A Declarative View of Inheritance in Logic Programming

M. Bugliesi

Dipartimento di Matematica Pura ed Applicata
Via Belzoni 7, Padova – ITALY
michele@blues.unipd.it

Abstract

We discuss a declarative characterization of inheritance in logic programming. Our approach is inspired both by existing literature on denotational models for inheritance and by earlier work on a compositional definition of the semantics of logic Programming. We consider a general form of inheritance which is defined with an overriding semantics between inherited definitions and incorporates two different mechanisms known in the literature as static and dynamic inheritance. The result of our semantic reconstruction is an elegant framework which enables us to capture the compositional properties of inheritance and offers a uniform basis for the analysis of the different mechanisms we consider.

1 Introduction

The interest in modular logic programming has given rise in the recent literature to several proposals for capturing, at the semantic level, the compositional properties inherent in an modular approach to software development.

Inheritance provides one of various composition tools program development whose effectiveness has become widely accepted after the experience gained by the Object-Oriented community. Although the study of inheritance is not new in the logic programming field [11, 15, 4], the existing solutions seem still to provide only partial answers to the semantic problem. Some of them fail to capture the whole power of the notion of inheritance as defined within the O-O community ([2]) whereas others ([15]) achieve a transformational view whereby inheritance is logically understood only from a strictly operational point of view.

In this paper we present a new approach which is motivated and inspired by both existing literature on denotational models for Object-Oriented languages [17, 7] and by earlier work on a compositional definition of the semantics of Logic Programming [10, 3, 1].

We first consider an extended horn clause language, ObjectLog, which has an embedded form of modularity whereby a set of clauses can be collected into a named theory – a *unit* – and different units can communicate by requesting one another the evaluation of a goal. The mechanism for inter-unit communication is captured by allowing *message-goals* of the form $u : g$ to occur in the body of a clause. The intended meaning of $u : g$ is

the obvious one, namely to enforce the evaluation of the goal g by using the set of clauses provided by the unit u. As in [11], units are viewed in this context as the logical re-interpretation of the Object-Oriented notion of *object*. Correspondingly, the mechanism for inter-unit goal invocation provides a primitive operation for program composition which captures the idea of *message-passing* found in Object Orientation. The semantics of ObjectLog is derived by using a composite structure for interpretations in which all the units of a program are interpreted simultaneously in much the same spirit as in [12]. In contrast to that case, this approach allows us to capture also a model-theoretic semantics for our language.

We then introduce inheritance as an orthogonal compositional mechanism for building standard logic programs out of a set of program fragments. Following the style of [17] we consider two different mechanisms. Both defined with an overriding semantics, they embed the two forms of inheritance known as static inheritance – a la' Simula67 – and dynamic inheritance– a la' Smalltalk. Our semantic characterization of inheritance relies on the assumption that the semantics of a logic program should be regarded as a function over Herbrand sets rather than as a simple Herbrand set. This idea, which we originally borrowed from Reddy's denotational model [17], has also motivated various other proposals for a compositional semantics of logic programming ([16, 10, 3, 1]).

As the next and final step we then show how to incorporate the notion of inheritance within the linguistic framework provided by ObjectLog. The resulting language – Selflog – exhibits the typical Object-Oriented behaviour whereby the meaning of the $u : g$ is now to enforce the evaluation of the goal g not simply in the unit u but rather in the composition of u with all its ancestors. At the semantic level, the integration is obtained by combining the semantic models developed for inheritance and inter-unit goal invocation. The result is an elegant framework which enables us to capture the compositional properties of inheritance and provides a uniform basis for the analysis of the different mechanisms we consider.

The rest of the paper is organized as follows. In section 2 we show how to declaratively model the mechanism for inter-unit goal invocation. In section 3 we introduce inheritance and discuss the semantics of the dynamic and static interpretations. In section 4 we show how to combine inheritance and the idea of inter-unit goal invocation into a single framework. Finally, in section 5, we discuss the relations of our approach with the existing literature on the subject.

2   ObjectLog

ObjectLog is a logic language instrumented with two basic mechanisms to support a primitive form of modularity. ObjectLog allows one to declare a unit as a collection of clauses and it extends the syntax of horn clauses by

allowing message-goals to occur in the body of a clause. The syntax for unit declaration is simply: $\langle unit\ def \rangle ::= \texttt{unit}\ \langle unit\ name \rangle [\langle clauses \rangle]$.

The bodies of the clauses of a unit may contain message-goals of the form $u : G$ where u is a unit name and G is a goal. Following the style introduced in [14], in the declaration $\texttt{unit}\ u\ [\cdots]$ we will denote with $|u|$ the set of the ground instances of the clauses defined by u. ObjectLog's operational semantics is defined in terms of the proof predicate $\vdash$ which can be viewed as a special case of the more general relation used to define the semantics of Contextual Logic Programming in [14]. In the following definition, which we give in a Natural Deduction style, we use the notation $\Box$ to stand for the empty goal formula and ϵ for the identity substitution.

$$(O_1) \qquad \frac{}{u \vdash_\epsilon \Box} \qquad\qquad (O_2) \quad \frac{u \vdash_\sigma G\theta}{u \vdash_{\theta\sigma} g} \quad \left(\begin{array}{c} h\texttt{:-}G \in u \\ \theta = mgu(g, h) \end{array} \right)$$

$$(O_3) \quad \frac{u \vdash_\theta G_1 \quad u \vdash_\sigma G_2\theta}{u \vdash_{\theta\sigma} G_1, G_2} \qquad (O_4) \quad \frac{\hat{u} \vdash_\theta G}{u \vdash_\theta \hat{u}{:}G}$$

The notation $u \vdash_\theta G$ has been used here to stand for $u \vdash G\theta$. As in [12] an O-proof for $u \vdash G\theta$ is a tree rooted at $u \vdash_\theta G$, whose internal nodes are instances of one of the above inference figures and whose leaf nodes are labeled with the *initial* figure (O_1).

