Math. Struct. in Comp. Science (1999), *vol.* ??, *pp.* 1–??. Printed in the United Kingdom

A Specification Logic for Concurrent Object-Oriented Programming

G. Delzanno[1], D. Galmiche[2] and M. Martelli[1]

[1] *Dipartimento di Informatica e Scienze dell'Informazione,*
Università di Genova,
Via Dodecaneso, 35, I-16146 Genova, Italy,
email:{giorgio,martelli}@disi.unige.it

[2] *LORIA UMR 7503 - UHP Nancy 1,*
Campus Scientifique - B.P. 239
54506 Vandœuvre-lès-Nancy Cedex, France.
e-mail: galmiche@loria.fr

Received 20 November 1998

The paper is focused on the use of linear logic for the specification of the operational semantics of advanced concepts of programming, and, in particular, of concurrent object-oriented programming. For this purpose, we shall employ a refined sequent-calculus based on the proof-theoretical characterization of linear logic programming. A synergic combination of higher-order logic programming and linear logic will be used to give an accurate encoding of the traditional features of concurrent object-oriented programming languages, whose corner-stone is the notion of *encapsulation.*

1. Introduction

The paper is focused on the use of linear logic as a language to specify advanced concepts of programming, and, in particular, to specify the salient aspects of concurrent object-oriented programming.

For these purposes, it is important to have in mind the refinement process which, from the original logic defined by Girard in (Girard 1987), leads to *executable linear logic specification languages.* In the paper we shall give a general methology for developing (sub)logics suitable to express the knowledge we want to represent. The operational aspects of linear logic will be emphasized by means of a step-wise *refinement* of the original Girard's proof calculus (Girard 1987). From a general point of view, such a refinement is based on specific interpretations of formulae, sequents and proofs. From a technical point of view, it consists of a proof theoretic analysis of the underlying logic aimed at the definition of classes of proofs which correspond to the considered operational view. In a sense, these are the ideas behind the design of traditional logic programmimg languages (Miller *et al.* 1991).

The refinement process will be presented through a classification of the different approaches in the literature. Such a classification is based on the following points: the set of connectives used as *primitive constructs* of the specification language (e.g. the different fragments used in ACL (Kobayashi and Yonezawa 1993), LO (Andreoli and Pareschi 1991), Lolli (Hodas 1994), etc); the level of non-determinism of the proof search procedure (e.g. canonical proofs (Galmiche and Perrier 1994b), uniform proofs (Hodas 1994), etc.); the form of sequents (e.g. single and multi-conclusion); In the course of the paper we shall explain which considerations led to the choice of the fragments studied in the literature.

To achieve our main goal, i.e., to study concurrent objects in the linear logic setting, we commit ourselves to a particular interpretation of sequents and proofs, i.e., *uniform proofs as computations*, as proposed by Miller in (Miller 1996) with the language Forum. Forum is a presentation of full higher-order linear logic. This choice is guided by previous work on extensions of logic programming, in which the logical connectives are given a precise operational interpretation in terms of *search directives*, see e.g. (Miller 1989a; Miller 1989b; Andreoli and Pareschi 1991; Hodas and Miller 1994; Miller 1996). In a sense, Forum (Miller 1996) can be viewed as an intermediate refinement step to achieve a *readable* form of linear logic specifications.

Our final refinement step is to consider a specific sublanguage, namely $\mathcal{E}_{hhf}$(Delzanno 1997; Delzanno and Martelli 1998), by which we emphasize the view of proofs as *state-based* computations (essential in the object-oriented paradigm). The considered restriction on the form of linear logic formulae allows us to define executable linear logic programs with a semantics dictated by the specialized proof system of $\mathcal{E}_{hhf}$. This restriction represents a good compromise between the expressiveness of the logic and the efficiency (and readability) of the proof-search process. Furthermore, it provides all the features needed to define logical foundations for a framework involving object-oriented, concurrent and logic programming, following the lines drawn in our previous works (Delzanno and Martelli 1995; Boudinet and Galmiche 96). We shall illustrate all of these points, by presenting an accurate encoding of the most common features of concurrent object-oriented programming. More specifically, we shall focus on *encapsulation*, *method invocation*, *inheritance* and *overriding* at the object-level. Concurrency at the execution level will be modeled in a natural way by assigning an *interleaving* semantics to the execution of methods.

