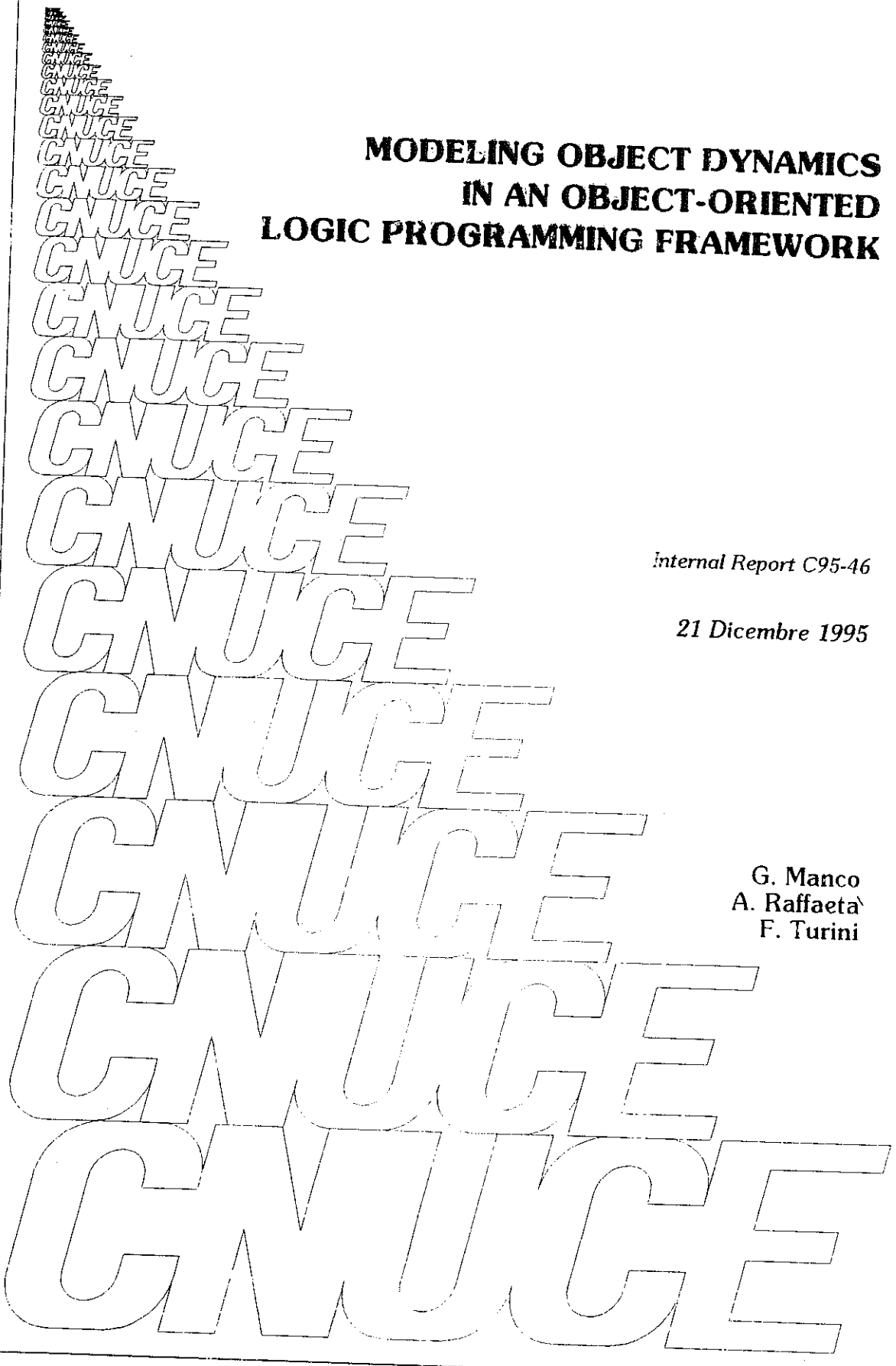

MODELING OBJECT DYNAMICS IN AN OBJECT-ORIENTED LOGIC PROGRAMMING FRAMEWORK

Internal Report C95-46

21 Dicembre 1995

G. Manco
A. Raffaetà
F. Turini

Modeling Object Dynamics in an Object-Oriented Logic Programming Framework

Giuseppe Manco

CNUCE–CNR
Via S. Maria 36, 56125 Pisa, Italy
e-mail manco@orione.cnuce.cnr.it

Alessandra Raffaetà Franco Turini

Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
e-mail {raffaeta,turini}@di.unipi.it

December 21, 1995

Abstract

We extend logic programming with object-oriented features. Classes and objects are represented by collection of clauses and features are introduced to handle object-identity, inheritance, roles and state update. Most of the extensions are based on the possibility of dynamically updating inheritance links. Our prime concern is to provide a logical account for the extensions. This is obtained directly via the definition of a suitable proof-system, and, indirectly, via a meta-logical definition in pure logic programming, on one side, and the translation into a fragment of linear logic, on the other side.

Keywords: Logic programming, object-oriented programming, role dynamics, semantics.

1 Introduction

In recent years, due to the increasing use of logic programming in various application fields, a series of deficiencies has been pointed out and extensions have been proposed to overcome them. For example, the unsuitability of the unification algorithm for numerical applications prompted the design of instances of constraint logic programming.

In this work we focus upon the problem of giving logic programming mechanisms for structuring knowledge and programs. A standard logic program consists of a flat set of Horn clauses, and there is no abstraction mechanism to support programming-in-the-large. Clearly solving such a problem is crucial to make logic programming a well-founded discipline for high-level programming. Many proposals have been made to improve the expressive power of most Prolog implementations [Ber91, Cas91, Mal91, McC91, MLV88, MLV89, Mos93], in which standard logic programming is augmented with constructs for declaring and using modules. In this perspective, we try to improve the usability of logic programming by augmenting it with object-oriented features. It is perhaps the case that "tools that support evolutionary improvements such as object-oriented programming might be more successful than technologies such as logic programming that are often promoted as offering the prospect of a revolutionary advance" [Kow90].

The practical usefulness of an object-oriented logic programming system is twofold. On the one hand, logic programming lacks abstraction and modularization mechanisms, whereas object-oriented

programming is a well-tested technique for the development of modularized programs. On the other hand, object-oriented programming lacks of a formal semantics, whereas logic programming has a well-defined semantics.

A problem in object-oriented programming is the lack of a standard definition. Object-identity, inheritance, and message passing are generally accepted as basic features. However, there are various interpretations of such features (e.g., inheritance can be seen as structural, anti-monotonic, or as delegation-oriented), and different implementations of object-oriented languages stress different aspects, such as persistence or concurrency, depending on the application they aim to cope with. From a knowledge representation perspective, a formalism capable of fully modeling the dynamic and many-faceted nature of real world entities, should take into account the characterization of the various forms of object dynamics - such as state and role evolution. The main problem to obtain such a characterization is that in logic a formula is true or false, indipendently from the time it is evaluated, whereas object states are time-dependent [And91, Ale93].

In our proposal we try to cope with this difficulty trying to mantain a logical semantics. We remark that in a goal-directed interpretation of logic programming, a module defines a context in which goals are evaluated. Standard logic programming provides only a static context: the program. In an object-based approach, the context of evaluation depends on the currently active object. Such a dynamics may be simply one-dimensional, when the program is statically partitioned into modules, or it may be multi-dimensional in the case of an object-oriented system. In particular, we consider three dimensions handling object identity and message passing, inheritance, and state updates and their sequentiality.

The basic choice in our approach is to consider objects and classes as theories in a multi-theories environment [BT93, Kow90, Bro93]. Object-identity is simply modelled by attaching a unique identifier to theories. State is represented as a set of unit clauses and it can be updated via a suitable metapredicate. In order to support update, we need to introduce sequentiality in the evaluation of formulas. Classical logic is unsuitable for providing a formal framework for sequentiality. Many solutions, however, have been proposed to deal with this gap (e.g., logics of action and linear logic). Finally, inheritance is modelled by metapredicates that relate an object to a class, or a class to another class. Such relations can be modified by updating the metapredicates representing them.

The advantages of the approach are twofold. First, we can provide a logical account for our extension. This is obtained directly via the definition of a suitable proof system, and, indirectly, via a metalogical definition in pure logic programming, on one side, and a translation into a fragment of linear logic, on the other side. Second, we obtain a greater expressive power: we can model multiple roles and object migration.

The structure of the paper is as follows. In section 2 we analyze how sequentiality and state update can be espressed in a logical framework. Section 3 introduces the language $\mathcal{OL}$, our proposal of object-oriented logic programming based on the properties we have described. Subsection 3.1 describes the operational semantics of $\mathcal{OL}$ by means of a proof-system, and subsection 3.2 shows how the formalism can be extended to cope with roles. In subsection 3.3 an example is presented aimed at showing the suitability of the $\mathcal{OL}$ formalism to build object-based models. Section 4 gives two interpretations of $\mathcal{OL}$ by using meta-logic and linear logic and finally, in section 5 we discuss the approach.

2 Logical Foundations of Dynamic Object-Oriented Logic Programming

A large amount of research has been devoted to the problem of expressing a notion of local state in object-oriented logic programming. The main approaches we mention can be grouped in two categories: the *clausal approach* and the *concurrent approach*. The definition of the operational semantics by means of a sequent-like system can help to capture the substantial difference between the two approaches.

In declarative languages the execution of a program coicides with the search for a proof. Proofs are carried on in stages, i.e. they transform formulas into formulas. This is the only dynamic component of a logic system. We can expect that each computational model capable of providing a theoretical basis

for OOLP will have the structure of the proof as its basis. Let us consider an interpreter in which the state of the computation is expressed by sequents of the kind $\Gamma \vdash \Delta$. The interpreter can be seen as a rewrite system in which transition rules are expressed by the inference rules of the logic. Our idealized interpreter succeeds if the sequence of rewritings leads to an empty set of sequents, or it fails if there are no applicable rules. So, inference rules can be interpreted, when they are read bottom-up, as actions that lead to a state transition. It has been shown [HM94], that a proof-rule described by means of a sequent system can be automatically translated into an abstract machine.

Consider the structure of a sequent $\Gamma \vdash \Delta$. In the concurrent approach, the dynamic object can be represented in the right part of the sequent, namely the succedent. The various properties of the object can be expressed via logical formulas, which characterize its dynamics during the proof development.

In the clausal approach instead, an object is represented in the left part of a sequent, as a logic theory that describes its methods and its attributes. An object can be seen as a program that interacts with other programs and can change dynamically. The goals in the right side of the sequent represent communication and methods activation.

In the concurrent approach, objects can be seen as processes that evolve during the computation. It is the case, for example, of the pioneering approach by Shapiro and Takeuchi [ST87, Fos91, Dav91, KTMB87]. In this context an operational semantics for a simple concurrent logic programming language can be given via the following inference rules:

$$\frac{}{\Lambda \vdash} \tag{1}$$

$$\frac{\Lambda \vdash \Gamma}{\Lambda \vdash \mathrm{true}, \Gamma} \tag{2}$$

$$\frac{\Lambda \vdash \Delta\theta, \Gamma\theta}{\Lambda \vdash A, \Gamma}, \quad A' \leftarrow \Delta \in \Lambda \wedge \theta = mgu(A, A') \wedge A\theta \equiv A \tag{3}$$

$$\frac{\Lambda \vdash \Gamma\theta}{\Lambda \vdash t_1 = t_2, \Gamma}, \quad \theta = mgu(t_1, t_2) \tag{4}$$

Such a concurrent implementation of logic programming expresses object-orientation as follows:

- an object is a process that retains its internal state by unshared variables;
- an object is represented as a recursive process. The internal state of a process is represented by an atom which holds its private information in unshared variables. Methods corresponds to recursive clauses with the name of the process as predicate as in the following scheme.

```
Object([Msg1|In],State,Out) ←
                    compute(State,Ans),
                    change(State,State1),
                    Out=[Ans|Out1],
                    Object(In,State1,Out1).
⋮
Object([Msgn|In],State,Out) ←
                    compute(State,Ans),
                    change(State,State1),
                    Out=[Ans|Out1],
                    Object(In,State1,Out1).
```

- an instance of an object can be created by reducing a process to another: in the clause

$$A \leftarrow \ldots B \ldots$$

object A creates an instance of object B.