The interpretation of the above rules is straightforward. (O_1), (O_2) and (O_3) model the standard operational semantics of logic programming with the only difference that our programs are now units. As for (O_4), it states that evaluating a message-goal $u : G$ in any unit 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 [14].

2.1 Interpretations and Models

An ObjectLog program $P(U)$ is constructed as a set of units U. Its semantics is given in terms of a composite set-theoretic structure which provides a *model* for each of the component units. This structure is defined in much the same way as standard interpretations and models are used for characterizing the semantics of a logic program. The difference is that in this latter case we interpret a single program whereas in ObjectLog we would like to interpret several programs – our units – simultaneously. An elegant solution to this problem was first given in [12] in terms of a Kripke-like structure used to interpret the possibly infinite set of programs arising in that case. The approach we follow here is similar, but simpler, being the set of units of a program always finite. This enables us to introduce a standard notion of truth as the basis for a model-theoretic characterization for the semantics of the language.

A program $P(U)$ is defined over a set Σ of function symbols and a set Π of predicate symbols. The Herbrand Universe HU for $P(U)$ is built over Σ; the Herbrand Base B for each $u \in U$ is built from HU over Π. An

interpretation I for a program $P(U)$ is defined as the tuple of sets $\langle I(u) : u \in U \rangle$, where each $I(u)$ is the subset of B that interprets the associated unit. To distinguish between tuples of sets and sets we will henceforth use the term T-interpretation to refer to a tuple of interpretations. The class of T-interpretations will be denoted by $\Im$.

We begin by introducing a notion of satisfiability. The following definition is based on the classical notion of truth as set-membership. For any interpretation s and any T-interpretation I, we defined the *truth of a ground formula F with respect to s and I ($s \models_I F$)* as follows:

$$
\begin{array}{lll}
(1) & s \models_I G & \Longleftrightarrow \quad G \in s \\
(2) & s \models_I G_1, G_2 & \Longleftrightarrow \quad (s \models_I G_1 \text{ and } s \models_I G_2) \\
(3) & s \models_I h\text{:-}G & \Longleftrightarrow \quad (s \models_I G \Rightarrow s \models_I h) \\
(4) & s \models_I u : G & \Longleftrightarrow \quad I(u) \models_I G
\end{array}
$$

The T-interpretation I is meant to convey all the necessary information to establish the truth of a message-goal on the account of the truth of the corresponding goal in the specified unit. With this understanding, the meaning of the above definition is rather intuitive. A ground goal G is true *wrt s* and I if so are all the conjuncts of G. A clause is true if the head is true whenever the body is true. A message goal $u : G$ is true *wrt s* and I if the goal G is true *wrt $I(u)$*, the interpretation that I associates with u, and I (independently of the interpretation s).

Based on this definition of satisfiability, we have a corresponding notion of T-model.

Definition 2.1 A T-interpretation I is a T-model for a program $P(U)$ iff for every $u \in U$ and each clause C of $|u|$, $I(u) \models_I C$. $\qquad\square$

Notice that if U is a singleton set, the notions of truth, T-interpretation and T-model coincide with the classical ones.

2.2 Model-Theoretic Semantics

The standard meaning of a logic program P is defined as the least Herbrand model M_P of P. The model-theoretic semantics for ObjectLog, based on T-models, is defined along the guidelines of the classical approach.

Partial Ordering on T-Interpretations. We first introduce the ordering relation $\sqsubseteq_\Im$ over T-interpretations. The definition of $\sqsubseteq_\Im$ is derived from the partial order $\subseteq$ (set inclusion) on interpretations in the usual way. For any program $P(U)$, if I_1 and I_2 are two T-interpretations, $I_1 \sqsubseteq_\Im I_2$ *iff* $\forall u \in U \, I_1(u) \subseteq I_2(u)$. The set of T-interpretations associated with $P(U)$ is a complete lattice with join and meet operators defined respectively as:

$$
\begin{array}{llll}
\text{(join)} & (I_1 \sqcup I_2) & = & \forall u \in U \; I_1(u) \cup I_2(u) \\
\text{(meet)} & (I_1 \sqcap I_2) & = & \forall u \in U \; I_1(u) \cap I_2(u)
\end{array}
$$

$I_\perp = \langle \oslash, \ldots, \oslash \rangle$ and $I_\top = \langle B, \ldots, B \rangle$ denote respectively the bottom and top element of this lattice. It's easy to see that $I_\top$ is a T-model for the associated program and that the standard model intersection property holds on T-models (given two T-models M_1 and M_2, $M_1 \sqcap M2$ is also a T-model). Hence, a minimal T-model exists for any program – the intersection (meet) of all the T-models –.

