce the new definition of the immediate consequence operator \( \tau_i \) for InheritLog. The idea is that for any interpretation \( i \), the set \( i(u) \) should contain all the true atoms in the ancestors of \( u \) whose predicate symbols don’t collide with the ones in the predicative signature of \( i(u) \), united with all the atoms that are provable in \( u \) using the interpretation of its ancestors as the initial set of hypotheses.

We introduce here two definitions, the first more intuitive and given inductively on the structure of a unit’s hierarchy. For any unit \( u \) let \( \text{inherit}(u) = \cup \tau_i(\_)(u) = \text{fix.} \tau_i(p(u)) \). Then:

\[
\tau_i(i)(u) = \{ A \mid A \in \| u \| \text{ or } A : -G \in \| u \| \text{ and } i(u) \models G } \diamond \| u \| \text{ inherit}(u)
\]

Notice that, although intuitive, this definition is not technically correct since it assumes that no mutual recursion is involved in the definition of units belonging to the same hierarchy – this allows us to use \( \text{inherit}(u) \) in the definition of \( \tau_i(i)(u) \).

The following definition allows us to drop this restriction.

\[
\tau_i(i)(u) = \{ A \mid A \in \| u \| \text{ or } A : -G \in \| u \| \text{ and } i(u) \models G \} \diamond \| u \| i(p(u))
\]

The interpretation of \( u \) and of all its ancestors can be now carried on simultaneously and the resulting interpretation, the fixpoint of \( \tau_i \), actually interprets all the units.

If not on the monotonicity of \( \tau_i \) we can already argue that the interpretation-sets computed at each step of the iterative computation of the fixpoint form an increasing sequence. In other words:

\[
\forall n \forall u \in U \quad \tau^n_i(i_\_)(u) \subseteq \tau^{n+1}_i(i_\_)(u)
\]

This, again informally derives from the fact that, at the \( n \)-th step, we include in \( \tau^n_i(i_\_)(u) \) only those atoms that will effectively be included in the set that \( \text{fix.} \tau_i \) associates with \( u \). This is accomplished by the use of \( \diamond \) that, at each step, excludes from the interpretation of \( p(u) \) all those atoms whose predicate symbols collide with the ones being defined by \( u \).

We can now state the two standard results that establish the connection between the fixpoint construction of the minimal model and the operational semantics for InheritLog.

**Theorem 3** Equivalence of model theory and operational semantics. Let \( i_M \) be the minimal model for a unit \( u \) and \( M = i_M(u) \) be the corresponding minimal model-set. Then for any ground atom \( G \):

\[
M \models_i G \iff u \vdash_i G
\]

**Theorem 4** Adequacy of the Fixpoint Semantics versus the Model Theory. Let \( M_u \) be the minimal model-set for a unit \( u \). Then \( M_u \) can be constructively defined in terms of the fixpoint of \( \tau_i \).

\[
M_u = \cup_{n=1}^{\infty} \tau^n_i(i_\_)(u)
\]
We conclude the discussion on InheritLog with an example that shows the construction of the minimal model for the three units considered at the beginning of this section.

**Example 2.** Consider the following unit declarations:

- `unit u: [ q(a) q(b) ]`
- `unit u_1 inherit u: [ p(x): -q(x) ]`
- `unit u_2 inherit u: [ p(x): -q(x) q(a) ]`

By definition \( i_\perp(u) = i_\perp(u_1) = i_\perp(u_2) = \emptyset \). It’s also easy to see that the fixpoint computation for \( u \) converges at the first step. Namely,

\[
\tau^1_\perp(i_\perp)(u) = \tau^2_\perp(i_\perp)(u) = \tau^3_\perp(i_\perp)(u) = \{q(a), q(b)\}
\]

The corresponding steps for \( u_1 \) and \( u_2 \) derive by simply applying the definition of \( \tau_i \). They also show how the use of \( \diamond < > \) supports a monotonic increase of the sequence \( \tau^n_i \) by, preventing, at each step, inserting any element which would eventually be overridden.