As it will be clear from the discussion in the first part of the paper, other fragments of linear logic can be applied in this context. For instance, thanks to the deep symmetry of linear logic it is possible to figure out *dual* encodings in which sequents, and proofs assume different operational interpretations, see e.g., the approach based on multi-conclusion intuitionistic linear logic in (Boudinet and Galmiche 96). To our opinion, such flexibility is one of the main reasons to further inspect the potentiality of linear logic and, more in general, of proof-theoretical approaches for the specification of the semantics of programming languages.

2. Linear Logic and Aspects of Programming

Linear logic (in the following abbreviated as LL) is a powerful and expressive logic which presents connections with a variety of topics in computer science (Alexiev 1994). From a proof-theoretical point of view, LL derives from Classical Logic (CL) by eliminating the structural rules of *weakening* and *contraction*. The result is a logic in which, intuitively, it is possible to treat formulae as *resources*. Contraction and weakening are re-introduced in a restricted way, i.e., they can be applied only to the subclass of formulae prefixed by the two modalities ! and ?.

The elimination of the structural rules has an important consequence in the formulation of the logical rules: the additive and the multiplicative formulations of the system are not equivalent anymore. As a consequence, each CL connective is split into two LL connectives (an additive and a multiplicative version); for instance, $\wedge$ is split into & and $\otimes$; $\vee$ is split into $\oplus$ and $\parr$; $\supset$ into $\leadsto$ and $\multimap$; *true* into $\top$ and $\mathbf{1}$; $false$ into $\mathbf{0}$ and $\bot$. Thanks to its simmetry, the inference systems of LL can be given with two equivalent formulations: one with one-sided sequents $\vdash \Delta$, the other with two-sided sequents $\Gamma \vdash \Delta$ (see Appendix A). Here Δ and Γ are *multisets* of formulas.

Intuitionistic linear logic (ILL) (Schellinx 1991) is usually defined by restricting the right-hand side of sequents to an empty or a singleton multiset. However, similarly to the classical case, there exist multi-conclusion formulations of ILL with special restriction on some of the rules, e.g., full intuitionistic linear logic (FILL) (Hyland and de Paiva 1993).

LL is often referred to as a logic for *concurrency* (see, e.g., (Meseguer 1991; Abramsky 1993)). The reason for this can be illustrated by considering the *process-view* of (Kobayashi and Yonezawa 1993). Their approach is based on an interpretation where formulae are viewed as processes and connectives as algebraic operations on processes: $\parr$ represents the parallel composition, whereas $^{\bot}$ (in combination with $\otimes$) implements message passing. The proof-rules of the Girard's sequent calculus assign a natural operational semantics to this interpretation. As an example, let us consider the following formula: $m \parr (m^{\bot} \otimes P_1) \parr P_2$. Here, m is viewed as a process that disappears after having sent the message m, and $(m^{\bot} \otimes P_1)$ as a process waiting for the message m and running in parallel with P_2. Based on the rules shown in Appendix A, the sequent $\vdash m \parr (m^{\bot} \otimes P_1) \parr P_2$ can be simplified via the $\parr_R$ rule into $\vdash m, (m^{\bot} \otimes P_1), P_2$ and, then, via the $\otimes_R$ rule, into $\vdash P_1, P_2$ (note that $\vdash m, m^{\bot}$ is an axiom). Thus, such (partial) proof can be interpreted in terms of process reduction.

With the same view in mind, let us now adopt a fragment that does not include the connective $\otimes$, but that includes the connective $\multimap$. By the duality of the linear connectives, the receiver can be rewritten as $P_1 \multimap m$. However, we need a two-sided sequent formulation, namely $P_1 \multimap m \vdash m, P_2$, to have the same operational behaviour as before.

Through the example, we see the importance of the interaction between the logical fragment, the use of specific formulae and sequents to specify concepts or mechanisms and the proof-search process. Every choice represents here a compromise between the expressiveness of the fragment and the efficiency of the corresponding proof-search process. In the following section we shall give a brief overview of the different components involved in this process.

3. Proof-theoretic analysis

The formalization of a logic based on sequent calculus provides for a better comprehension of the *operational* aspects of proof construction. In the meantime, sequent calculi provide a powerful organization of the knowledge specified through the formulas of the considered logic language. Let us give some ideas on these two aspects in the case of LL.