2.3 Fixpoint Semantics

A constructive characterization of the minimal T-model can be given following the same idea as [12]. Given a program $P(U)$, we define a transformation $T_{P(U)}$ for whose fixpoint is the minimal T-model for $P(U)$. In contrast to [12], $T_{P(U)}$ is defined here in terms of the immediate-consequence transformation of each of the units of $P(U)$. For u in U and $I \in \Im$, let $T_{u,I} : P(B) \mapsto P(B)$ be defined as follows:

$$T_{u,I}(s) \quad = \quad \{A \mid A\text{:-}G \in |u| \text{ and } s \models_I G\}$$

We then define the immediate consequence operator $T_{P(U)}$ for the program as a transformation from T-interpretations to T-interpretations as

$$T_{P(U)}(I) = \langle T_{u,I}(I(u)) \mid u \in U \rangle$$

$T_{P(U)}$ is monotonic and continuous on $\Im$. The proof, (see [6]), follows by showing that for any $I \in \Im$, $T_{u,I}$ is continuous on $P(B)$ and that for any increasing sequence of T-interpretations $I_1 \sqsubseteq_\Im \ldots \sqsubseteq_\Im I_n \ldots$, $T_{u,\sqcup_{j=1}^\infty I_j} = \sqcup_{j=1}^\infty T_{u,I_j}$.

The continuity of $T_{P(U)}$ guarantees the existence of a least fixpoint. The fixpoint is built by simultaneously carrying on the computation of the transformations associated with each unit thus capturing the interdependency among the interpretations for the units in $P(U)$ implicit in the idea of message-goal. The fixpoint of $T_{P(U)}$ gives precisely the minimal T-model $M_{P(U)}$ for $P(U)$.

Theorem 2.1 [6] $M_{P(U)} = lfp(T_{P(U)}) = \sqcup_{k=1}^\infty T_{P(U)}^k(I_\perp)$. $\qquad\qquad \square$

As in the classical approach, we can now use the fixpoint construction to show that the minimal T-model for a program coincides with the set of atomic and ground logical consequences of the program.

Theorem 2.2 [6] Let $M = \langle M(u) \mid u \in U \rangle$ be the minimal model for a program $P(U)$. Then for any $u \in U$ and any ground goal G, there exists an O-proof for $u \vdash_\epsilon G$ if and only if $M(u) \models_M G$. $\qquad\qquad \square$

Example 2.1 Consider a program $P(U)$ with $U = \{u_1, u_2\}$ where:

$$u_1 \quad [\; p\text{:-}u_2 : q. \quad r. \;] \qquad\qquad\qquad u_2 \quad [\; q\text{:-}u_1 : r. \;]$$

The minimal T-model for $P(U)$ is the T-interpretation $M_{P(U)} = \langle \{r, p\}, \{q\} \rangle$. $M_{P(U)}$ is obtained at the third step of the iteration $T_{P(U)}^{(n)}(I_\perp)$. The T-interpretation computed at the first step is $\langle \{r\}, \oslash \rangle$. At the next step we are to compute $T_{P(U)}^{(2)}(I_\perp) = \langle T_{u_1,I}(\{r\}), T_{u_2,I}(\oslash) \rangle$, where $I = \langle \{r\}, \oslash \rangle$ and $T_{u_2,I}(\oslash) = \{A \mid A\texttt{:-}G \in |u_2| \text{ and } \oslash \models_I G\}$. Now, $\oslash \models_I u_1 : r$ iff $\{r\} \models_I r$ and thus $T_{u_2,I}(\oslash) = \{q\}$ and $T_{P(U)}^{(2)}(I_\perp) = \langle \{r\}, \{q\} \rangle$. Following the same argument, it's easy to see that $M_{P(U)}$ is indeed the T-interpretation obtained at the next iteration and that it is the fixpoint. $\square$

3 Inheritance

In this section we introduce inheritance as an independent composition mechanism for logic programs Throughout this section we will use the terms logic program, program fragment and unit interchangeably, with the understanding that we are now restricting ourselves to consider (the composition of) programs built over the language of horn clauses. We begin by informally introducing the interpretation of inheritance we use throughout.

A first abstract model for inheritance was developed using a denotational framework in two independent papers ([7] and [17]). In [17], Reddy carries out a systematic analysis of two different interpretations of inheritance and investigates their respective semantic characterizations. We illustrate the point here in terms of two possible ways of interpreting the hierarchical composition of logic programs (units). Let's first introduce a little notation. Given two units, u and su, we denote with u $\lhd$ su the composition of u and su into a hierarchy where su is u's immediate ancestor (its *super* unit). We also assume an overriding semantics for $\lhd$ whereby if both u and su in u $\lhd$ su contain a definition for same predicate, then the definition in u overrides the one found in su.

Example 3.1 Consider a program built as the composition of su and u, where u inherits from su and redefines one of its super-unit's predicates.

$$
\text{su} \quad \begin{bmatrix} iAm(\texttt{'su'}). \\ whoAreYou(x)\texttt{:-}iAm(x). \end{bmatrix} \qquad \text{u} \ [\, iAm(\texttt{'u'}).\,]
$$

We want to look at the evaluation of the query $whoAreYou(x)$ in u $\lhd$ su. Clearly, the result depends on the binding for the predicate iAm in su. In fact, we can either bind it to su's local definition, or consider u $\lhd$ su as a modified version of su where iAm is bound to the definition provided in u. Put a little differently, we can think of the invocation $iAm(x)$ as a shorthand for the typical Object-Oriented notation $\texttt{self:}iAm(x)$, and rephrase the above statement in terms of the two corresponding bindings for self. The two choices reflect the behaviour of the two different mechanisms that Reddy identifies respectively as static and dynamic inheritance. If $iAm(x)$ is evaluated in su (self is *statically* bound to su) then the answer will be the

substitution $\{x \mapsto \text{'su'}\}$. Conversely, if $iAm(x)$ is evaluated in the composition of su with u, (self is *dynamically* bound to $u \lhd su$), then the resulting substitution will be $\{x \mapsto \text{'u'}\}$. $\qquad\qquad\qquad\qquad\qquad\qquad$ □