\[
(n = 1) \quad \begin{cases} 
\tau^1_\perp(i_\perp)(u_1) = \emptyset \diamond_{\|u_1\|} \emptyset = \emptyset \\
\tau^1_\perp(i_\perp)(u_2) = \{q(a)\} \diamond_{\|u_2\|} \emptyset = \{q(a)\} \\
\tau^2_\perp(i_\perp)(u_1) = \emptyset \diamond_{\|u_1\|} \tau^1_\perp(u) = \{q(a), q(b)\} \\
\tau^2_\perp(i_\perp)(u_2) = \{q(a), p(a)\} \diamond_{\|u_2\|} \tau^1_\perp(u) = \{q(a), p(a)\} \\
\tau^3_\perp(i_\perp)(u_1) = \{q(a), q(b), p(a), p(b)\} \diamond_{\|u_1\|} \tau^2_\perp(u) \\
\tau^3_\perp(i_\perp)(u_2) = \{q(a), q(b), p(a), p(b)\}
\end{cases}
\]

It’s easy to see that \( \tau^3_\perp(i_\perp) \) is indeed the fixpoint of \( \tau_i \). Notice furthermore that such a fixpoint computes for both \( u_1 \) and \( u_2 \) precisely the models mentioned earlier in this section.

## 5 SelfLog

SelfLog represents the final step of our construction. SelfLog is not properly an extension of InheritLog, it just embeds the different form of inheritance which we called *dynamic inheritance*. The difference becomes immediately apparent after considering SelfLog’s operational semantics. We use here the same scheme used for InheritLog. As in that case, we represent a unit-hierarchy as a context.

SelfLog’s operational semantics is defined in term of the inference relation \( \vdash_s \). The subscript \( s \) is omitted as usually to make the definition more readable. In the following rule we also use the shorthand \( u_g \) to denote the closest ancestor of the unit \( u \) which defines the predicate name of the atomic formula \( g \).
The key difference from the corresponding definition of InheritLog is in rule (2) which replaces rules (2.a) and (2.b) in the definition of $\Gamma_\iota$. Notice that now, regardless of the unit where the invocation of a message goal occurs, the search for a matching clause starts now from the original receiver of the message (which is just the top unit of the context).

We can interpret rules (1), (2), (3) and (4) according to an Object-Oriented perspective by assuming that that each goal invocation $G$ in which no explicit unit is mentioned actually corresponds to an explicit method invocation of the form $\text{self}.G$. Then the difference between InheritLog and SelfLog is that in InheritLog $\text{self}$ always refers to the unit where the method is being invoked, whereas in SelfLog, it refers to the unit that was the receiver of the last message. Let's reconsider the example of the student hierarchy introduced earlier in the paper, restated now in terms of the logic language we have developed. This will help to clarify the difference between the two mechanisms.

```
unit student : [ whoAmI(aStudent) \\
                whoAreYou(x):-whoAmI(x) ]
unit cs_student inherit student : [ whoAmI(aCsStudent) ]
```

We want to look at the evaluation of the query $cs\_student.whoAreYou(x)$. It's easy to see that, if we assume InheritLog's operational semantics ($\Gamma_\iota$), the answer to such a query is the binding $\{x \leftarrow aStudent\}$, whereas according to the rules for SelfLog, the evaluation of the query results into the substitution $\{x \leftarrow aCsStudent\}$.

This is precisely the same effect we've seen earlier in the paper and is a consequence of the choice between the two different forms of inheritance we called respectively static and dynamic.

### 5.1 A declarative view of dynamic inheritance

This new interpretation of inheritance has a remarkable impact on the semantic framework we've been developing.

We have so far characterized the semantics of units in terms of sets. Each unit denotes, through an interpretation, the subset of the basis $B$ given by its interpretation-set. We have also shown how to define the model-set of a unit in terms of the fixpoint of the functional $\tau : \mathfrak{S} \mapsto \mathfrak{S}$. Given, $i_M$, the fixpoint of $\tau$, the minimal model-set for a unit $u$ is simply obtained as the set $i_M(u)$.

This has been possible due to the properties of the languages studied so far. In ObjectLog as well as in InheritLog a unit declaration contained all the information needed to determine its interpretation-set: the list of the unit's clauses and its immediate ancestor.
Now the scenario has changed. The semantics of a unit does not depend only on that of its ancestors but also on that of its heirs. Indeed, a unit’s behaviour is a function of the interpretation of its heirs. This appeared explicitly in the previous example which showed how the definition of the unit cs_student contributes to modify the behaviour of its immediate ancestor student.

The natural consequence is that the denotation of a unit will be a function over sets rather than a simple set. Intuitively, a unit will denote a function that accepts the interpretation of its heir as a parameter and produces a new set that interprets the composition of the two units via inheritance.