This kind of formalization reaches perhaps its best in the work by Andreoli and others [AP91a, AP91b, And92], describing a concurrent logic language based on linear logic [Gir87, Sce94]. Object-orientation is modeled by means of concurrency and the language permits a clear and simple specification of inheritance. An object can be seen as a multiset of formulas, each of which represents either a property or a method activation request. Non-overriding inheritance can be obtained by increasing or restricting the context of properties, and by adding multiple-headed clauses. In this framework rule (3) is substituted by:

$$\frac{\Lambda \vdash \Delta\theta, \Gamma\theta}{\Lambda \vdash \Upsilon, \Gamma}, \quad \theta = mgu(\Upsilon, \Upsilon') \wedge \Upsilon' \leftarrow \Delta \in \Lambda \tag{5}$$

This rule shows the dynamics of the object (identified with a branch of the proof-tree), i.e. the transition from a state Υ, Γ to a state $\Delta\theta, \Gamma\theta$. Other proposals that improve the expressive power and logical interpretation of message passing have been made [KY92, KY93, ACP92].

The concurrent approach has three main advantages:

- state-change is modeled in a very natural way;
- many techniques for implementing message-passingare are available;
- many higher level specifications (which improve the expressive power: see [Dav91, KTMB87]) can be easily compiled in the language, thus obtaining a simple implementation.

On the other hand, the concurrent approach does not seem so good for representing the "long-term state-change" [Ale93]. In other words, the integration of such a language with a database is too difficult. Furthermore, the specification of inheritance (as delegation or as non-overriding inheritance in the examples above) is too restrictive to provide a good expressive power and a complete integration. Finally, the semantics of languages based on this approach is quite complex, because of the interaction between concurrency and Horn clause logic.

In the clausal approach, an object can be identified with a logic theory in which each clause either describes a method or an attribute (e.g., unit clauses). In this context a goal in the right hand side of a sequent represents a message sent to an object. The evolution of the state of objects is captured by changes in the left-hand side of a sequent. The clausal approach has two main formalizations, one based on higher-order logic and the other based on modal logic. In both cases all aspects of object-orientation are handled in a uniform way, i.e., using a very restricting set of constructs.

Modal logics [Far86], with its *possible-worlds* interpretation, is well suited for representing a conceptualization of multiple objects. Consider the inference system **T** obtained by adding the following rules to the inference rules of classical logic:

$$\frac{A, \Gamma \vdash \Delta}{[i]A, \Gamma \vdash \Delta} \tag{6}$$

$$\frac{\Gamma \vdash A}{[i]\Gamma \vdash [i]A} \tag{7}$$

where modality $[i]$ refers to the i-th agent. Such an inference system allows us to represent a formalization of a multiple-agent world, as it can be seen in the work by Baldano and others [BGM94, GMR92, BLM94]. Moreover, many other aspects of object-oriented programming, such as inheritance, can be formalized in a modal approach. It is the case of the approach proposed in [Uus92], where inheritance and time-depending change are modeled by making use of particular Kripke-structures.

From a declarative point of view, modeling local state-change in the clausal approach is a hard problem, because it is necessary to perform an update of the static knowledge represented in the left side of the sequent. The analogy of object representation with deductive databases, however, suggests us solutions inspired by the theory of updates in databases. In fact, extensional databases can represent the internal state of an object, while intensional databases can represent the set of methods of such an object. So, our problem becomes the problem of updating the extensional database. For example, in [MW86, ?] updates

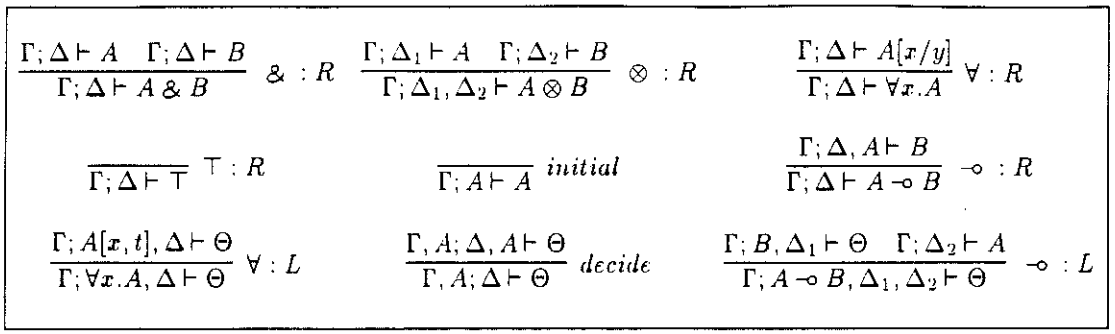

Figure 1: The proof-system for the linear fragment used in the translation.

are formalized by the formulas $assume[p(t)]G$ and $forget[p(t)]G$. The modal prefix $assume[p(t)]$ allows the evaluation of the goal G in an environment extended with the extensional predicate $p(t)$. Similarly, the modal prefix $forget[p(t)]$ causes the goal G to be evaluated in an environment obtained by deleting the extensional predicate $p(t)$. Other related proposals can be found in [BGM95, BK93, AK91].

The modal approach is not, however, the only way to express object-orientation in the clausal approach. A different approach has been proposed by Miller and others [HM90, HM91, Mil89, Mil94, MNPS91]. In their work, they extended Horn clause rules with implications in clause bodies, obtaining a formalization for updates. The main feature of their approach is the use of a fragment of linear logic [Gir87]. Unlike classical logic, linear logic regards formulas as resources which are consumed during the inference process, and not as universally valid (or universally false) assertions. Following such a formalization, class specifications are represented by means of reusable formulas, and the states of objects by means of consumable resources in the left side of the sequent. So, instance creation consists in adding new consumable resources (the attributes) in the left side of the sequent, and instance deletion consists in consuming such formulas. Formulas are built with the following abstract syntax:

$$G \ ::= \ \top \mid A \mid G\,\&\,G \mid G\otimes G \mid \forall x G \mid C\multimap G$$

$$C \ ::= \ \top \mid A \mid C\otimes C \mid \forall x C \mid A\circ\!\!-G$$

The proof-system for such a fragment is given in fig. 2. Each proof is given for a sequent $\Delta;\Gamma\vdash\Lambda$, where the intended meaning is $!\Gamma,\Delta\vdash\Lambda$ and Δ and Γ are C-formulas and Λ is a G-formula. State transition is obtained by means of nested implications. In such a context, the use of higher order logic can be useful, as it is shown by the following program.

$$\begin{aligned}
make(g) &\circ\!\!- sw(Off)\multimap g\\
verify(s,g) &\circ\!\!- sw(s)\,\&\,g\\
set(g) &\circ\!\!- sw(Off)\otimes(sw(On)\multimap g)\\
set(g) &\circ\!\!- sw(On)\,\&\,g\\
reset(g) &\circ\!\!- sw(On)\otimes(sw(Off)\multimap g)\\
reset(g) &\circ\!\!- sw(Off)\,\&\,g
\end{aligned}$$

Figure 2 shows a process of computation for such a program (for more details see [HM91]).

In our opinion, the clausal approach is better suited than the concurrent approach to express object-orientedness in logic programming. In fact, the various features of object-orientation can have a direct logical interpretation and it is easier a merge with deductive databases.

There are many other approaches aimed at integrating logic and object-oriented paradigms. Aït Kaci and others [AKN86, AKP93, BJ94, KLW93], for example, redefine the unification algorithm for letting object-identity and inheritance be implemented in a lattice of objects. This approach, however, does not handle state-change.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{switch; sw(Off) \vdash sw(Off)}\,[\text{identity}] \quad \dfrac{\dfrac{\vdots}{switch; sw(On) \vdash \cdots}}{switch; \emptyset \vdash sw(On) \multimap \cdots}\,[\multimap]}{switch; sw(Off) \vdash sw(Off) \otimes (sw(On) \multimap \cdots)}\,[\otimes]}{switch; sw(Off) \vdash set(\cdots)}\,[\circ\!\!-]}{switch; \emptyset \vdash sw(Off) \multimap set(\cdots)}\,[\multimap]}{switch; \vdash make(set(\cdots))}\,[\circ\!\!-]}{\emptyset; \emptyset \vdash switch \Rightarrow make(set(\cdots))}\,[\Rightarrow]$$

Figure 2: A linear derivation for $\emptyset; \emptyset \vdash switch \Rightarrow make(set(\ldots))$(*switch* represents the program of the previous figure).

3 $\mathcal{OL}$ as an Object-Oriented Formalism in Logic Programming

In this section we present a formal description of our approach. With respect to the classification made in the previous section, $\mathcal{OL}$ can be considered a language based on the clausal approach. As already pointed out, such an approach is more attractive than the concurrent approach, because:

- it allows a more explicit semantic definition of the key features of OOLP
- it has a greater expressive power, i.e. it allows us to capture all the dynamic aspects of the object-oriented data model
- it is very simple and conservative, and is not constrained to any implementation details.

The universe of discourse is represented by a language $\mathcal{L}$ and a set of labels Ω, disjoint from the Herbrand universe $U_{\mathcal{L}}$. Consider the structure $\langle \Omega, \mathcal{L} \rangle$, where $\mathcal{L} = \langle \Pi, \Sigma \rangle$ and $\Omega \cap \Sigma = \emptyset$. The set of terms for the system is built on the alphabet $\Omega \cup \Sigma$ (each symbol in Ω has 0-arity), and the set of predicates can be divided in two subsets: the set of predicates used to code state predicates, Π_s, and the set of method predicates Π_m.

The abstract syntax of $\mathcal{OL}$ is the following:

Program	::=	LbClauses
LbClauses	::=	Label::Clause | Label::Clause $\wedge$ LbClauses
Clause	::=	Atom — Body | Atom.
Body	::=	**true** | Atom | *add*(Atoms) | *del*(Atom)
		| *update*(Atom) | Body $\wedge$ Body | Body $\otimes$ Body | Label::Body
Atom	::=	$p(t_1, \ldots, t_n)$ | Label **isa** Label

where p is a generic predicate symbol, that is $p \in \Pi$.

An $\mathcal{OL}$ program is a set of formulas similar to Horn clauses, with the difference that the atoms and the clauses themselves can be labelled. Each label defines a world. According to a declarative interpretation, each formula must be evaluated in a world and can be either true or false, depending on the degree of knowledge of such a world. On the other hand, according to a procedural interpretation, a clause τ :: *head* ← *body* has the following meaning: *to evaluate the method head in the module* $\tau \in \Omega$, *evaluate the goal body in the environment determined by* τ. Notice that, differently from the procedural reading of definite Horn clauses, the procedural reading for $\mathcal{OL}$ clauses needs an evaluation environment composed by multiple modules (worlds).

The language allows us to support the most important features of object-oriented programming, i.e. class, object-identity, message-passing, inheritance and state evolution.

An object is a logic theory, which retains its internal state by means of private (labelled) formulas and can be referred to by means of a label associated to the theory. A label is unique and it identifies an object. Uniqueness of labels is guaranteed by a generator that exploits existantional quantification as in Kifer et al [KLW93]. A class can be modelled by a set of clauses sharing the same label. Therefore there is no real distinction between classes and objects, as in [Mal91, MLV89].

The formula $Label :: Body$ is used to send the message $Body$ to the object $Label$. According to [Mal91, Kow90, BK82, BRT95] and in the style of Smalltalk, we model message passing as the composition of two primitive operations:

$$message\ passing = lookup \circ apply$$

When a message is sent to an object, the local environment for the object is identified, and the method corresponding to the message is applied. The application of a method is performed as a resolution step, whilst the identification of the execution context is obtained by first looking for the object and then exploiting the inheritance hierarchy. Moreover, self-communication is supported by the special label **self**, which can be seen as a variable unifying with the active object in the context of evaluation. Actually, a label for a formula can be considered as an extra argument for the predicates involved in the formula. So labels can be considered and used as terms.

The inheritance hierarchy is modelled via the metapredicate **isa**, which links two labels. For instance

$$student\ \mathbf{isa}\ person.$$

means that *student* inherits from *person*. We handle inheritance relations as generic attributes of the objects, and provide an ad hoc inference rule for visiting the hierarchy.