The semantics of the composition $u \lhd su$ can be formally specified at different levels. At the operational level, by defining the rules for evaluating a goal into the composed program $u \lhd su$. Alternatively, we can take a transformational view and define a way for constructing a new logic program which behaves as the composition of u and su. Finally, at the declarative level we can try and interpret u and su to derive the interpretation of $u \lhd su$ in terms of the interpretation of the components

In the next section we will consider these three approaches, the goal being to achieve a declarative characterization and to establish its relationship with the operational semantics. The transformational view provides the necessary link between these two end-points. The discussion will be carried on by assuming a dynamic connotation for the composition operator $\lhd$ and simply addressing analogies and differences with respect to the static case.

3.1 Hierarchies of logic programs

The operational semantics of the composition $\lhd$ has been already studied in the literature [2, 14]. We include here a brief survey to make the discussion self-contained. We will denote with $\pi(u)$ the set predicate symbols *defined* by the unit u. The same notation will be used also for arbitrary hierarchies according to the definition: $\pi(u \lhd su) = \pi(u) \cup \pi(su)$. H will denote a generic hierarchy $u_1 \lhd u_2 \lhd \ldots \lhd u_n$ and for any atom A, $pred(A)$ will stand for the predicate symbol of A. Finally, for any goal G and any hierarchy H, $u_{H,G}$ will stand for the first unit found in a left-to right scan of H such that $\pi(u)$ contains $pred(G)$.

The operational semantics of the composition $\lhd$ can be defined in terms of the following inference figures which we borrow from [2].

$$(I_1) \qquad \overline{H \vdash_\epsilon \square}$$

$$(I_2) \qquad \frac{H \vdash_\sigma G\theta}{H \vdash_{\theta\sigma} A} \qquad (\theta = mgu(A, \bar{A}) \text{ and } \bar{A}\text{:-}G \in u_{H,A})$$

$$(I_3) \qquad \frac{H \vdash_\theta G_1 \qquad H \vdash_\sigma G_2\theta}{H \vdash_{\sigma\theta} G_1, G_2}$$

We say that $H \vdash G\theta$ has an I-proof if there is a tree rooted at $H \vdash_\theta G$ whose internal nodes are instances of one of the above figures and whose leaf nodes are labeled with figure (I_1). The meaning of (I_1) and (I_3) is straightforward. As for (I_2), it formalizes the overriding semantics of $\lhd$. The search for a matching clause for A stops at $u_{H,A}$ – the first unit in the hierarchy which defines $pred(A)$ – thus hiding any other definition for $pred(A)$ found in any of u_A's ancestors. Notice also that the evaluation of the body of the clause

takes place in the whole hierarchy H, regardless of the unit in H where the clause occurs. This reflects the dynamic connotation for $\lhd$ which we have addressed earlier in this section. If $H = u_1 \lhd \ldots \lhd u_{H,A} \lhd \ldots \lhd u_n$, the corresponding rule for the static case would be drawn as follows ([14]):

$$(I_{2'}) \quad \frac{u_{H,A} \lhd \ldots \lhd u_n \vdash_\sigma G\theta}{H \vdash_{\theta\sigma} A} \quad (\theta = mgu(A, \bar{A}) \text{ and } \bar{A}\text{:-}G \in u_{H,A})$$

The body G of $\bar{A}\text{:-}G$ is now evaluated in the sub-hierarchy whose tip node is $u_{H,A}$, using therefore only $u_{H,A}$'s local and inherited definitions. This suggests that static inheritance should indeed be understood as a scope mechanism for the clauses of a hierarchy. Later in this section we will see how its semantics can be given in essentially the same style as the semantics of the *static scope rules for local definitions* defined in [9]. This analogy was first pointed out in [2].

From figures $(I_1) - (I_3)$ it's easy to see how to construct a logic program equivalent to the composition of two programs through $\lhd$.

Definition 3.1 Let u_P and u_Q be two logic programs. $u_P \oplus u_Q$ denotes the logic program obtained as the union of the clauses of u_P with the clauses of u_Q which do not define any of the predicates in $\pi(u_P)$.
Formally: $u_P \oplus u_Q = \{A\text{:-}G \in u_P\} \cup \{A\text{:-}G \in u_Q : pred(A) \notin \pi(u_P)\}$. $\square$

The relation between $u_P \oplus u_Q$ and $u_P \lhd u_Q$ is formalized on the basis of their respective operational semantics by the following result.

Proposition 1 For any goal G, there exists an I-proof for $u_P \lhd u_Q \vdash G\theta$ *iff* there exists an SLD refutation for G in $u_P \oplus u_Q$ with final substitution θ.
Proof. By induction on the height of the I-proof for $u_P \lhd u_Q \vdash_\theta G$. The I-proof has height n if and only if G is refutable in n SLD steps from $u_P \oplus u_Q$ and θ is the computed substitution.