With this picture in mind, we can now turn to the definition of SelfLog’s declarative semantics.

5.2 Model Theory

The first step is a new definition of interpretations and of the denotation of unit names. As already mentioned, in ObjectLog as well as in InheritLog unit names denoted, through interpretations, subset of the herbrand base. Now, following the idea mentioned above, we change such denotations to functions over sets. Accordingly, interpretations are defined as follows:

\[ \mathfrak{I} : \mathcal{U} \mapsto \mathcal{P}(B) \mapsto \mathcal{P}(B) \]

and the denotation of a unit \( u \) through an interpretation \( i \) is a function \( i(u) \in [\mathcal{P}(B) \mapsto \mathcal{P}(B)] \). We will henceforth use the notation \( \Psi_B \) to refer to the set \( [\mathcal{P}(B) \mapsto \mathcal{P}(B)] \). The partial order \( (\mathcal{P}(B), \subseteq) \) induces the usual partial order over \( \Psi_B \). Namely,

\[ \psi_1 \subseteq \psi_2 \iff \forall x \in \mathcal{P}(B) \psi_1(x) \subseteq \psi_2(x) \]

The algebraic structure \( (\Psi_B, \subseteq) \) is a lattice with bottom and top element \( \perp_\Psi \) and \( \top_\Psi \) respectively. The definition of \( \perp_\Psi \) and \( \top_\Psi \) as well as of the join and meet operators over \( \Psi_B \) is straightforward and it is given below.

\[ \perp_\Psi = \lambda \omega . \emptyset \]
\[ \top_\Psi = \lambda \omega . B \]
\[ (\psi_1 \cup \psi_2)(x) = \psi_1(x) \cup \psi_2(x) \]
\[ (\psi_1 \cap \psi_2)(x) = \psi_1(x) \cap \psi_2(x) \]

The corresponding partial order over interpretation is naturally defined in terms of \( \subseteq_\Psi \) as follows.

\[ i_1 \subseteq i_2 \iff \forall u \in \mathcal{U} \ i_1(u) \subseteq_\Psi i_2(u) \]

Finally, it’s immediate to see that \( (\mathfrak{I}, \subseteq) \) is again a lattice with bottom and top element:

\[ i_\perp = \lambda u . \perp_\Psi \]
\[ i_\top = \lambda u . \top_\Psi \]

Interpretation sets. We then introduce the notion of interpretation-sets which – in much the same spirit as in ObjectLog – provide the endpoint of the semantic characterization of a unit.

17
For any interpretation $i$ and unit $u$ let $\psi_u$ denote the function that $i$ associates with $u$. We say that for any set $\omega$, $\psi_u(\omega)$ is the $\omega$-interpretation-set for $u$.

The notion of $\omega$-interpretation is not new in logic programming and has been introduced elsewhere to characterize the semantics of open logic programming [BLM91, BGLM91]).

**Satisfiability.** The idea behind SelfLog's declarative semantics is that the denotation of a unit is a function which lets the set computed by the denotation of its heirs override the set of provable atoms in the unit itself.

Furthermore, since each unit may have more than one direct heir, its behaviour will depends on which heir is being considered at the different stages of the computation. These considerations apply then recursively to the unit's heirs.

We finally come to the crucial question: how and when can we break this chain of unit interdependencies and determine an $\omega$-interpretation-set for a unit and its heirs? The answer is rather intuitive. Indeed, it's the occurrence of a unit in a message goal that determines which part of its hierarchy is being used at the different stages of the computation. If $[u_n, \ldots, u_1]$ is the closure of a unit $u_n$ then, when evaluating the message-goal $u_n.G$, we are considering the interpretation of $u_1$ as modified by $u_2$ and recursively by $u_n$. At this particular stage, regardless of which heirs $u_n$ might have, we are closing $u_n$'s interpretation and that of its ancestors. Accordingly, none of $u_n$'s heirs will contribute to computing the interpretation-set associated with $u_n$ at this specific stage. Technically, if $\psi_{u_n}$ is the denotation of $u_n$, this simply amounts to computing the interpretation-set resulting from the application $\psi_{u_n}(\emptyset)$. The effect of closing $u_n$ will then propagate upwards to the root of the hierarchy, thus yielding an $\omega$-interpretation-set for all the units occurring such a hierarchy.