We can also support conditional inheritance by allowing bodies specifying conditions about the hierarchical relationship. Take for instance the following clause:

$$\begin{aligned} student :: \mathbf{self}\ \mathbf{isa}\ teacher \leftarrow \\ X :: X\ \mathbf{isa}\ course \\ \wedge\ X :: Assistant(\mathbf{self}). \end{aligned}$$

It states that a student can be considered a teacher if there is a course for which he/she is an assistant. Notice that the inheritance link can be inferred only under certain conditions, that can be false or true depending on the evolution of knowledge. So, if the course for which the student is an assistant terminates, the predicate $X :: X$ **isa** *course* can be inferred no longer and therefore the inheritance link itself cannot be inferred and the student cannot inherit the properties of a teacher.
In standard object-oriented languages, such a situation is modelled with the introduction of a new class, e.g. $teacher - assistant$, which inherits from both *student* and *teacher*. A student which is an assistant is created by instantiating such a class. The drawback of such an approach, however, is that the sole purpose of such an *intersection class* is to allow an instance to be of multiple types: it adds no new state or behaviour. Moreover, the condition can be temporary, as in the case of our example, where the student could be an assistant only for a limited period of time, and only under certain conditions. So, the traditional approach can lead to a combinatorial explosion of sparsely populated classes [MZ86].

In order to support state change, we provide the metapredicates $add(p(t))$, $update(p(t))$ and $del(p(t))$, that respectively add, modify and delete the ground atom $p(t)$ to (from) the current environment. Moreover, to take into account updates, some form of sequentiality must be introduced. In the style of [BK93], we introduce two forms of conjunctions: a parallel one ($\wedge$) and a sequential one ($\otimes$). In the sequential conjunction $F \otimes G$ the right-hand-side formula must be evaluated in the environment obtained by the evaluation of the left-hand-side formula, whilst in the parallel conjunction $F \wedge G$ the formulas must be evaluated in the same initial environment. Each formula has an interpretation as a transition from a Herbrand interpretation to another. In particular, the resulting Herbrand interpretation of an update formula is obtained by deleting or adding the atom involved by the operation itself. A sequential conjunction is interpreted as the transition from the initial interpretation to the interpretation resulting from the evaluation of the right-hand-side formula in the interpretation obtained by the resolution of the left-hand-side formula. On the other hand, a parallel conjunction is interpreted as a transition from

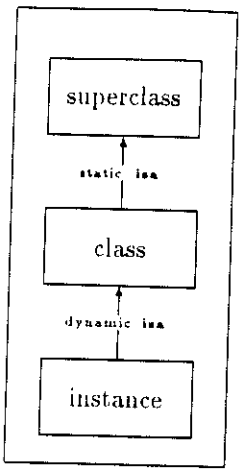

Figure 3: Instantiation as dynamic inheritance.

an initial interpretation to the set-union of the interpretations resulting from the evaluation of the two formulas.

The above features are enough to model object creation. In traditional object-oriented languages, classes can be considered a prototype for object-instances. Classes specify the structure (state) and the behavior (methods) of objects. A class is a type definition, and an object is a variable of that type. This model is not, however, the only possible one. According to [Mal91], in fact, we can think of classes as a collection of statically defined objects, that can interact and own a local state distinguishable from the local state of their instances. Instance creation consists in creating a new, unique label and establishing an inheritance link between the label and the label of the class.

In such a context, we can distinguish two kinds of links that are represented by way of the same **isa** metapredicate (see fig. 3): a static one, and a dynamic one. The definition of the inheritance link by means of **isa** metapredicates allows us either to define the **isa** predicates by means of clauses, or to dynamically change the inheritance relation, by updating on such a predicate. Such a formalization has many advantages.

First of all, it is not constrained to any implementation detail. It is not necessary to have *ad hoc* constructs that express object creation and deletion since they can be programmed directly. A natural way to do this is to define a standard *metaobject* with the task of keeping track of the hierarchy:

$$o_1: \quad \begin{array}{l} object :: new(O, C) \leftarrow \\ \quad new\_label(O) \\ \quad O :: add(O \textbf{ isa } C). \end{array}$$

$$o_2: \quad \begin{array}{l} object :: delete(O) \leftarrow \\ \quad O :: update(O \textbf{ isa } nil). \end{array}$$ [1]

where the resolution of $new\_label(O)$ (quale etichetta ha...) binds O to a new label. *object* is a superclass of each class defined by the programmer. In this way, any class inherits methods to create and delete objects from *object*[1]. Notice that this is not the only way to define *new* and *delete*: other approaches can be easily implemented according to different strategies in storing and handling the primitive operations over the hierarchy.

[1] *nil* can represent any undefined class

[1] Such an approach is rather different, however, from the ObjVLisp (and ObjVProlog: see [?, MLV89]) approach. Here, *object* is defined with the only aim to collect properties common to any class. There *object* and *class* provide the prototype of the structure of any object and class that can be created

Second, it provides a great amount of expressive power, in particular it gives a natural way to deal with object migration and roles (see section 3.2).

3.1 A Proof-System Semantics

We present the operational semantics of the $\mathcal{OL}$ language by means of a proof system. We will adopt the following conventions. x,y,z (possibly with subscripts) will denote object-level variables, C, O, σ, τ (possibly with subscripts) will denote meta-level labels, s, t will denote object-terms, p, q, w, s will denote predicate symbols, f, g will denote function symbols and $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ will denote object-level labels.

We are interested in the derivability of a formula from a program $\mathcal{P}$. The successful evaluation of a formula produces a new environment. We express this derivability notion by means of sequents of the kind $Super\ Self\ Env \longrightarrow \langle Goal, NewEnv\rangle$. $Self$ is the label of the current active object and $Super$ is the label of an anchestor of such an object not necessarily a direct superclass. The evaluation of $Goal$ entails a state transition from the environment Env to the environment $NewEnv$.

Formally, sequents are represented by the ground formula[2] $C\ O\ \gamma \longrightarrow \langle g,\omega\rangle$, where $\gamma,\omega \in Env = \Omega \rightarrowtail 2^{State}$ and $State$ is the Herbrand base for the language $(\Pi_s, \Sigma \cup \Omega)$. A proof-tree for a sequent $C\ O\ \gamma \longrightarrow \langle G,\omega\rangle$ is a tree with the following properties:

- the root is labeled by the sequent;
- leaves have empty labels;
- new nodes are generated by instantiating the following rules:

$$\frac{}{C\ O\ \omega \longrightarrow \langle \mathbf{true},\omega\rangle} \tag{1}$$

$$\frac{C\ O\ \gamma \longrightarrow \langle g_1,\omega_1\rangle \qquad C\ O\ \gamma \longrightarrow \langle g_2,\omega_2\rangle}{C\ O\ \gamma \longrightarrow \langle g_1 \wedge g_2,\omega\rangle}, \omega = \omega_1 \sqcup \omega_2{}^{3} \tag{2}$$

$$\frac{}{C\ O\ \gamma \longrightarrow \langle add(p(t)),\omega\rangle}, \quad \omega(O) = \gamma(O) \cup \{p(t)\} \tag{3}$$

$$\frac{}{C\ O\ \gamma \longrightarrow \langle update(p(t)),\omega\rangle}, \quad \omega(O) = \{p(t)\} \cup (\gamma(O) \prec \{p(t)\}) \tag{4}$$

$$\frac{}{C\ O\ \gamma \longrightarrow \langle del(p(t)),\omega\rangle}, \quad \omega(O) = \gamma(O) - \{p(t)\} \tag{5}$$

$$\frac{C\ O\ \gamma \longrightarrow \langle g_1,\omega_1\rangle \qquad C\ O\ \omega_1 \longrightarrow \langle g_2,\omega\rangle}{C\ O\ \gamma \longrightarrow \langle g_1 \otimes g_2,\omega\rangle} \tag{6}$$

$$\frac{O_i\ O_i\ \gamma \longrightarrow \langle g,\omega\rangle}{C\ O\ \gamma \longrightarrow \langle O_i :: g,\omega\rangle} \tag{7}$$

$$\frac{O\ O\ \gamma \longrightarrow \langle b,\omega\rangle}{C\ O\ \gamma \longrightarrow \langle a,\omega\rangle}, a \leftarrow b \in [C] \cup (\gamma(C)^*)^{5} \tag{8}$$

$$\frac{C_i\ O\ \gamma \longrightarrow \langle C_i\ \mathbf{isa}\ C_j,\gamma\rangle \qquad C_j\ O\ \gamma \longrightarrow \langle a,\omega\rangle}{C_i\ O\ \gamma \longrightarrow \langle a,\omega\rangle} \tag{9}$$

if $name(a) \notin defs(C_i \cup \gamma(C_i)^*)$

[2] For the sake of simplicity we do not consider non-ground goals. However, it is not so difficult to relax the groundness condition in order to provide computed answer substitutions.

[3] $(\omega_1 \sqcup \omega_2)(\tau) = \omega_1(\tau) \cup \omega_2(\tau)$

[4] $\sigma^* = \{A \leftarrow true \mid A \in \sigma\}$, $[C]$ is a subset of $ground(C)$ such that the occurrences of the **self** keyword are replaced with O.

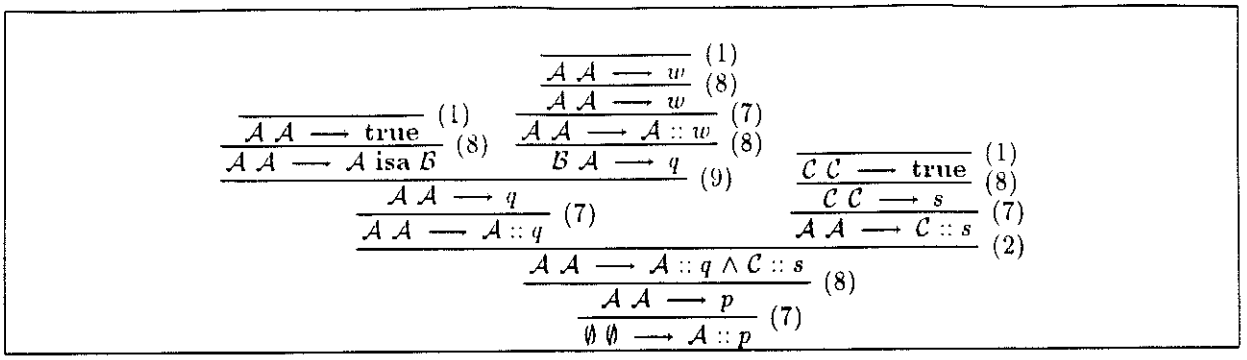

Figure 4: A Proof-tree for $\mathcal{P} \vdash \mathcal{A} :: p$.

In each step of the computation the sequent $C_i\ C_j\ \gamma \longrightarrow \langle F, \omega \rangle$ represents the computation state of an abstract interpreter. The **true** constant is always provable (rule (1)) without causing any state transition. A deductive step (rule 8) is made with respect to the current context identified by C and γ, while context changes are obtained via rules (7) and (9). These rules implement the *lookup* function, while the *apply* function is implemented by rule (8). Rules (2) and (6) express sequentiality, by letting the evaluation of the conjunction of formulas to be done either in parallel or in sequence. An update (rules (3), (4), (5)) for a predicate symbol $p \in \Pi_s$ causes a state transition in which the value for that atom changes. Sometimes we will write $C\ O \longrightarrow g$ if the derivation does not produce any state update. We say that a formula G is derivable from a program $\mathcal{P}$ and from an initial state γ producing an environment ω ($\mathcal{P} \vdash_\omega G$) if there exists a proof-tree $\emptyset\ \emptyset\ \gamma \longrightarrow \langle G, \omega \rangle$.

EXAMPLE 1 *Consider the program $\mathcal{P}$ composed by modules $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$.*

$\mathcal{A} :: \mathcal{A}\ \mathbf{isa}\ \mathcal{B}.$ $\quad \mathcal{B} :: q \leftarrow \mathbf{self} :: w. \quad \mathcal{C} :: s.$
$\mathcal{A} :: p \leftarrow \mathbf{self} :: q \wedge \mathcal{C} :: s.$
$\mathcal{A} :: w.$

Figure 4 describes a proof tree for $\mathcal{P} \vdash \mathcal{A} :: p$. Notice the use of the **self***-label when $\mathcal{A}$ inherits from $\mathcal{B}$ during the proof of q.* □

The next example shows both an object instantiation and an update operation.