3.2 Declarative semantics

At the declarative level, the semantics of $\lhd$ is given relying on the assumption that the semantics of a logic program should be regarded as a function over Herbrand sets rather than as a simple Herbrand set. As noted in [10], this choice is crucial to achieve a compositional semantics for modular logic programming. The idea of "moving from an *object level* semantics to a *function level* semantics ([10])" has been largely investigated in the recent literature. First introduced in [16], it has been similarly exploited in [10] to define a set of algebraic operators on logic programs and, more recently, it has also inspired the study of the semantics of *open* logic programs in [1] and [3]. Here we take the same view as [10] and we base our construction on the assumption that the meaning of a program be its associated transformation T_P. This characterization has an immediate counterpart in the

use of *functionals* for defining the semantics of (sub-)classes in [17] and provides a uniform basis for the analysis of the static and dynamic models for inheritance we are considering.

We first introduce a new operator on $P(B)$ which provides the formal device for modeling, in a set-theoretic sense, the overriding semantics of $\lhd$.

Definition 3.2 Let π denote an arbitrary set of predicate symbols and let S_1 and S_2 be two sets in $P(B)$. Then we define a function $\diamondsuit_\pi : P(B) \times P(B) \mapsto P(B)$ as follows: $\quad S_1 \diamondsuit_\pi S_2 = S_1 \cup \{t \in S_2 \mid pred(t) \notin \pi\}$ $\hfill \square$

The use of $\diamondsuit_\pi$ for modeling the overriding semantics of inheritance can be intuitively motivated in terms of the classical minimal-model semantics as follows. For two programs u_P and u_Q, if M_P and M_Q are two models of u_P and u_Q, a model for $u_P \oplus u_Q$ is obtained as a superset of $M_P \diamondsuit_{\pi(u_P)} M_Q$. A more precise result holds if we assume a function-level semantics.

Now let Φ denote the set of continuous mappings from $P(B)$ to $P(B)$. Φ is a complete lattice under the order relation $\leq$ obtained from $\subseteq$ in the standard way (for $T_1, T_2 \in \Phi$, $T_1 \leq T2$ *iff* $\forall I \in P(B)$ $T_1(I) \subseteq T_2(I)$). We can then lift the definition of $\diamondsuit_\pi$ at the function level.

Definition 3.3 Let T_1, T_2 be two continuous functions in Φ, and π be a set of predicate symbols. Then $\quad T_1 \diamondsuit_\pi T_2 = \lambda I \,.\, T_1(I) \diamondsuit_\pi T_2(I)$ $\hfill \square$

We have abused the notation here and let $\diamondsuit_\pi$ denote two different operators defined respectively over Φ and $P(B)$. For any choice of π, $\diamondsuit_\pi$ can be shown to be co-continuous on $P(B)$, and on Φ ([6]). Hence $\diamondsuit_\pi$ is well defined on Φ. Its use as the semantic counterpart of the syntactic operator $\oplus$ is formalized by the following result.

Theorem 3.1 For any two programs u_P and u_Q, $T_{(u_P \oplus u_Q)} = T_{u_P} \diamondsuit_{\pi(u_P)} T_{u_Q}$.
Proof. For any $I \subseteq B$,

$$
\begin{aligned}
T_{(u_P \oplus u_Q)}(I) &= \{A \mid A\text{:-}G \in |u_P \oplus u_Q| : G \subseteq I\} \\
&= \{A \mid A\text{:-}G \in |u_P| : G \subseteq I\} \cup \\
&\quad \cup \{A \mid A\text{:-}G \in |u_Q| : pred(A) \notin \pi(u_P) \ \& \ G \subseteq I\} \\
&= T_{u_P}(I) \cup \{A \in T_{u_Q}(I) : pred(A) \notin \pi(u_P)\} \\
&= T_{u_P}(I) \diamondsuit_{\pi(u_P)} T_{u_Q}(I) \hfill \square
\end{aligned}
$$

As a corollary, under the assumption $[\![u_P]\!] = T_{u_P}$, we have a proof for the identity: $[\![u_P \oplus u_Q]\!] = [\![u_P]\!] \diamondsuit_{\pi(u_P)} [\![u_Q]\!]$. Notice that the above equality wouldn't hold if we interpreted $[\![\cdot]\!]$ as the classical minimal-model semantics. Simply take $u_P = \{p(a)\text{:-}q(b)\}$ and $u_Q = \{q(b)\}$. Then $M_{u_P} = \oslash$ and $M_{u_Q} = \{q(b)\}$ whereas $M_{u_P \oplus u_Q} = \{p(a), q(b)\} \neq M_{u_P} \diamondsuit_{\pi(u_P)} M_{u_Q}$. This is an instance of the well known problem due to the non OR-compositionality of the minimal-model approach to the semantics of logic programming.

Now, on the account of the relation between $u_P \oplus u_Q$ and $u_P \lhd u_Q$ established in proposition 1, we have the following *compositional* [8] definition for the declarative semantics of $\lhd$.

Definition 3.4 For any two programs u_P and u_Q,

$$[\![u_P \lhd u_Q]\!] \stackrel{\text{def}}{=} [\![u_P \oplus u_Q]\!] = [\![u_P]\!]\diamond_{\pi(u_P)}[\![u_Q]\!] \qquad (1)$$

In [5] a similar construction is used to characterize the semantics of the algebraic operator *is_a* which has essentially the same connotation as $\lhd$. The key difference is that our definition is compositional whereas in [5] the semantics of u_P *is_a* u_Q is given in terms of the semantics of u_P and of a *new* program $\overline{u_Q}$ obtained by filtering out of u_Q the definitions for all the predicates in $\pi(u_P)$. As such, the overriding semantics of *is_a* is captured at the syntactic level in [5] whereas the meaning of $\lhd$ is defined entirely at the semantic level. This also makes (1) independent of the choice of the partition u_P, u_Q.