This intuitive argument motivates the following new definition of satisfiability. Let $\omega$ be any subset $\mathcal{P}(B)$, $i$ be an interpretation and $u$ a unit name. Again we use the notation $u_G$ with the same intended meaning as in the case of InheritLog.

\begin{align*}
(1) & \quad i(u)\omega \models \_ G \iff G \in i(u_G)\omega \\
(2) & \quad i(u)\omega \models h: G \iff i(u)\omega \models G \Rightarrow i(u)\omega \models h \\
(3) & \quad i(u)\omega \models G_1, G_2 \iff i(u)\omega \models G_1 \text{ and } i(u)\omega \models G_2 \\
(4) & \quad i(u)\omega \models \_ \_ u.G \iff i(\_\_ u)\emptyset \models G
\end{align*}

**Models and Model-sets.** We can now introduce the notion of models and model-sets. Let $i$ be an interpretation, $u$ a unit and $\omega$ a subset of $\mathcal{P}(B)$.

Then the transformation $\psi_u$ is an $\omega$-model for $u$ if and only if $\psi_u(\omega) \models c$ for any clause $c \in |u|$. The set $\psi_u(\omega)$ is the corresponding $\omega$-models-set for $u$.

**Minimal Models.** The notion of minimal model comes as a byproduct of the previous definition. First notice that for any $\omega$, the existence of an $\omega$-model is again guaranteed by the fact that $\Gamma_\Psi$ is an $\omega$-model-set for any unit. For any $\omega$ we can therefore choose the minimal $\omega$-model for a unit, by taking the $\text{glb}(\Gamma)$ of all the unit's $\omega$-model-sets.
5.3 Fixpoint Semantics

The final step of SelfLog's semantic characterization is given, as usually, by the fixpoint characterization of the minimal model. The definition of the transformation $\tau$ is modified now to conform to the new domain $\mathfrak{S} = [\mathcal{U} \mapsto \Psi \mathcal{G}]$ for interpretations. The immediate operator consequence $\tau_s$ for SelfLog is now defined as follows.

$$\tau_s(i)(u)(\omega) = \{ A \mid A \in \omega \Diamond_{|u|} \] u \] or
\quad A :- G \in [u] \text{ and } \omega \Diamond_{|u|} i(u)(\omega) \models \ G \} \Diamond_{|u|} i(p(u))(i(u)(\omega))$$

It's worth comparing this definition with the corresponding one for InheritLog. Notice that now, given $\omega$, the interpretation-set $\tau_s(i)(u)(\omega)$, includes:

- $\omega \Diamond_{|u|} [u]$, the ground instances of the unit clauses of $u$ ($[u]$) possibly overridden by the atoms contained in $\omega$, united with
- all the immediate consequences of the set of hypotheses $\omega \Diamond_{|u|} i(u)(\omega)$,

In both cases, the use of $\Diamond$ models the dependency of the interpretation of $u$ on $\omega$. This effect is then recursively propagated to the interpretation of $p(u)$ by means of the application $i(p(u))(i(u)(\omega))$.

We can now state the two usual results that establish the connection between the fixpoint construction of the minimal model and the operational semantics for SelfLog.

\textbf{Theorem 5} Equivalence of model theory and operational semantics. Let $i_M$ be an interpretation such that $i_M(u) = \psi_u$ is the minimal $\omega$-model for $u$. Then, the corresponding $\Diamond$-model-set $\psi_u(\Diamond)$ coincides with the set of provable atoms in $u$. Formally

$$\forall u \in \mathcal{U} \psi_u(\Diamond) \models_G \iff u \models_G$$

\textbf{Theorem 6} Adequacy of the Fixpoint Semantics versus the Model Theory. Let $i$ be an interpretation and $i(u) = \psi_u$ be the minimal $\omega$-model for a unit $u$. Then:

$$\forall \omega \psi_u(\omega) = \bigcup_{n=1}^{\infty} \tau_i^n(i_u)(u)(\omega)$$

As an immediate corollary of the above theorem, $\psi_u(\Diamond) = \bigcup_{n=1}^{\infty} \tau_i^n(i_u)(u)(\Diamond)$.