EXAMPLE 2 *Consider the program $\mathcal{P}$ composed by modules $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$.*

$\mathcal{A} :: \mathcal{A}\ \mathbf{isa}\ \mathcal{B}.$ $\quad \mathcal{B} :: q \leftarrow update(w(1)). \quad \mathcal{C} :: s \leftarrow \mathcal{A}_1 :: w(1).$
$\mathcal{A} :: create(X) \leftarrow object :: new(X, \mathbf{self}).$
$\mathcal{A} :: p \leftarrow \mathbf{self} :: q \otimes \mathcal{C} :: s.$

Figure 5 is a proof-tree for $\mathcal{P} \vdash_\omega \mathcal{A} :: create(\mathcal{A}_1) \otimes \mathcal{A}_1 :: p$. From the initial sequent we can construct, by rule (6), the subtrees Ξ_1 and Ξ_2. Ξ_2 further develops into the subtrees Ξ_3 and Ξ_4. Finally, we obtain $\omega = \omega_2$, where:

$$\omega_2(\tau) = \begin{cases} \{w(1), \mathcal{A}_1\ \mathbf{isa}\ \mathcal{A}\}, & \text{if } \tau = \mathcal{A}_1 \\ \emptyset, & \text{otherwise} \end{cases}$$

$$\omega_1(\tau) = \begin{cases} \{\mathcal{A}_1\ \mathbf{isa}\ \mathcal{A}\}, & \text{if } \tau = \mathcal{A}_1 \\ \emptyset, & \text{otherwise} \end{cases}$$

Subtree Ξ_1 shows how the instance is obtained (the environment created is ω_1). In subtree Ξ_2 the evaluation of the formula $\mathcal{A}_1 :: p$ is made. In subtree Ξ_3 the new environment ω_2 is created. Finally, notice that in Ξ_3 both the dynamic and the static **isa** *link are used.* □

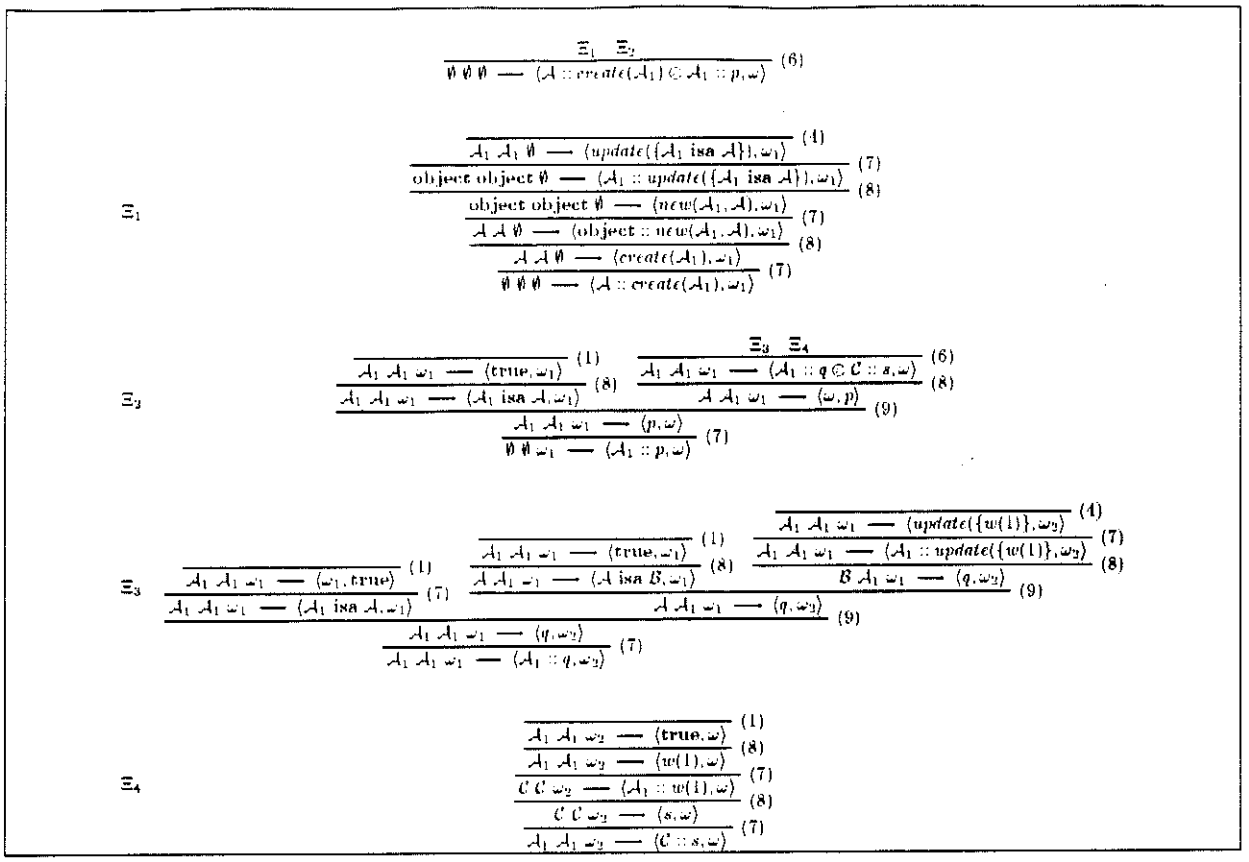

Figure 5: A Proof-Tree for $\mathcal{P} \vdash_\omega \mathcal{A} :: new(\mathcal{A}_1) \otimes \mathcal{A}_1 :: p$

3.2 Role Dynamics

The formalization of objects as theories combined with the notion of programmable **isa** provides a natural way to deal with object migration and roles. Traditional object-oriented languages are not able to fully model the dynamic nature and the many-faceted nature of common, real-world entities. The intimate and permanent binding of an object's identity to a single type inhibits most object-oriented systems to track accurately a real-world entity over time. Take the case of a person, who can become first a student and then change its status to being a worker (migration of an object from a class to another), or a teacher who can simultaneously be a professional (multiple roles).

On the contrary, the $\mathcal{OL}$ formalism can easily model these features, by allowing a direct manipulation of the **isa** metapredicate. Since objects are bound to classes by means of the **isa** predicate, we can update, delete and add new links, thus realizing object-role dynamics.

To handle role-dynamics we can define a metaobject *dynamics* as follows:

$$dynamics :: extend\_object(O, R) \leftarrow O :: add(O \textbf{ isa } R). \qquad d_1$$

$$dynamics :: drop\_role(O, R) \leftarrow O :: del(O \textbf{ isa } R). \qquad d_2$$

Method d_1 allows us to add the role R to the object O, by adding an inheritance link between O and R. In such a way, the object O inherits all the properties of the role R, and consequently its behaviour. Conversely, method d_2 deletes the link between O and R, and consequently O looses the behaviour described by R. For example, suppose that *giuseppe* **isa** *student* is provable in the current

state. The execution of the method *dynamics* :: *extend_object*(*giuseppe*, *employee*) adds the predicate *giuseppe* **isa** *employee* in the current state. Now, *giuseppe* can behave both as a student and as an employee.

However, problems concerning the maintenance of consistent information can arise. Since the role of an object is updated by *simply* handling the **isa** metapredicate, no effective deletion of the attributes of the object referring to the updated role is made. Such a situation can be very attractive on one hand because it allows the maintenance of a sort of history for the object life-cycle, but it can lead to an inconsistent situation on the other hand. Suppose that *giuseppe* has an attribute *code* whose value is 12*ad* as a *student*, that is *code*(12*ad*) is in *giuseppe*'s current state, and that the class *employee* has, among its attributes, the *code* attribute, too. Consider now the goal *dynamics* :: *drop_role*(*giuseppe*, *student*). *giuseppe* looses the role *student*, but the state attributes of the previous role are mantained, and therefore if somebody asks *giuseppe* what is his *code*, he can answer with the student code instead of the employee code.

To face this kind of conflicts, we need a more structured approach. Since an object can assume a set of roles, it is natural to refer to such an object only with respect to a certain role. So, the attribute *code* is referred in the current object with respect to one of the roles the object assumes, and it is meaningless to query the object for *code* indipendently from the role. Moreover, we need to distinguish the **isa** predicate from the other state predicates, because the **isa** predicate specifies the actual properties of an object with respect to the schema, whereas the state predicates are related to all the roles the object has assumed. We can extend the proposed formalization by giving each state predicate an *invisible* label representing the actual role in which the object finds the state predicate definition. The idea is to exploit these labels when solving a goal, choosing only the state predicates related to the current role of the object. In our example this will allow us to discard the clause *student* : *code*(12*ad*) in the state and to select the right definition for code, that is *employee* : *code*(*emp*31).

The role an object assumes is made explicit by the new prefix O *as* C, where O is the object-identifier for the object and C is the role which O assumes. The formula O *as* $C :: G$ means that G must be evaluated in object O that behaves as C. Now, the proof-system has to be modified in the following way:

$$\frac{}{C_i\ C_j\ O\ \omega \longrightarrow \langle \mathbf{true}, \omega \rangle} \tag{10}$$

$$\frac{C_i\ C_j\ O\ \gamma \longrightarrow \langle g_1, \omega_1 \rangle \qquad C_i\ C_j\ O\ \gamma \longrightarrow \langle g_2, \omega_2 \rangle}{C_i\ C_j\ O\ \gamma \longrightarrow \langle g_1 \wedge g_2, \omega \rangle}, \omega = \omega_1 \sqcup \omega_2{}^3 \tag{11}$$

$$\frac{}{C_i\ C_j\ O\ \gamma \longrightarrow \langle add(p(t)), \omega \rangle}, \quad \omega(O) = \gamma(O) \cup \{C_j : p(t)\}, p \neq \mathbf{isa} \tag{12}$$

$$\frac{}{C_i\ C_j\ O\ \gamma \longrightarrow \langle add(\tau\ \mathbf{isa}\ \sigma), \omega \rangle}, \quad \omega(O) = \gamma(O) \cup \{\tau\ \mathbf{isa}\ \sigma\} \tag{13}$$

$$\frac{C_i\ C_j\ O\ \gamma \longrightarrow \langle G_1, \omega_1 \rangle \qquad C_i\ C_j\ O\ \omega_1 \longrightarrow \langle G_2, \omega \rangle}{C_i\ C_j\ O\ \gamma \longrightarrow \langle G_1 \otimes G_2, \omega \rangle} \tag{14}$$

$$\frac{O_k\ O_k\ O_k\ \gamma \longrightarrow \langle O_k\ \mathbf{isa}\ C, \gamma \rangle \qquad C\ C\ O_i\ \gamma \longrightarrow \langle G, \omega \rangle}{C_i\ C_j\ O\ \gamma \longrightarrow \langle O_k\ as\ C :: G, \omega \rangle} \tag{15}$$

$$\frac{O_j\ O_j\ O_j\ \gamma \longrightarrow \langle G, \omega \rangle}{C_i\ C_j\ O\ \gamma \longrightarrow \langle O_j :: G, \omega \rangle} \tag{16}$$

$$\frac{C_j\ C_j\ O\ \gamma \longrightarrow \langle G, \omega \rangle}{C_i\ C_j\ O\ \gamma \longrightarrow \langle A, \omega \rangle}, A \leftarrow G \in [C_i] \tag{17}$$

$$\frac{}{C_i\ C_j\ O\ \gamma \longrightarrow \langle A, \gamma \rangle}, \quad C_j : A \in \gamma(O) \text{ or } name(A) = \mathbf{isa} \text{ and } A \in \gamma(O)$$

$$\frac{C_i\ C_j\ O\ \gamma \longrightarrow \langle C_i \text{ isa } C, \gamma\rangle \qquad C\ C_j\ O\ \gamma \longrightarrow \langle A, \omega\rangle}{C_i\ C_j\ O\ \gamma \longrightarrow \langle A, \omega\rangle} \quad (18)$$

if $name(A) \notin defs(C_i \cup \gamma(O))$

Each sequent specifies also the current role of the object, that is in a sequent $C_i\ C_j\ O\ \gamma \longrightarrow \langle G, \omega\rangle$ O specifies the current object, C_j the current role, C_i an ancestor of the role or the role itself.

Rules (10), (11), (14), (16), (18) differ from the corresponding (1), (2), (6), (7) only for the role component in the sequent.

Rule (??) allows us to add a state predicate with the right invisible label. This label is the current role of the object, that is C_j. Since the **isa** predicate is a predicate wherever visible, in the state of objects it is not labelled by an invisible label (rule (??)). Rules (4) and (5) must be replaced in an analogous way as rule (??). Rule (8) is divided in two rules to deal with clauses and state predicates respectively. If name(A) is a state predicate, the current value of this predicate is in the state of the object bound to the current role of the object (rule 18).