Proposition 2 For u_P, u_Q and u_R programs,

$$[\![u_P \lhd u_Q \lhd u_R]\!] = [\![u_P \lhd u_Q]\!]\diamond_{\pi(u_P\oplus u_Q)}[\![u_R]\!] = [\![u_P]\!]\diamond_{\pi(u_P)}[\![u_Q \lhd u_R]\!]$$

Proof. Immediate by observing that, for any $S, S_1, S_2 \in P(B)$ and any $\pi_1, \pi_2 \in \Pi$, $\ S\diamond_{\pi_1}(S_1\diamond_{\pi_2}S_2) = (S\diamond_{\pi_1}S_1)\diamond_{\pi_1\cup\pi_2}S_2$ $\hfill\square$

3.3 A note on static inheritance

Using the above characterization, we can furtherly investigate the semantic implications of the two different interpretations of $\lhd$. We will use the notation $\lhd_s$ to identify the static composition, whereas $\lhd$ will denote the dynamic one.

First notice that, according to (1), the meaning of $u_P \lhd u_Q$ is the result of the mutual dependency between u_P and u_Q. This becomes clear by observing that $[\![u_P \lhd u_Q]\!] = \lambda I . T(I) = \lambda I . T_{u_P}(I)\diamond_{\pi(u_P)}T_{u_Q}(I)$, and thus that T_{u_P} and T_{u_Q} both depend on the same interpretation I. If we took the fixpoint of T, the set computed at each step of the iterative computation $T^{(1)}(\oslash),\ldots,T^{(n)}(\oslash)$ would be a subset of the set obtained by performing one deduction step for u_P and u_Q based on the same set of hypothesis I. Operationally, this corresponds to the fact that the evaluation of a goal G in u_P might use a clause of u_Q (if G is not defined in u_P), and vice-versa, evaluating G in u_Q might involve a clause of u_P.

The case for $\lhd_s$ is substantially different since the behaviour of each unit in a hierarchy becomes determinate by only looking at the local and the inherited definitions. As such, in $u_P \lhd_s u_Q$, u_P depends on u_Q but not vice-versa. At the semantic level, this results into a different relation which shows how the meaning of u_Q is indeed determined independently of u_P. Namely: $[\![u_P \lhd_s u_Q]\!] = \lambda I . T_{u_P}(I)\diamond_{\pi(u_P)}T_{u_Q}^\infty(\oslash)$. Notice how we are now taking the fixpoint of T_{u_Q} (instead of T_{u_Q} itself) thus *closing* the interpretation of u_Q. The resulting set, filtered by $\diamond_{\pi(u_P)}$ is then used as the set of hypotheses for the deduction steps of T_{u_P}. A similar characterization is used in [9] to define

the fixpoint semantics for the scope rules considered there. This relation between the different semantic properties of $\triangleleft$ and $\triangleleft_s$ was first similarly explained by Reddy in his denotational model.

4 SelfLog

In this section we show how to integrate the notion of (dynamic) inheritance introduced before within the linguistic framework of section 2. The integration is studied in terms of the semantics of a concrete language, Self-Log, which extends ObjectLog by allowing the hierarchical relation between two units to be declared explicitly. Syntactically, the extension amounts to adding a new production for unit declaration. The declaration `unit` $u \prec$ su [$\langle clauses \rangle$] associates $\langle clauses \rangle$ with u and declares su as the immediate ancestor of u. The corresponding hierarchy is denoted by $u \prec su$. A SelfLog program $P(U)$ is a set of units configured as a tree-like hierarchy where for each $u \in U$ there is at most one other unit $\bar{u}$ such that $u \prec \bar{u}$. In the following $H(u)$ will denote the hierarchy obtained by taking the transitive closure of $\prec$ starting from u and the term *base*-unit will be used to refer to a unit without any ancestor ($u = H(u)$ if u is a base-unit).

Operationally, SelfLog can be described by simply integrating the inference figures for $\triangleleft$ within the operational characterization of ObjectLog.

(S_1)
$$\overline{H \vdash_\epsilon \square}$$

(S_2)
$$\frac{H \vdash_\sigma G\theta}{H \vdash_{\theta\sigma} A} \qquad (\theta = mgu(A, \bar{A}) \text{ and } \bar{A}\text{:-}G \in u_{H,A})$$

(S_3)
$$\frac{H \vdash_\theta G_1 \qquad H \vdash_\sigma G_2\theta}{H \vdash_{\sigma\theta} G_1, G_2}$$

(S_4)
$$\frac{u \triangleleft \ldots \triangleleft u_n \vdash_\theta G}{H \vdash_\theta u : G} \qquad (H(u) = u \prec u_1 \prec \ldots \prec u_n)$$

The notation $u_{H,A}$ has been used with the same meaning as in $(I_1) - (I_3)$. An S-proof for $H \vdash G\theta$ is defined, as before, as a proof for $H \vdash_\theta G$ which uses the inference figures $(S_1) - (S_4)$. Notice that we have also implicitly extended the definition of $\triangleleft$ to apply to arbitrary SelfLog units rather than to simple logic programs. With the same understanding, we extend the notation $u_P \oplus u_Q$ to refer to the *syntactic* composition introduced in definition 3.1 applied now to two units. We can, correspondingly, establish the relation between $u_P \triangleleft u_Q$ and $u_P \oplus u_Q$ in the context of SelfLog.