\textbf{Example}. The following example will help to clarify the actual nature of the computation of the fixpoint of $\tau_s$. Consider the following two unit definitions:

\begin{align*}
\text{unit } u: & \quad \begin{bmatrix} p(x) :- q(x) \\ q(a) \end{bmatrix} \\
\text{unit } u_1 \text{ inherit } u: & \quad \begin{bmatrix} q(b) \end{bmatrix}
\end{align*}

Notice that $\{p(b), q(b)\}$ is the set of provable atoms in $u_1$, since the definition for $q$ in $u_1$ overrides the corresponding definition in $u$. The following steps show how this is indeed the set computed by $\bigcup_{n=1}^{\infty} \tau_i^n(i_u)(u_1)(\Diamond)$.
The computation of the fixpoint proceeds as follows. Let $\sigma$ be the signature of $\omega$.

$$
\tau_s(i_\perp)(u_1)(\omega) = \{ A \mid A \in \omega \circ \sigma |u_1| \text{ or } A : -G \in |u_1| \text{ and } \omega \circ \sigma i_\perp(u_1)(\omega) \models_s G \} \circ |u_1| i_\perp(u)(i_\perp(u_1)(\omega))
$$

(1)

Since $i_\perp(u) = \bot_\emptyset$ for any unit $u$, from the definition of $\bot_\emptyset$, it follows that $i_\perp(u_1)(\omega) = \emptyset$ and $i_\perp(u)(i_\perp(u_1)(\omega)) = \emptyset$. Then (1) can be rewritten as

$$
\tau_s(i_\perp)(u_1)(\omega) = \{ A \mid A \in \omega \circ \sigma |u_1| \text{ or } A : -G \in |u_1| \text{ and } \omega \models_s G \}
$$

or equivalently, $\tau_s(i_\perp)(u_1) = \lambda \omega. \omega \circ \sigma \{ q(b) \}$.

Now, to compute $\tau_s^2(u_1)(\omega)$ we proceed as follows. Let $\omega_1$ denote the set $\tau_s(i_\perp)(u_1)(\omega)$. Accordingly, $\omega_1 = \omega \circ \sigma \{ q(b) \}$ and let also $\sigma_1 = \| \omega_1 \|$. From the definition of $\tau_s$:

$$
\tau_s^2(i_\perp)(u_1)(\omega) = \{ A \mid A \in \omega \circ \sigma |u_1| \text{ or } A : -G \in |u_1| \text{ and } \omega \circ \sigma_1 i_\perp(u)(\omega_1) \models_s G \} \circ |u_1| i_\perp(u)(i_\perp(u_1)(\omega))
$$

(2)

To evaluate (2) we then need the result of $\tau_s(i_\perp)(u)(\omega_1)$. Again, from the definition:

$$
\tau_s(i_\perp)(u)(\omega_1) = \{ A \mid A \in \omega_1 \circ \sigma_1 |u| \text{ or } A : -G \in |u| \text{ and } \omega_1 \circ \sigma_1 i_\perp(u)(\omega_1) \models_s G \} \circ |u| i_\perp(u)(i_\perp(u)(\omega))
$$

(3)

and (3) can be simplified as follows:

$$
\tau_s(i_\perp)(u)(\omega_1) = \{ A \mid A \in \omega_1 \circ \sigma_1 |u| \text{ or } A : -G \in |u| \text{ and } \omega_1 \models_s G \}
$$

being $i_\perp(u)(i_\perp(u)(\omega)) = i_\perp(u)(\omega_1) = \emptyset$. By substituting $\omega_1$ in the previous expression, we obtain:

$$
\tau_s(i_\perp)(u)(\omega_1) = (\omega \circ \sigma \{ q(b) \} \circ \sigma \{ q(a) \} \cup \{ A \mid A : -G \text{ and } \omega \circ \sigma \{ q(b) \} \models_s G \}
$$

(4)

Finally, by choosing $\omega = \emptyset$, we can explicitly compute $\omega_1 = \{ q(b) \}$ and $\sigma_1 = \{ q \}$. Then, (4) simplifies to: $\tau_s(i_\perp)(u)(\omega_1) = \{ p(b), q(b) \}$ and (2) can be reduced simply to:

$$
\tau_s^2(i_\perp)(u_1)(\emptyset) = \{ q(b) \} \circ \{ p(b), q(b) \} = \{ p(b), q(b) \}
$$

(5)

which is the fixpoint.

6 Related Work

As repeatedly mentioned in the paper, Miller’s work on a logical analysis of modules and Monteiro and Porto’s Contextual Logic Programming have both greatly influenced the semantic framework described in these pages.