Since an object can see only the classes it is linked to, the set of attributes and methods, it can use to solve the goal, are only the ones reachable by following the actual inheritance links. Rule (??) defines a deductive step for methods and **isa** predicates, whereas rule (??) defines a deductive step for state predicates. In this way, if the object has assumed different roles, we select the state predicate values of the current role.

3.3 An Example

Suppose a high school office is composed by the following entities (see fig. 6):

- the entity school with properties curricula;
- the entity course with properties name and term;
- the entity person with properties name, sex, age, birth, lives_in; this class has two subclasses, student and employee, that are not disjoint;
- the entity student with properties passed_exam, exams_to_take, add_exam, average, studies_in, degree, code, info;
- the entity employee with properties works_in, code; teacher is a subclass of employee;
- the entity teacher with properties e_mail, info;

The relationships among the entities are:

- the relationship between school and teacher, that represents the fact that a teacher works in a certain school;
- the relationship between school and student, that represents the fact that a student attends a certain school;
- the relationship between school and course that represents the courses provided by a certain school;
- two relationships between course and teacher representing the teacher and the assistant of a course respectively.

Figure 7 presents the $\mathcal{OL}$ program modeling the school office in an object-oriented style. Each class is represented by the set of clauses with the same label: student, teacher, school, person, employee, course. Notice that each class is characterized by two different kinds of clauses. The definite clauses define the methods, and the unit clauses define the **attribute** predicate, representing the state predicates for the class. It is worth noting that even if the state predicates *name* and *age* are used inside *student* (*info*

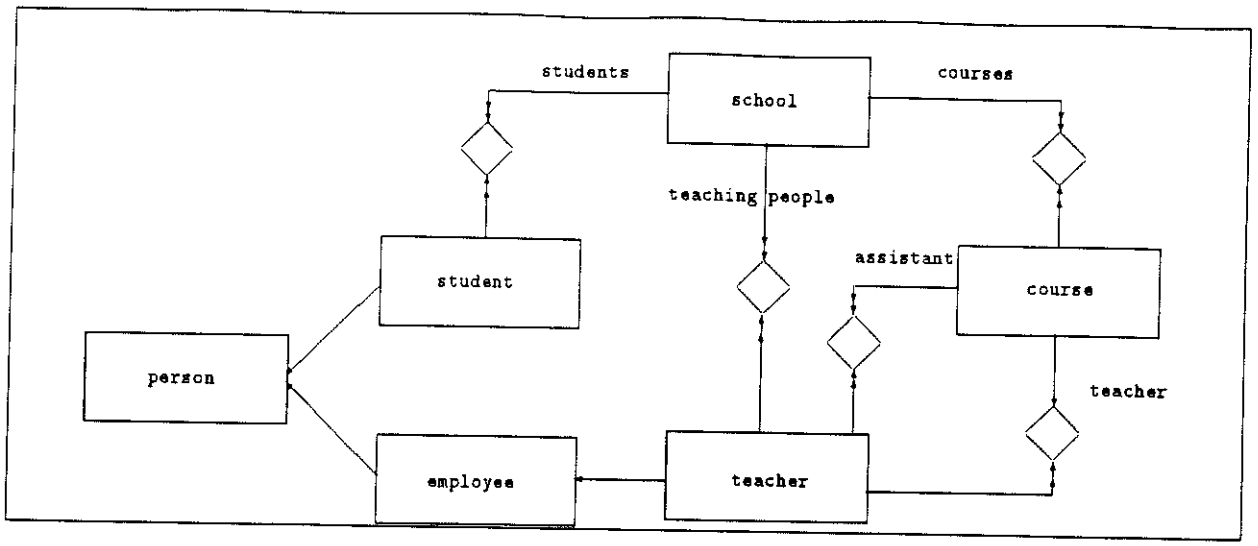

Figure 6: A graphical representation of the school office.

method), there is no **attribute** clause for them because these predicates are inherited from *person*. Consider for example the following clause for the computation of the exams still to be passed by a student.

$$\begin{aligned} &student :: exams\_to\_take(Xs) \leftarrow \\ &\quad degree(S) \\ &\quad \wedge\ S :: curricula(\mathbf{self}, Ys) \\ &\quad \wedge\ bagof(E, \mathbf{self} :: exam(E, V, D), Zs) \\ &\quad \wedge\ difference(Ys, Zs, Xs). \end{aligned}$$

Intuitively, in order to find the exams the student has to take, we need to look for his/her curriculum, and then compute the difference between the exams he/she has chosen and those he/she has already passed. $degree(S)$ is a state predicate defining the school that the student attends. Notice that, the literal is not labelled, and therefore it has to be solved in the local environment of the object representing the student. The value bound to S is the object identifier of the school. Such a label is used to send a message to the object itself, in order to find the curriculum chosen by the student. Finally, the predicates *bagof* and $difference$ compute the list of exams passed by the student and the difference between such a list and the list in curriculum respectively. It is worth remarking that **self** is used as an argument of the predicate *curricula*, and as a label for a formula in the predicate *bagof*, too. In both cases it is bound to the active object.

In order to establish the hierarchical links among classes, we introduce the clauses defining the **isa** metapredicate. For example, the links for a student are described by the following clauses already presented in section 3

$$\begin{aligned} s_1: &\quad student :: student\ \mathbf{isa}\ person. \\ &\quad student :: \mathbf{self}\ \mathbf{isa}\ teacher \leftarrow \\ s_2: &\quad X :: X\ \mathbf{isa}\ course \\ &\quad \wedge\ X :: assistant(\mathbf{self}). \end{aligned}$$

The method for adding new exams to an instance of the class *student* is modelled by the following clause

$$\begin{aligned} s_3: &\quad student :: ins\_exam(E, V, D) \leftarrow \\ &\quad add(exam(E, V, D)). \end{aligned}$$

It defines a transition from the initial database of exams of a *student* object to the final database in which the information related to the new exam is added.

```
% clauses for the class student
student :: average(X) ←
    bagof(V, self :: passed_exam(E, V, Y), Vs)
     ∧ count(Vs, X).
student :: exams_to_take(Xs) ←
    degree(S)
     ∧ S :: curricula(self, Ys)
     ∧ bagof(E, self :: passed_exam(E, V, D), Zs)
     ∧ difference(Ys, Zs, Xs).
student :: info(Name, Age, Code) ←
    name(Name)
     ∧ age(Age)
     ∧ code(Code).
student :: student isa person.
student :: self isa teacher ←
    X :: X isa course
     ∧ X :: Assistant(self).
student :: add_exam(E, V, D) ←
    add(exam(E, V, D)).
student :: passed_exam attribute student.
student :: degree attribute student.
student :: code attribute student.
student :: studies_in attribute student.

    % clauses for the class teacher
    teacher :: info(Name, Age, Addr) ←
        age(Age) ∧ lives_in(Addr)
         ∧ name(Name).
    teacher :: teacher isa employee.
    teacher :: e_mail attribute teacher.

% clauses for the class school
school :: curricula attribute school.
school :: teaching_people attribute school.
school :: courses attribute school.

% clauses for the class person
person :: lives_in(X) ←
    works_in(X).
person :: lives_in(X) ←
    studies_in(X).
person :: age(X) ←
    this_year(Y)
     ∧ birth(_, _, Z)
     ∧ minus(Y, Z, X).
person :: name attribute person.
person :: sex attribute person.
person :: birth attribute person.
person :: lives_in attribute person.

% clauses for the class employee
employee :: works_in attribute employee.
employee :: code attribute employee.
employee :: eployee isa person

% clauses for the class course
course :: name attribute course.
course :: term attribute course.
course :: assistant attribute course.
course :: teacher attribute course.
```

Figure 7: $\mathcal{OL}$ code for the school office.

Finally, consider the following clauses for the class person

$$p_1: \quad person :: lives\_in(X) \leftarrow works\_in(X).$$

$$p_2: \quad person :: lives\_in(X) \leftarrow studies\_in(X).$$

Suppose that the following instances of *student* and *teacher* are available in the current state.

$$\omega(giuseppe) = \{giuseppe \textbf{ isa } student. \; studies\_in(pisa). \; degree(computer\ science). \; name(giuseppe\ manco).\}$$

$$\omega(franco) = \{franco \textbf{ isa } teacher. \; lives\_in(livorno). \; works\_in(pisa).\}$$

Consider now the query $\leftarrow giuseppe :: lives\_in(X)$. The inference system takes all the clauses associated with the object *giuseppe* as the actual context of evaluation. Then the *lookup* function is applied. Since no definition for *lives_in* is in *giuseppe*, an inheritance link is looked for, that is an attempt to prove the query $\leftarrow giuseppe$ **isa** $Super \wedge Super :: lives\_in(X)$ is made, and a link is established with the class *person* because of clause s_1. Now, clause p_2 unifies with the second atom of the query. Then the original query is reduced to $\leftarrow giuseppe :: studies\_in(X)$, and is solved with correct answer substitution $X = pisa$. On the contrary, the query $\leftarrow franco :: lives\_in(pisa)$ fails, because the predicate *lives_in* is defined in *franco* itself and so, by way of the overriding mechaninsm, the clause p_1 cannot be applied.

Finally, fig. 8 shows an example of computation of role-change for the object g. The figure shows the 5 branches of the proof-tree for the sequential conjunction

$$object :: new(g, student) \otimes g :: add(code(a)) \otimes g :: add(g \textbf{ isa } employee) \otimes g :: del(g \textbf{ isa } student) \otimes g :: add(code(b))$$

Notice that, after the evaluation of Ξ_2, the local state for g contains the fact $student : code(a)$, visible by rule ?? since g **isa** *student* is provable. After the addition of the new role *employee* and the deletion of the originary role *student*, the addition of the attribute *code* is made with respect to the role *employee*, and the fact $student : code(a)$ is no more visible.

4 Logical Roots of $\mathcal{OL}$

From a proof-theoretic point of view, the $\mathcal{OL}$ formalism is a conservative extension of logic programming. Consider for instance a traditional logic program, that is a set of Horn clauses without update atoms nor labelled atoms. Such a program can be considered as an $\mathcal{OL}$ program composed by a single module and in which the conjunction of atoms is interpreted as the parallel conjunction in $\mathcal{OL}$. Then the operational semantics of $\mathcal{OL}$ is equivalent to the traditional operational semantics of logic programming.

Here, we show the rooting of $\mathcal{OL}$ in logic in two other respects: by axiomatizing the proof-theory of $\mathcal{OL}$ via a metainterpreter written in logic programming, and by showing a direct interpretation of $\mathcal{OL}$ in terms of linear logic. The reading of the metainterpreter is straightforward and, most importantly, the meta-logical definition shows that $\mathcal{OL}$ can be expressed within logic programming itself. The semantics characterizations of logic programs can be applied to $\mathcal{OL}$ programs by applying them to their metalogical definition.

The interpretation of $\mathcal{OL}$ in terms of linear logic shows how it naturally fits into a theoretical framework explicitly designed to provide an account for state change.

4.1 Meta-logical definition of $\mathcal{OL}$

We present a meta-logical definition of the language $\mathcal{OL}$. The meta-logical definition is obtained by adding new clauses to the vanilla metainterpreter [SS86]. It is worth noting that meta-logic provides an executable specification of $\mathcal{OL}$.