3.1. *Sequents and proofs*

Though the basic meaning of sequents is to express theorems of the considered logic, they can be given other interpretations. For instance, under the previously mentioned *formulae-as-processes* view of (Kobayashi and Yonezawa 1993), one-sided sequents can be used to specify configurations of processes. Differently, in (Lincoln and Saraswat 1993; Perrier 1995) a sequent is viewed as a reduction of processes (i.e. $P \vdash P'$ is interpreted as P reduces to P'). This can be achieved in a single-conclusion setting as in in fragments of ILL (Perrier 1995) or in a multi-conclusion setting as in fragments of FILL (Boudinet and Galmiche 96).

In the context of logic programming, one usually considers sequents of the form $P \vdash G$ where the set of formulae P represents the program and the formula G represents the goal to satisfy. In the linear logic programming setting, it comes naturally to further refine the sequent syntax in order to couple with the notion of *bounded-use* formulas. For example, as we will explain later, Forum sequents (Miller 1994) have the form $\Sigma : \Gamma_1; \Gamma_2 \vdash \Delta; \Upsilon$ where Σ is a signature, and Γ_1, Γ_1 are, respectively, the intuitionistic (i.e. re-usable) and linear (i.e. bounded-use) context. Such refinements are motivated by the need of providing simple and readable *judgements* that take into account the considered operational interpretation.

We shall see that the choice between single-conclusion sequents, for instance in intuitionistic linear logic (ILL) (Hodas and Miller 1994), and multiple-conclusion sequents, for instance in LL (Miller 1994) and in FILL (Hyland and de Paiva 1993), has consequences both at the specification and at the proof-search level.

In a sense, the operational interpretation of sequents makes the difference between theorem proving and logic programming. Both theorem proving and logic programming can be viewed as the process of constructing the proof of a sequent from the bottom up (i.e. from the sequent to prove). As well explained in (Hodas and Polakow 1996) the line between theorem provers and logic programming systems is drawn by what sort of proof procedure is used in proving goal formulae Δ from Γ. It is clear that, for theorem proving, any "reasonable" procedure is a good choice. On the other hand, in logic programming, the programmer must be able to clearly understand the behaviour of a program. Thus, the proof procedure has to be simple, predictable and *goal-directed* (Hodas and Polakow 1996). In other words, it must reflect the operational interpretation assigned to formulas and sequents.

3.2. *Proof-search in linear logic*

There are many works devoted to proof-search in linear logic and some of its sub-fragments. All of these proposals are based on the non-permutability results of the LL inference rules in the corresponding sequent calculus. Example are given by the proof search strategies defined in (Galmiche and Perrier 1994a; Lincoln and Shankar 1994; Tammet 1994), and by the classes of proofs defined in (Andreoli 92; Galmiche and Perrier 1994b; Hodas 1994; Pym and Harland 1994), that are complete w.r.t. the provability.

Let us analyse how we can naturally design and justify such strategies and proofs for a given logical fragment. For this purpose, we recall a general, two-steps, method applied to LL and to other fragments in (Galmiche and Perrier 1994a; Galmiche and Perrier 1994b). The first step consists of the study of the permutability of the inference rules and of the analysis of the possibilities of inference movements (by permutability) in a proof. This can be done sistematically by an exhaustive case analysis. After fixing a given strategy (e.g., bottom-up or top-down), the second step consists of defining a notion of (cut-free) *normal* proof, which reflects the considered strategy, i.e., in which we impose some order in the application of the rules. The degree of non-determinism left in this phase depends on the operational interpretation assigned to the proofs (i.e. it can reflect the non-determinism of the object-level). Further constraints on the proof search (e.g. we restrict ourselves to goal-directed proofs) can lead to different classes of normals proofs. Normal proofs are significant just in case they can be proved complete with respect to the full class of proofs of the considered fragment. As an example, let us consider the design of *canonical proofs* in LL as defined in (Galmiche and Perrier 1994b). Starting from the one-sided sequent calculus, we study the inference permutability and the possible movements of inferences in a proof. After fixing a bottom-up search strategy, we refine the previous results and determine the set of inferences to move up and down in a proof, respectively, $I_{\uparrow} = \{\otimes, \oplus_1, \oplus_2, c?, w?, ?, \exists\}$ and $I_{\downarrow} = \{⅋, \&, \forall, \perp\}$.