Proposition 3 Let u_P and u_Q be two units and G be a goal. An S-proof for $u_P \triangleleft u_Q \vdash G\theta$ exists if and only if there exists an O-proof for $u_P \oplus u_Q \vdash G\theta$.
Proof. By induction on the height of the proofs. For the inductive step, by considering the cases where the last inference if the S-proof is an instance of $(S_1) - (S_4)$ (and correspondingly for the O-proof and $(O_1) - (O_4)$). $\square$

4.1 Declarative Semantics

The operational characterization given in the previous section also suggests a declarative interpretation for a SelfLog program. The fact that the evaluation of $u : G$ is carried out by evaluating G in $H(u)$ can be declaratively interpreted by taking as the meaning of each unit u in $P(U)$ the meaning of the associated hierarchy $H(u)$. Under this assumption, a declarative characterization of a SelfLog program can be still given in terms of the notions of T-interpretation and of truth defined for ObjectLog. What changes is the definition of T-model which generalizes the definition given for ObjectLog. A T-model will again be a tuple of interpretations where now each component provides a model for the hierarchy associated with the corresponding unit.

Definition 4.1 Let $P(U)$ be a SelfLog program and let M be a T-interpretation for $P(U)$. M is a T-model for $P(U)$ iff for every u such that $H(u) = u \prec u_1 \prec \ldots, \prec u_n$, $M(u) \models_M C$ for every clause C of $|u \oplus \ldots \oplus u_n|$.

Again, the top element $I_\top$ of $\Im$ is a T-model for any program and the intersection of all the T-models is the minimal T-model for the program itself.

4.2 Fixpoint Semantics

Following the same idea used for the declarative characterization, we can also give a constructive definition of the minimal T-model in terms of a fixpoint computation. Under the assumption $[\![u_P]\!] = T_{u_P}$, equating the meaning of a hierarchy with the meaning of its top unit amounts to taking as $[\![u_P]\!]$ not just T_{u_P} but rather the immediate-consequence operator for the hierarchy. For any unit u and $I \in \Im$, let then again $T_{u,I} : P(B) \mapsto P(B)$ be defined as in section 2.3 by: $T_{u,I}(s) = \{A \mid A\text{:-}G \in |u| \text{ and } s \models_I G\}$. Now, if $H(u) = u \prec u_1 \prec \ldots, \prec u_n$, we would like to associate with u not $T_{u,I}$ but a new transformation $T_{u.I}$ which incorporates the semantics of the composition $u \triangleleft u_1 \triangleleft \ldots, \triangleleft u_n$. Definition 3.4 suggests how $T_{u,I}$ should be defined.

Definition 4.2 For any unit u and T-interpretation I, the transformation $T_{u,I} : P(B) \mapsto P(B)$ is given inductively by:

$$
\begin{aligned}
T_{u,I}(s) &= T_{u,I}(s) && \text{if } u \text{ is a base unit} \\
T_{u,I}(s) &= T_{u,I}(s) \Diamond_{\pi(u)} T_{su,I}(s) && \text{if } u \prec su \qquad \square
\end{aligned}
$$

The definition of the base case is a consequence that any base units coincides with its associated hierarchy. As for the inductive case, if $H(u) = u \prec u_1 \prec \ldots, \prec u_n$, the equation above can be expanded as follows:

$$
\begin{aligned}
T_{u,I}(s) &= T_{u,I}(s) \Diamond_{\pi(u)} (T_{u_1,I}(s) \Diamond_{\pi(u_1)} (\cdots (\Diamond_{\pi(u_n)} T_{u_n,I}(s)))) \\
\text{(by prop. 2)} &= T_{u,I}(s) \Diamond_{\pi(u)} T_{u_1,I}(s) \Diamond_{\pi(u \oplus u_1)} \cdots \Diamond_{\pi(u \oplus \ldots \oplus u_n)} T_{u_n,I}(s)
\end{aligned}
$$

Accordingly, for any $u_j \in H(u)$ we obtain the expected effect of filtering out of the interpretation for u_j all the elements which are produced by local definitions and whose predicate symbol is defined also in any of u_j's heirs.

The transformation $T_{P(U)} : \Im \mapsto \Im$ can be finally defined as for ObjectLog as: $T_{P(U)}(I) = \langle T_{u,I}(I(u)) : u \in U \rangle$. $T_{P(U)}$ can be taken as the semantics of a SelfLog program by imposing $[\![P(U)]\!] = T_{P(U)}$. This is justified by the following results which generalize the corresponding ones introduced for ObjectLog.

Theorem 4.1 [6] $M_{P(U)} = lfp(T_{P(U)}) = \sqcup_{k=1}^{\infty} T_{P(U)}^k(I_{\perp})$. $\qquad\qquad\square$

Theorem 4.2 [6] Let M be the minimal model for a program $P(U)$. For any u in $P(U)$ and any ground goal G, there exists an S-proof for $u \vdash_{\epsilon} G$ if and only if $M(u) \models_M G$. $\qquad\qquad\square$

Notice that in the computation of the fixpoint, many instances of $T_{u,I}$ for the same u may be active at the same time. As a matter of fact, there are as many active instances of $T_{u,I}$ as the number of the hierarchies which u is part of. In each of these hierarchies u assumes a different semantic connotation depending on its heirs.