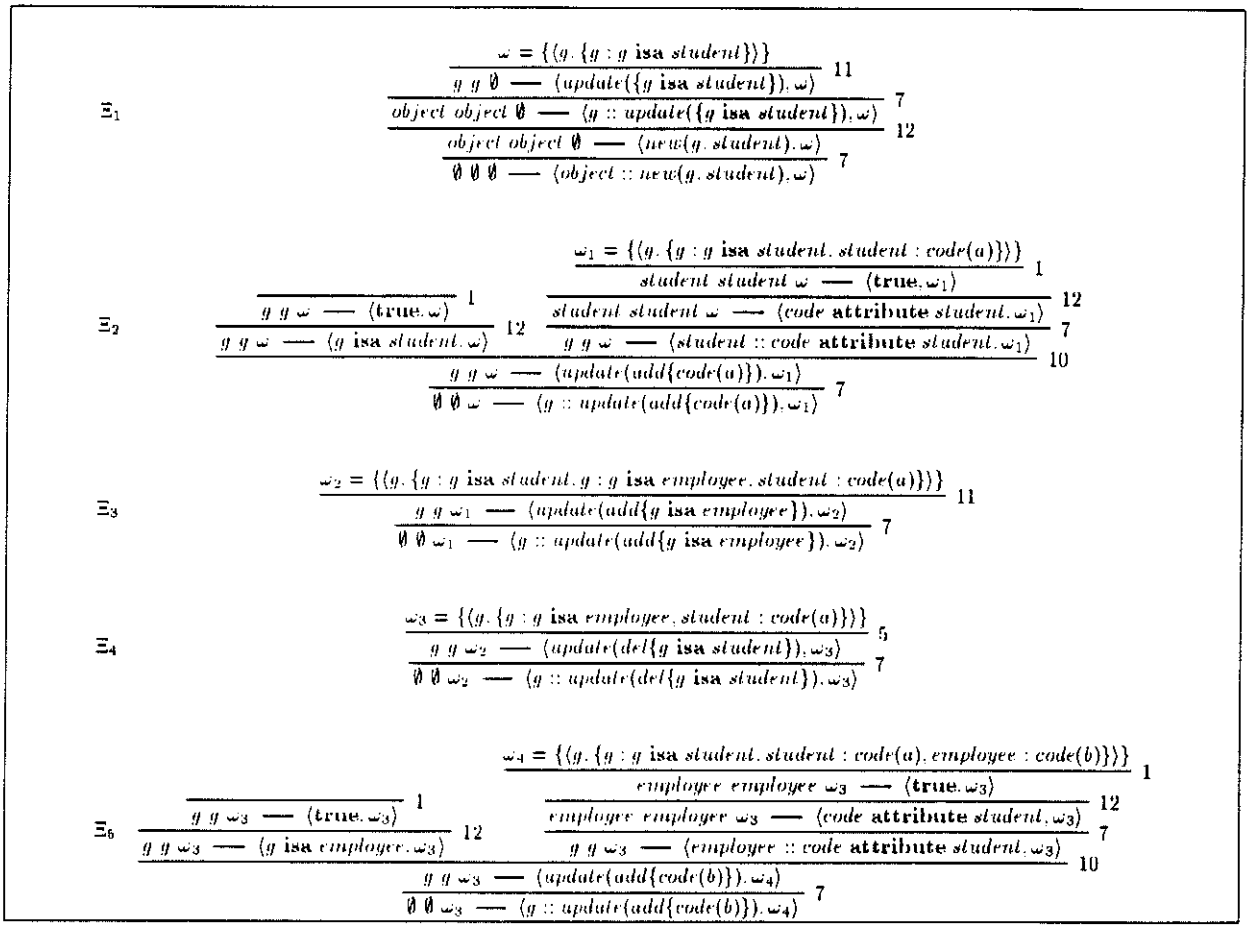

Figure 8: A fragment of proof-tree for the computation of the change of a role.

We extend the basic framework of logic programming by considering a collection of logic programs and a union composition operator defined over them. The union operator (denoted by $\cup$) allows one to compose theories into a single theory according to the following definition.

Definition 4.1 *Given two theories P and Q, $P \cup Q$ denotes the theory obtained by putting the clauses of theories P and Q together.* □

This approach is developed in [Bro93, BT93, BMPT94]. In the following we use the $\cup$ operator to represent the state of an object.

We employ a five argument predicate *demo* to represent the provability relation. Namely, $demo(\mathcal{E}, C, Obj, G, \mathcal{E}')$ states that the goal G has to be solved by using the clauses of C and the multi-object environment $\mathcal{E}$. The resolution of G can modify the multi-object environment and the resulting environment is $\mathcal{E}'$.

The multi-object environment represents the set of objects that have been created up to now. For each object, we are interested in its name and its current state. This set is denoted by a sequence defined by the following abstract syntax:

$$Env ::= \lambda \mid (IdO, S) \mid Env \ \& \ Env$$

where IdO is an object identifier and S is a theory formed by theories defining only state predicates and composed together via the $\cup$ operation. $\mathcal{E}, \mathcal{E}' \in Env$ are sequences constructed by & operator, and they give all the information we need about objects. λ represents the empty sequence: no object has yet been created. Via the *update* predicate we replace tuples of the environment with new ones, where only the state has changed. In order to select the state associated with an object identifier IdO in an environment $\mathcal{E}$, we provide a predicate $select(IdO, \mathcal{E}, S, E_1, E_2)$.

$$select(IdO, \mathcal{E}, S, E_1, E_2) \quad \leftarrow \quad \mathcal{E} = E_1 \& (IdO, S) \& E_2.$$
$$select(IdO, \mathcal{E}, \{\}, \lambda, \mathcal{E}) \quad \leftarrow \quad \mathcal{E} \neq E_1 \& (IdO, \_) \& E_2.$$

The clauses state that if the object IdO has been created (i.e., it belongs to the environment $\mathcal{E}$), then S is its current state. Otherwise, the object IdO has not yet been created, and therefore its state is empty.

The second argument of the *demo* predicate is the name of a class or an object (remember that there is no difference between these two notions in our model). This name refers to the set of clauses of the kind $C :: clause$ where C is such a name.

The third argument of the *demo* predicate is the identifier of the active object. Any change, determined by the resolution of the current goal, affects the state of such an object.

The fourth argument of the *demo* predicate is just the current goal.

The theory associated with a class $IdeCl$ is named by $IdeCl$ and it is represented at the meta-level by a set of axioms of the kind $clause(IdeCl, X, A \leftarrow B) \leftarrow$, one for each object level clause of the kind $IdeCl :: A' \leftarrow B'$ where the **self** keyword in $A' \leftarrow B'$ is replaced by X. X is a variable not occurring in $A' \leftarrow B'$.

The possibility of changing the state of objects implies the dynamic construction of theories. The following axiom extends the representation of object level theories by means of constant names by supporting the reference to unnamed theories.

$$clause(\{X \leftarrow Y\}, O, X \leftarrow Y) \leftarrow \qquad \underline{\text{Unnamed theory}}$$

The state of an object can be considered as a knowledge base Kb consisting of unit ground clauses only. Basic updates on a knowledge base are the insertion and the deletion of formulas. The predicate $update(Kb, U, Newkb)$ states that knowledge base Kb evolves into $Newkb$ due to the update U. The following axioms formally define the three kinds of updates.

$$update(Kb, insert(A), \{A \leftarrow\} \cup Kb) \leftarrow \qquad \underline{\text{insert}}$$

$$update(\{\}, delete(A), \{\}) \leftarrow \qquad \underline{\text{del1}}$$

$$update(\{A \leftarrow\} \cup Kb, delete(A), Newkb) \leftarrow update(Kb, delete(A), Newkb) \qquad \underline{\text{del2}}$$

$$update(\{B \leftarrow\} \cup Kb, delete(A), \{B \leftarrow\} \cup Newkb) \leftarrow A \neq B, \; update(Kb, delete(A), Newkb) \qquad \underline{\text{del3}}$$

In the above axioms, a knowledge base is represented by the union of its clauses. The axioms rely on the basic properties (idempotence, associativity and communativity) of the theory-composition operator union.

The axioms modelling the union operator are the following:

$$clause(\mathcal{P} \cup \mathcal{Q}, Obj, A \leftarrow G) \leftarrow clause(\mathcal{P}, Obj, A \leftarrow G) \qquad \underline{\text{Union1}}$$

$$clause(\mathcal{P} \cup \mathcal{Q}, Obj, A \leftarrow G) \leftarrow clause(\mathcal{Q}, Obj, A \leftarrow G) \qquad \underline{\text{Union2}}$$

The above clauses state that an $\mathcal{OL}$ clause $A \leftarrow G$ belongs to the union of two programs $\mathcal{P}$ and $\mathcal{Q}$, if it belongs to either $\mathcal{P}$ or $\mathcal{Q}$.

The following four clauses extend the standard vanilla metainterpreter for logic programs with extra arguments.

$$demo(\mathcal{E}, C, Obj, true, \mathcal{E}) \leftarrow \qquad \underline{\text{true}}$$

This clause states that the goal true is solved in any class and it does not change the environment $\mathcal{E}$.

$$demo(\mathcal{E}, C, Obj, (G_1 \wedge G_2), \mathcal{E}') \leftarrow demo(\mathcal{E}, C, Obj, G_1, \mathcal{E}''), \; demo(\mathcal{E}'', C, Obj, G_2, \mathcal{E}') \qquad \underline{\text{Conjunction 1}}$$

$$demo(\mathcal{E}, C, Obj, (G_1 \otimes G_2), \mathcal{E}') \leftarrow demo(\mathcal{E}, C, Obj, G_1, \mathcal{E}''), \; demo(\mathcal{E}, C, Obj, G_2, \mathcal{E}'''), \; merge(\mathcal{E}'', \mathcal{E}''', \mathcal{E}'). \qquad \underline{\text{Conjunction 2}}$$

Those clauses deal with conjunctive parallel and sequential goals. A conjunction $G_1 \wedge G_2$ is solved by proving G_1 and G_2 starting with the same environment $\mathcal{E}$, and merging the resulting environments (the predicate *merge* returns the set-union of the related environment). A conjunction $(G_1 \otimes G_2)$ is solved in C if G_1 is solved in C and G_2 is solved in C and with respect to the new environment $\mathcal{E}''$, obtained from the computation of G_1,

$$demo(\mathcal{E}, C, Obj, A, \mathcal{E}') \leftarrow clause(C, Obj, A \leftarrow G), \; demo(\mathcal{E}, Obj, Obj, G, \mathcal{E}') \qquad \underline{\text{Atom resolution 1}}$$

$$demo(\mathcal{E}, C, Obj, A, \mathcal{E}) \leftarrow select(Obj, \mathcal{E}, S, E_1, E_2), \; clause(S, Obj, A \leftarrow empty) \qquad \underline{\text{Atom resolution 2}}$$

These clauses state that an atomic goal A is provable if a clause $A \leftarrow G$ belongs to either the class C or to the environment $\mathcal{E}$, and G is recursively solved in C.

The following clause models inheritance:

$$demo(\mathcal{E}, C, Obj, A, \mathcal{E}') \leftarrow undefined(\mathcal{E}, C, A), \; demo(\mathcal{E}, C, Obj, C \text{ isa } K, \mathcal{E}''), \; demo(\mathcal{E}'', K, Obj, A, \mathcal{E}') \qquad \underline{\text{inheritance}}$$

This clause enriches the set of atoms provable in C by exploiting the hierarchical link between classes. The predicate *undefined* expresses the condition $name(A) \notin defs(C \cup S)$, such that $\mathcal{E} = E_1 \& (C, S) \& E_2$. Notice that we exploit backtracking to visit all the superclasses of C.

The next clauses deal with *update*.

$$demo(\mathcal{E}, C, Obj, add(A), \mathcal{E}') \leftarrow select(Obj, \mathcal{E}, S, E_1, E_2), \; update(S, insert(A), S'), \; \mathcal{E}' = E_1 \& (Obj, S') \& E_2 \qquad \underline{\text{update 1}}$$

$$demo(\mathcal{E}, C, Obj, del(A), \mathcal{E}') \leftarrow select(Obj, \mathcal{E}, S, E_1, E_2), \quad update(S, delete(A), S'), \quad \mathcal{E}' = E_1 \& (Obj, S') \& E_2 \qquad \text{update 2}$$

Handling the update of existing atoms implies the definition of another composition operator. We omit this extension here, also because from a programming view point update can be substituted by use of insert and delete.

This clause states that solving an *update* goal in the object Obj entails knowledge assimilation operations.

$$demo(\mathcal{E}, C, Obj, O :: G, \mathcal{E}') \leftarrow demo(\mathcal{E}, O, O, G, \mathcal{E}') \qquad \text{msg}$$

This clause states that a goal of the form $O :: G$ is provable in C if the goal G is provable in O.
By exploiting unification and backtracking, we can use the above clause to search for an object where the goal G can be solved. This exploitation of the logical variable offers a mechanism much more powerful than the ones supported by traditional object-oriented languages.