Miller’s approach was originally motivated by the idea of extending the power of positive horn clauses by allowing implications to occur in the bodies of clauses. The use of implication-goals of the form $D \supset G$ in a clause is logically justified by the deduction theorem: a goal $D \supset G$ is provable in a program $P$ if $G$ is provable in the extended program $P \cup \{ D \}$. If $D$ is a conjunction of clauses, then we can interpret the goal $D \supset G$ as a
scoping construct which requires that the clauses in $D$ be loaded before evaluating $G$ and then unloaded after $G$ succeeds or fails. This very same idea provides the foundation for the theory of modules in logic programming developed by Miller in [Mil89] and by other authors in [GMR88].

Monteiro and Porto's approach to the declarative semantics of Contextual Logic Programming is more closely related to our characterization of inheritance. Their framework results in a fixpoint computation for an immediate-consequence operator which is quite similar to our definition of $\tau_i$ for InheritLog. The major difference is in the language. In our case we are able to capture the semantics of dynamic inheritance which is not considered in [MP89]. In fact, Contextual Logic Programming has almost the same semantic connotation as InheritLog with the difference that InheritLog supports only static unit hierarchies whereas CxP's rules for unit composition are inherently dynamic. By providing a mechanism for dynamically specifying (and modifying) a unit's hierarchical links, CxP's context extension captures in effect a more general notion than inheritance which is known as delegation [Weg87].

Conversely, InheritLog's simplified nature allows us to use a first order framework to describe the semantics of a unit, whereby a unit's denotation is simply a set. Conversely, the use of higher order functions allows us to capture the semantics of the dynamic determination of self.

In a more recent paper [MP91] Monteiro and Porto took a more direct approach to the study of inheritance. The notion of inheritance they consider in (the bulk of) that paper is essentially the same we have assumed here. The semantic problem is instead approached from a completely different perspective. Their view is strictly transformational. The methodology to capture the meaning of an inheritance system is to transform it into a logic program to then show the equivalence between the respective operational semantics. A declarative interpretation for inheritance is then derived indirectly on the account of the well-known equivalence between the operational and declarative semantics in logic programming. Our declarative characterization of inheritance, we believe, represents a step forward from that stage and offers a better understanding of the nature of inheritance systems.

An extensive study on the semantics of various forms of composition mechanisms for logic programming has also been developed in [BLM]. In that paper, inheritance systems are viewed as a special case of more general forms of composition mechanisms which are derived as extensions or variations of Contextual Logic Programming. The approach is again rather different than the one presented in this paper, in at least two respects. The first is our use of functions to capture the meaning of dynamic inheritance as opposed to the use of standard set-based interpretations in [BLM]. The second, and perhaps more important one, is that the definition of inheritance assumed in that paper is based on the idea of extension rather than overriding between inherited definitions. This assumption is crucial for the framework presented in [BLM] to prove the existence of a fixpoint for the immediate consequence operator they define.

A final note concerns an implementation issue. It's interesting to address the connections between the semantic framework presented in this paper and the implementation schema reported in [LMN]. The key point here is that static inheritance allows one to compute the bindings "goal invocation - goal definition" at compile time in much the same way as its semantics can be characterized in terms of simple sets. Conversely, dynamic inheritance
requires that the computation of these bindings be deferred until run-time. Correspondingly, its semantics imposes the use of functions over sets which can be close only on a goal invocation.

Current Limitations. The framework we have presented in this paper is, we believe, an adequate one for characterizing general inheritance systems. There are some important features which are still to be cocered though. Two of them are perhaps more important and deserve further investigation. The first is is a characterization of super as defined in Smalltalk. Furthermore, the use of parametric units to be dynamically instantiated on method invocation would also be a desirable feature which would allow us to model the distinction between classes and objects as well as the dynamic creation of new objects. This extension seems actually an easy one, at least under the requirement that the parameters be strictly first order, i.e. elements of the domain $\mathcal{D}$. In that case, including parametric units should just result into a different characterization of the denotation of units as, for instance functions of type $i(u) : D^n \rightarrow \mathcal{P}(B) \rightarrow \mathcal{P}(B)$.

Aknowledgements

I'd like to thank Dr. Vincent Russo for kindly reading earlier versions of this paper and for his suggestions. I'm also most grateful to Dr. Cristina Ruggieri for her valuable feedback and insightful comments on the technical ideas contained in these pages.

References


[GR83] A. Goldberg and D. Robson. Smalltalk-80: The Language and its Implementation. Addison-Wesley, 1983.