Example 4.1 Let $P(U)$ be the following modified version of the program reported in example 3.1.

$$u \prec su\,[\ i(\mathtt{u}).\ ] \qquad su \begin{bmatrix} i(\mathtt{su}). \\ w(x) \ :\text{-}\ i(x). \end{bmatrix} \qquad u_1\,[\ t(x)\ :\text{-}\ u : w(x).\ ]$$

We have used w and i respectively as shorthands for $whoAreYou$ and iAm and t as a shorthand for $test$. Any T-interpretation for $P(U)$ is a 3-tuple of the form $\langle I(u), I(su), I(u_1) \rangle$. The expected minimal T-model is $M_{P(U)} = \langle \{i(u), w(u)\},\ \{i(su), w(su)\},\ \{t(u)\} \rangle$. The fixpoint is obtained by iterating the computation of $T_{P(U)}^{(n)}(I_{\perp})$ where:

$$\begin{aligned}
T_{P(U)}(I) &= \langle T_{u,I}(I(u)),\ T_{su,I}(I(u)),\ T_{u_1,I}(I(u)) \rangle \\
&= \langle T_{u,I}(I(u))\Diamond_{\pi(u)}T_{su,I}(I(u)),\ T_{su,I}(I(su)),\ T_{u_1,I}(I(u_1)) \rangle
\end{aligned}$$

The fixpoint is reached in three steps: $T_{P(U)}^1(I_{\perp}) = \langle \{i(\mathtt{u})\}, \{i(\mathtt{su})\}, \oslash \rangle$, $T_{P(U)}^2(I_{\perp}) = \langle \{i(\mathtt{u}), w(\mathtt{u})\}\}, \{i(\mathtt{su}), w(\mathtt{su})\}, \{\oslash)\}$ and finally $T_{P(U)}^3(I_{\perp}) = M_{P(U)}$. $\qquad\qquad\square$

5   Related Work

We have already pointed out analogies and differences for the approach described here with respect to various related solutions found in the literature.

An extensive study on the semantics of various forms of composition mechanisms for logic programming has also been developed in [2]. 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 rather different than the one presented in this paper, in 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 [2]. The second but not less important one is that the definition of inheritance assumed in that paper is based on the notion of *extension*, rather than overriding, between inherited definitions. This assumption is crucial for the framework presented in [2] to prove the existence of a fixpoint for the immediate consequence operator they define. The same assumption constitutes the basis also for the compositional approach developed in a more recent paper ([4]) by the same authors.

A notion of inheritance which is essentially the same as the dynamic interpretation we have assumed here is also considered in [15]. The semantic problem is instead approached from a different and strictly transformational perspective. 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.

Recently, an approach similar to the one we haved presented in this paper has been independently studied by Monteiro and Porto in a paper which is to appear in the forthcoming proceedings of the *PHOENIX Seminar on Declarative Programming* [13].

References

[1] A. Bossi, M. Gabbrielli, G. Levi, and M. Meo. Contributions to the Semantics of Open Logic Programs. In *in Proceedings of FGCS'92 Int. Conf*, 1992. (to appear).

[2] A. Brogi, E. Lamma, and P. Mello. Structuring Logic Programs: A Unifying Framework and its Declarative and Operational Semantics. Technical Report 4/1, Progetto Finalizzato C.N.R. Sistemi Informatic e Calcolo Parallelo, 1990.

[3] A. Brogi, E. Lamma, and P. Mello. Compositional Model-theoretic Semantics for Logic Programs. Technical Report 8-91, Comp. Sci. Dept. University of Pisa, 1991.

[4] A. Brogi, E. Lamma, and P. Mello. Objects in a Logic Programming Framework. In *Proceedings of the 2nd Russian Conf. on Logic Programming*, Leningrad, 1991.

[5] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Composition Operators for Logic Theories. In *Proc. Symposium on Computational Logic*. Springer-Verlag, Basic Research Series, 1990.

[6] M. Bugliesi. Inheritance Systems in Logic Programming: Semantics and Implementation. Ms Thesis, Dept. of Computer Science, Purdue University, West-Lafayette IN, USA.

[7] W. Cook and J. Palsberg. A Denotational Semantics of Inheritance and its Correctness. In *Proceedings of the ACM OOPSLA'89 Int. Conf.*, pages 433–443. ACM, 1989.

[8] H. Gaifman and E. Shapiro. Fully Abstract Compositional Semantics for Logic Programs. In *Proceedings of the ACM Conf. on Principle of Programming Languages*, 1989.

[9] L. Giordano, A. Martelli, and F. Rossi. Local definitions with static scope rules in Logic Languages. In *Proceedings of the FGCS'88 Int. Conf.*, 1988.

[10] P. Mancarella and D. Pedreschi. An Algebra of Logic Programs. In *Proceedings of the 5th ALP Int. Conf. on Logic Programming*, Seattle, 1988.

[11] F. G. McCabe. *Logic and Objects. Language, application and implementation*. PhD thesis, Dept. of Computing, Imperial College of Science and Technology, Univ. of London, 1988.

[12] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6(2):79–108, 1989.

[13] L. Monteiro. Personal communication. February/March 1992.

[14] L. Monteiro and A. Porto. Contextual Logic Programming. In *Proceeding of the 6th ALP Int. Conf. on Logic Programming*, Lisbon, 1989.

[15] L. Monteiro and A. Porto. A Transformational View of Inheritance in Logic Programming. In *Procof the 7th ALP Int. Conf. on Logic Programming*, 1990.

[16] R. A. O'Keef. Towards an Algebra for Constructing Logic Programs. In *Proceeding of the IEEE Symposium on Logic Programming*, 1985.

[17] U. Reddy. Objects as Closures: Abstract Semantics of Object Oriented Languages. In *Proceedings of the ACM Int. Conf. on Lisp and Functional Programming*, pages 289–297, 1988.