4.2 $\mathcal{OL}$ and Linear Logic

In this section we will provide another formal viewpoint on the evalution of sequentiality and assignment in $\mathcal{OL}$. We will show how the core properties of $\mathcal{OL}$ can be expressed in a first order fragment of linear logic. For the sake of simplicity we will deal with the fragment of the language in which only the use of sequential conjunction and of the metapredicates $add(p(t))$ and $del(p(t))$ are allowed. In such a way, we will not deal with multiple objects and therefore either message passing or inheritance. Moreover, method predicates have to be distinguished from state predicates, i.e., the sorts Π_s and Π_m are disjoint (clearly such a restriction is not substantial).

We will follow the basic concepts of [BG95] in formalizing the interpretation, properly extending the underlying ideas in order to model state update. Suppose we have the clause

$$A_0 \leftarrow A_1 \otimes \ldots \otimes A_n$$

where $n \geq 0$. The idea is to delay the evaluation of A_i until A_{i-1} is not evaluated, by means of a new predicate symbol undefined in the program. The new predicate is bound to the evaluation of A_{i-1} by means of an extra variable. The role of the new predicate is to "witness" the successful evaluation of A_{i-1}.

We now provide the notation that will be used in the translation of $\mathcal{OL}$ programs. A sequence is defined as usual, and the concatenation of two sequences k and h is written as $k \cdot h$. Suppose that the set V of variables is numerable, and that there is a bijective map from sequences k to natural numbers $\overline{k}$. So, given a sequence k, it is possible to associate to it a variable $i_{\overline{k}}$. We now define the functions $bot[G]_k$ and $tot[G]_k$, which associate to each goal the last and the entire set of variables respectively, according to the sequential order in which the atoms appear.

Definition 4.2 *Let G be an $\mathcal{OL}$ goal. Then $bot[G]_k$ and $tot[G]_k$ are recursively defined as:*

- $bot[A]_k = tot[A]_k = i_{\overline{k}}$, *where* $A \equiv p(t)$ *or* $A \equiv$ **true**;
- $bot[G_1 \otimes G_2]_k = bot[G_2]_{k \cdot 2}$, $tot[G_1 \otimes G_2]_k = tot[G_1]_{k \cdot 1} \cdot tot[G_2]_{k \cdot 2}$.

□

The translation is defined as follows.

Definition 4.3 *Let G be a goal formula. Then the translation of G, denoted as $[\![G]\!]_k$, is defined recursively as:*

- $[\![p(t)]\!]_k = p(t, i_{\overline{k}})$;

- $[\![\mathbf{true}]\!]_k = \vartheta(i_{\overline{k}})$;
- $[\![G_1 \otimes G_2]\!]_k = ([\![G_2]\!]_{k\cdot 2} \multimap \vartheta(bot[G_1]_{k\cdot 1})) \multimap [\![G_1]\!]_{k\cdot 1}$.

□

Intuitively, in a sequential conjunction $G_1 \otimes G_2$ G_2 must be evaluated only after the successful evaluation of the last atom in G_1. The translation binds the resolution of G_2 to the resolution of the witness predicate ϑ, that, in turn, is bound to the last atom in G_1 by means of the last variable associated to G_1. Clearly, if the goal is an atom, the variable must be bound to the resolution of the atom itself, and if the goal is the constant **true** the goal is automatically solved.

Definition 4.4 *Consider a program $\mathcal{P}$ and a clause $A \leftarrow G$. Then*

- $[\![A \leftarrow G]\!] = \forall\ bot[A]_0 x_1 \ldots x_n.([\![A]\!]_0 \circ\!\!- \forall tot[G]_1.((\vartheta(bot[A]_0) \multimap \vartheta(bot[G]_1)) \multimap [\![G]\!]_1)$, *where $x_1, \ldots, x_n$ are the variables occurring in $A \leftarrow G$;*
- $[\![\mathcal{P}]\!] = \bigcup_{C \in \mathcal{P}} [\![C]\!]$.

□

In a generic clause, the head is solved only if the bottom of the body is solved. Notice that the universal quantifier $\forall tot[G]_1$ is necessary to guarantee standardization apart during the inference process.

Let us see an example.

EXAMPLE 3 *Consider the following program $\mathcal{P}$*

$$p(x, y) \leftarrow q(x) \otimes r(y)$$
$$q(a).$$
$$r(b).$$

The result of the transformation $[\![\mathcal{P}]\!]$ is the following:

$$\forall i_0 xy.(p(x, y, i_0) \circ\!\!- (\forall i_1 i_2.(\vartheta(i_0) \multimap \vartheta(i_2)) \multimap ((r(y, i_2) \multimap \vartheta(i_1)) \multimap q(x, i_1)))))$$
$$\forall i_0.(q(a, i_0) \circ\!\!- (\forall i_1.(\vartheta(i_0) \multimap \vartheta(i_1)) \multimap \vartheta(i_1))$$
$$\forall i_0.(r(b, i_0) \circ\!\!- (\forall i_1.(\vartheta(i_0) \multimap \vartheta(i_1)) \multimap \vartheta(i_1)))$$

□

Notice that the transformation can be optimized, in order to avoid the generation of unuseful. For example, the second and third clauses can be easily rewritten as follows:

$$\forall i_0.(q(a, i_0) \circ\!\!- \vartheta(i_0))$$

$$\forall i_0.(r(b, i_0) \circ\!\!- \vartheta(i_0))$$

Definition 4.5 *Consider a program $\mathcal{P}$ and a goal formula G. Then the transformation of the sequent $\mathcal{P}\ \mathcal{P}\ \gamma \longrightarrow (G, \omega)$ is $[\![\mathcal{P}]\!]; \Gamma, \top \multimap \vartheta(bot[G]_0) \vdash [\![G]\!]_0$, where $\Gamma = \{\forall x.\top \multimap p(t, x) \mid p(t) \in \gamma\}$.* □

The $\top \multimap \vartheta(bot[G]_0)$ formula in the consumable part of the sequent states that the proof terminates only when the last atom in the goal G is solved (and consequently its witness is made available). The underlying idea in the transformation is to obtain the sequentialization of the sequents by a sequentialization of the proof. Given a proof

$$\frac{\Xi_1 \quad \Xi_2}{\Xi}$$

we want to translate such a proof in the following

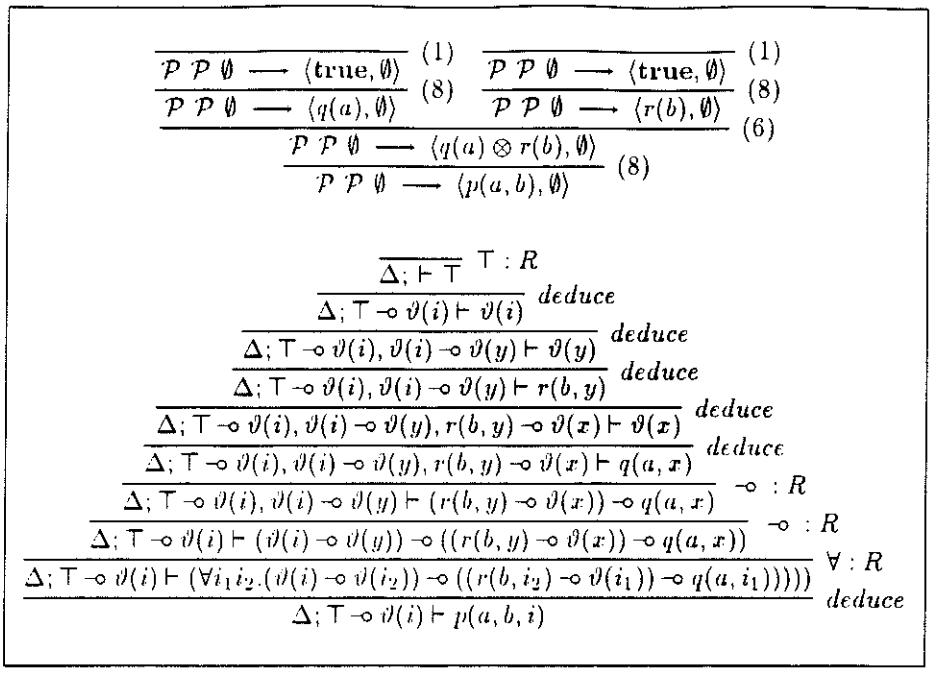

Figure 9: The proof-tree for Θ_1 and the corresponding proof-tree for Θ_2

$$\frac{[\![\Xi_2]\!]}{[\![\Xi_1]\!]}$$
$$[\![\Xi]\!]$$

That is to say, the sequent Ξ_2 must be evaluated only after the successful evaluation of the sequent Ξ_1.

We introduce the *deduce* inference rule, in order to abbreviate the instances of the following proof-trees:

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma; A[x/t] \vdash A[t/x]}\ initial \qquad \Gamma; \Delta \vdash G[x/t]}{\Gamma; A[x/t] \multimap G[x/t], \Delta \vdash A[t/x]}\ \multimap : L}{\Gamma; \forall x.(A \multimap G), \Delta \vdash A[t/x]}\ \forall : L}{\Gamma; \Delta \vdash A[t/x]}\ decide$$

Where $\forall x.(A \multimap G) \in \Gamma$. The subtree is simplified by omitting the $\forall : L$ inference rule if the clause is not universally quantified, or the *decide* inference rule if the clause r is chosen in the consumable part of the sequent. Notice how the *deduce* rule plays a fundamental role in the evaluation of the sequentiality, because a witness is made available and consequently a new formula can be evaluated.

EXAMPLE 4 *Consider the sequent* $\Theta_1 \equiv \mathcal{P}\ \mathcal{P}\ \emptyset \longrightarrow \langle p(a,b), \emptyset \rangle$ *and the corresponding translation in the fragment of linear logic* $\Theta_2 \equiv \Delta; \top \multimap \vartheta(i) \vdash p(a,b,i)$, *where* $\mathcal{P}$ *is the program of the previous example and* Δ *the transformed linear program. The proof-trees for* Θ_1 *and* Θ_2 *are shown in fig. 4.2. Notice that in the leaf of the proof-tree for* Θ_2 *the consumable part of the antecedent is empty, because no update is in* $\mathcal{P}$.

□

EXAMPLE 5 *Consider now the next $\mathcal{OL}$ program:*

$v(x,y) \leftarrow q(x) \otimes r(y).$
$q(a) \leftarrow add(p(a)).$
$r(a) \leftarrow p(a).$

Notice that the evaluation of $q(a)$ entails the modification of the context. □

Now we provide a transformation for the *stateupdate* metapredicates and for the state predicates.

Definition 4.6 *Consider $[\![G]\!]_k$ defined in def. 4.3. Let us extend the definition as follows:*

- $[\![A_j]\!]_k = (\forall i.\top \multimap p(t,i)) \multimap \vartheta(i_{\overline{k}})$, *when* $A_j \equiv add(p(t))$ *with* t *ground term;*
- $[\![A_j]\!]_k = p(t, i_{\overline{k}}) \otimes \vartheta(i_{\overline{k}})$ *when* $A_j \equiv del(p(t))$ *with* t *ground term;*
- $[\![A_j]\!]_k = p(t, i_{\overline{k}}) \,\&\, \vartheta(i_{\overline{k}})$, *when* $A_j \equiv p(t)$ *and* p *is a state predicate.*

□

The linear implication allows us to add new consumable formulas to the actual state. Such formulas can be consumed (and so deleted) by means of the $\otimes$ linear operator, because it splits the context in two subcontexts in which the consumable part is used to solve the $p(t)$ goal. Finally, in order to solve a state predicate we need to duplicate the context of evaluation.

EXAMPLE 6 *The transformation for the program of the example 5 is*

$$\forall x y i_0.(v(x,y,i_0) \circ\!\!- (\forall i_1 i_2.(\vartheta(i_0) \multimap \vartheta(i_2)) \multimap ((r(y,i_2) \multimap \vartheta(i_1)) \multimap q(x,i_1)))).$$
$$\forall i_0.(q(a,i_0) \circ\!\!- \forall i_1.((\vartheta(i_0) \multimap \vartheta(i_1)) \multimap ((\forall x_1.\top \multimap p(a,x_1)) \multimap \vartheta(i_1))))$$
$$\forall i_0.(r(a,i_0) \circ\!\!- \forall i_1.((\vartheta(i_0) \multimap \vartheta(i_1)) \multimap (p(a,i_1) \,\&\, \vartheta(i_1))))$$

□

EXAMPLE 7 *Consider the sequent $\Theta_1 \equiv \mathcal{P}\ \mathcal{P}\ \emptyset \longrightarrow \langle v(a,a), \{p(a)\}\rangle$ and the corresponding translation in the fragment of linear logic $\Theta_2 \equiv \Delta; \top \multimap \vartheta(i) \vdash v(a,a,i)$, where $\mathcal{P}$ is the original program and Δ the transformed linear program. The proof-trees for Θ_1 and Θ_2 are shown in fig. 4.2.* □

5 Conclusions

We have examined the role of state update in expressing integration between object-oriented programming and logic programming. As remarked, it is an important feature in proving the applicability of logic programming to interesting software-engineering problems [CL94].

We have recalled the importance of computation-as-proof interpretation in expressing dynamics in logic programming. The two approaches we have considered express a different philosophy of approaching the problem. It is clear, in fact, that a concurrent approach, though very elegant, does not allow to cover the problem of constructing large software systems, and so it has to be considered a "logic programming emulation" of object-orientedness. Moreover, a clausal approach allows covering such problems, and allows us to express a clean semantic interpretation of OOLP. It "only" remains to logically formalize the concept of assignment.

We have discussed the main properties of local state in OOLP: local state derives directly from the dynamics of the context of evaluation, and one of the dimensions of such dynamics is assignment, which can be formalized via computation-as-proof. The fundamental requirement consists in expressing inheritance via updatable metapredicates: the local state problem then reduces itself to the assignment problem. Notice also that data-encapsulation can be implicitly modeled by letting non-labelled predicates to be expressed in the program clauses: the non-labelled predicates can only be evaluated in the context of the object that received the message first.

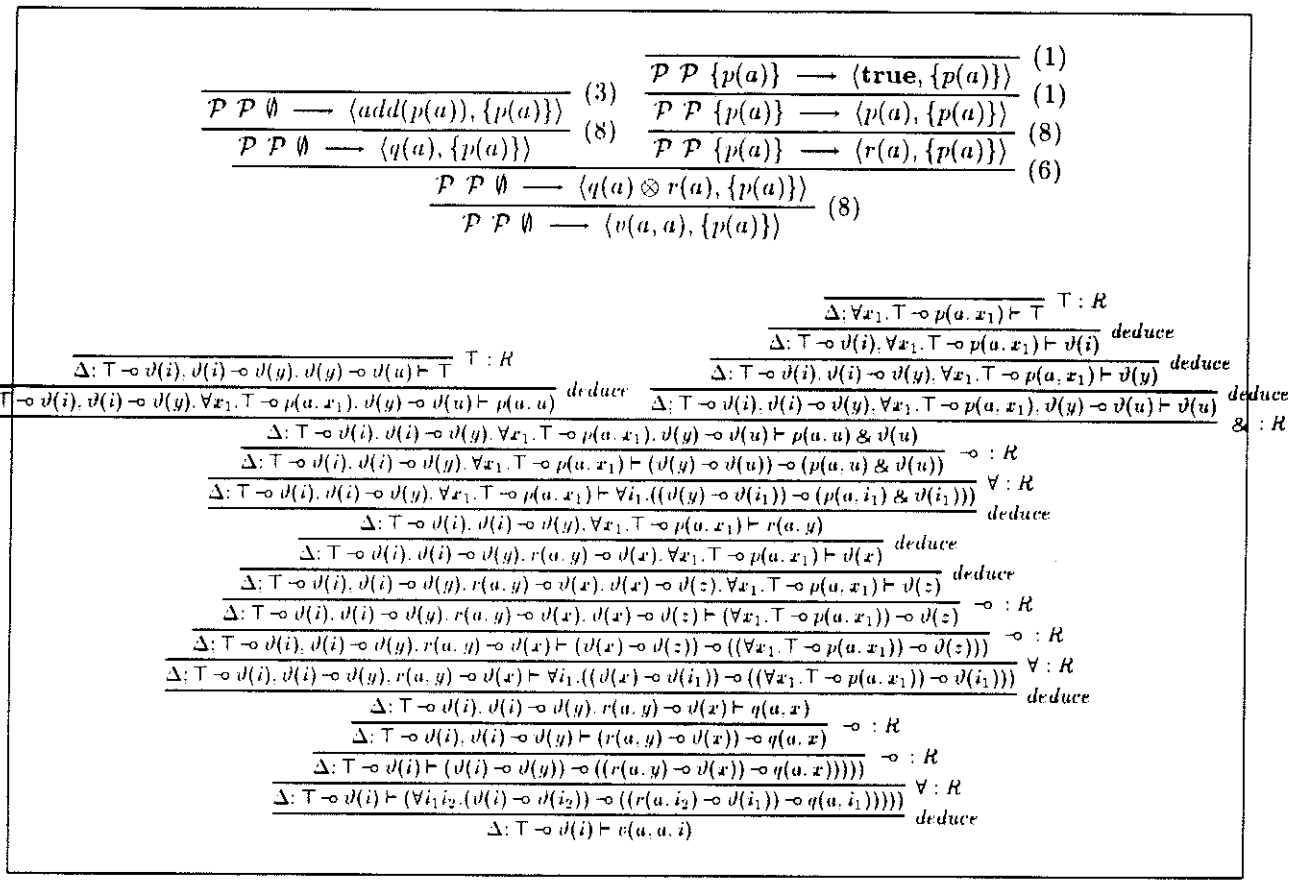

Figure 10: The proof-tree for Θ_1 and the corresponding proof-tree for Θ_2

The $\mathcal{OL}$ formalism is defined by looking explicitly at the described properties, thus allowing a formalization of object-oriented programming in a logic programming context. As a matter of fact, the very attractive point is that the expressive power of $\mathcal{OL}$ is much more than "simply" object-oriented: the way of representing inheritance, object-identity and assignment allows us to establish dynamic links between classes, or multiple links between an object and many classes. We can also establish conditional inheritance, by treating clauses expressing the **isa** hierarchy as definite clauses, or consider parameterized modules, by relaxing the condition that labels must have 0-arity. All these aspects formalize in a very elegant way the concept of role-dynamics in a logic programming framework. The logical foundation of the approach is shown by its definition via metalogics, and its mapping into linear logic features.

We have ignored a model-theoretic approach to the problem. This choice is due to many aspects we have looked at. We agree with [Kow90] in observing that the algebraic models we can give for the language are not enough to capture a declarative approach to programming. The declarative style of programming is influenced over all by the possibility of making computationally tractable and innovative the process of deduction of a fragment of logic we aim to implement. Clearly, looking at a model theoretic semantics does not help us too much in isolating the computational properties of such fragment.

References

[ACP92] J. M. Andreoli, P. Ciancarini, and R. Pareschi. Interaction Abstract Machines. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object Oriented Programming*. 1992. To appear.

[AK91] Serge Abiteboul and Paris C. Kanellakis. On the Formalization of Object-Oriented Databases Models. In *Proceedings of the ICLP'91 Workshop on Merging Object-Oriented and Logic Programming*, 1991. Position Paper.

[AKN86] H. Aït Kaci and R. Nasr. LOGIN: A Logic Programming Language with Built-in Inheritance. *Journal of Logic Programming*, 3(3):185–215, 1986.

[AKP93] H. Aït Kaci and A. Podelski. Towards a Meaning of LIFE. *Journal of Logic Programming*, 16(3):195–234, 1993.

[Ale93] V. Alexiev. Mutable State for Object-Oriented Logic Programming: A Survey. Technical Report 93–15, Department of Computer Science, University of Alberta, 1993.

[And91] J. M. Andreoli. For a Logic of Action. In *Proceedings of the ICLP'91 Workshop on Merging Object-Oriented and Logic Programming*, 1991. Position Paper.

[And92] J. M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation*, 2(4):297–347, 1992.

[AP91a] J. M. Andreoli and R. Pareschi. Communication as Fair Distribution of Knowledge. In *proceedings of the International Conference on Object-Oriented Programming Systems, Languages and Applications (OOPLSA'91)*, pages 212–229, 1991.

[AP91b] J. M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-in Inheritance. *New Generation Computing*, (3):359–483, 1991.

[Ber91] H. Berse. Integrating Object-Oriented Programming and Logic Programming. In *Proceedings of the ICLP'91 Workshop on Merging Object-Oriented and Logic Programming*, 1991. Position Paper.

[BG95] P. Bruscoli and A. Guglielmi. A Linear Logic Programming Language with Sequential and Parallel Conjunction. In *Proceedings of the GULP-PRODE'95 Joint Conference on Declarative Programming*, 1995. to appear.

[BGM94] M. Baldano, L. Giordano, and A. Martelli. A Modal Extension of Logic Programming. In Maria Alpuente, Roberto Barbuti, and Isidro Ramos, editors, *Proceedings of the GULP-PRODE'94 Joint Conference on Declarative Programming*, volume 2, pages 324–335, September 1994.

[BGM95] Elisa Bertino, Giovanna Guerrini, and Danilo Montesi. Deductive Object Databases. In *Proocedings of ECOOP'95*, 1995.

[BJ94] M. Bugliesi and H. M. Jamil. A Logic For Encapsulation in Obiect-Oriented Languages. In Maria Alpuente, Roberto Barbuti, and Isidro Ramos, editors, *Proceedings of the GULP-PRODE'94 Joint Conference on Declarative Programming*, volume 2, pages 161–175, September 1994.

[BK82] K. A. Bowen and R. A. Kowalski. Amalgamating Language and Metalanguage in Logic programming. In K. L. Clark and S. A. Tarnlund, editors, *Logic Programming*. Academic Press, 1982.

[BK93] A. J. Bonner and M. Kifer. Transaction Logic Programming. Technical Report CSRI-270, Computer System Research Institute, University of Toronto, December 1993.

[BLM94] Michele Bugliesi, Evelina Lamma, and Paola Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19,20:443–502, 1994.

[BMPT94] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 1994.

[Bro93] A. Brogi. *Program Construction in Computational Logic*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 1993.

[BRT95] A. Brogi, C. Renso, and F. Turini. Amalgamating Language and Metalanguage for Composing Logic Programs. In M. Alpuente, R. Barbuti, and I. Ramos, editors, *Proceedings of the GULP-PRODE'94 Joint Conference on Declarative Programming*, volume 2, 1995.

[BT93] A. Brogi and F. Turini. Metalogic for State Oriented Programming. In E. Lamma and P. Mello, editors, *Estensions of Logic Programming*, volume 660 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, Berlin, 1993.

[Cas91] Y. Caseau. Mixing Objects and Logic: An Experimental Point of Wiew. In *Proceedings of the ICLP'91 Workshop on Merging Object-Oriented and Logic Programming*, 1991. Position Paper.

[CL94] P. Ciancarini and G. Levi. What is Logic Programming Good for in Software Engineering. Technical report, Department of Computer Science University of Bologna, 1994.

[Dav91] A. Davison. From Parlog to Polka in Two Easy Tteps. In J. Maluszynsky and M. Wirsing, editors, *Third International Symposium on Programming Language Implementation and Logic Programming*, number 528 in Lecture Notes in Computer Science, pages 255–274, 1991.

[Far86] L. Farinas Del Cerro. Molog: A System that Extends Prolog with Modal Logic. *New Generation Computing*, (4):35–50, 1986.

[Fos91] I. Foster. A Declarative State Transition System. *Journal of Logic Programming*, pages 45–67, 1991.

[Gir87] J. Y. Girard. Linear Logic. *Theoretical Computer Science*, 50, 1987.

[GMR92] Laura Giordano, Alberto Martelli, and Gianfranco Rossi. Extending Horn Clause Logic with Implications Goals. *Theoretical Computer Science*, 95(1):43–74, March 1992.

[MZ86] D. McAllister and R. Zabih. Boolean Classes. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Application*, pages 417–423, 1986.

[Sce94] A. Scedrov. Linear Logic and Computation: A Survey. In H. Schwichtenberg, editor, *Proof and Computation*. Springer Verlag, 1994. To appear.

[SS86] L. Sterling and E. Shapiro. *The Art of Prolog*. The MIT Press, 1986.

[ST87] E. Shapiro and A. Takeuchi. Object-Oriented Programming in Concurrent Prolog. In E. Shapiro, editor, *Concurrent Prolog*, volume 2, pages 251–273. The MIT Press, 1987.

[Uus92] T. Uustalu. Combining Object-Oriented and Logic Paradigms: A Modal Logic Programming Approach. In *Proceedings of the Europan Conference on Object-Oriented Programming (ECOOP'92)*, number 615 in Lecture Notes in Computer Science, pages 98–113, 1992.