Isn't it too much condensed?

Canonical proofs (Galmiche and Perrier 1994b) result from imposing an order in the application of the rules, according to the two previously defined sets:

A *canonical proof* in (full) LL, for bottom-up proof-search, is a proof without *cut*s, without conflicts between $w?$ and $c?$ rules where, for any intermediate conclusion, we first apply the $c?$ rule, then the ! rule, then a rule of $I_{\downarrow}$ and, finally, a rule of $I_{\uparrow}$.

What does it mean *without conflicts*?

As shown in (Galmiche and Perrier 1994b), canonical proofs are complete wrt. provability in LL, i.e., if a sequent s is provable then it has a canonical proof (Galmiche and Perrier 1994b). This definition well illustrates the generality of the above *methodology* for a given logic.

I'd say methodology instead of definition. The definition is given for the case of (full) LL

On the way to a more concrete framework, we commit ourselves to a special interpretation of sequents and on special proof search strategies. For this reason, in the following section we shall restrict ourselves to the so-called *proofs as computations* interpretation in the context of linear logic programming.

3.3. *Proof-search as computation*

As already mentioned in the previous section, after a preliminary proof-theoretic analysis, it is important to fix a strategy in accord with the operational interpretation assigned to sequents and proofs. In particular, goal-directed proof-search is the strategy at the basis of logic programming. This strategy is reflected in the notion of *uniform provability*, originally introduced in the case of intuitionistic (classical) logic in (Miller *et al.* 1991). Uniform proofs are suitabe to assign a clear operational meaning to logical connectives in terms of search control directives.

Goal-directed proof construction strongly depends on the form of sequents considered in the calculus, i.e., single or multi-conclusion calculi.

3.3.1. *Single-conclusion sequents.* As customary in proof-theoretical characterization of logic programming, a sequent $P \vdash G$ is viewed as the instantaneous configuration of an ideal interpreter, where the formulae of P represent the current program and the formulae G represents the current goal to be establish. Goal-directed provability is formalized by means of the notion of uniform proof (Miller *et al.* 1991):

A *uniform proof* is a cut-free proof in which every occurrence of a sequent whose right-hand side is non-atomic is the conclusion of a right-introduction rule.

Thus, in this setting left-rules can be applied only after the right-hand side of a sequent has been reduced to an atomic formula. This definition has been extended to the case of linear logic in for particular subclasses of linear logic formulas (Hodas and Miller 1994; ?). The resulting languages, e.g. Lolli (Hodas and Miller 1994) and Lygon (?), provide new *programming constructs* with respect to traditional extensions of logic programming based on intuitionistic logic such as λProlog.

To complete the overview of the methodology used to define special purpose sublogics, we will briefly analyze the logic introduced in (Hodas and Miller 1994), restricting ourselves to goal-directed provability (a further parameter that must be considered in the definition of *normal* proof).

In this setting formulas are defined according to the following grammar:

$$\begin{array}{lcl} \mathcal{D} & ::= & \top \mid A \mid \mathcal{D}\&\mathcal{D} \mid \mathcal{G} \multimap A \mid \forall x.\mathcal{D}. \\ \mathcal{G} & ::= & \top \mid A \mid \mathcal{G}\&\mathcal{G} \mid \mathcal{G} \otimes \mathcal{G} \mid \mathcal{G} \oplus \mathcal{G} \mid \mathcal{D} \multimap \mathcal{G} \mid !\mathcal{D} \multimap \mathcal{G} \mid !\mathcal{G} \mid \exists x.\mathcal{G} \mid \forall x.\mathcal{G}. \end{array}$$

The sequents have the specific form $!\Gamma, \Delta \vdash G$ where Γ is a set of $\mathcal{D}$-formulas, Δ is a multiset of $\mathcal{D}$-formuals and $\mathcal{G}$ is a goal.

By using the sub-formula property, we can restrict ouselves to consider the following logical rules:

$$\mathcal{R} = \{\multimap_L, id, \&_L, !_L, c!_L, w!_L, \forall_L, \top_R, \otimes_R, \multimap_R, \&_R, \oplus_R, \forall_R, !_R, \exists_R\}$$

We study now the permutability properties of the $\mathcal{R}$ rules (see Fig. 10 Appendix Appendix A). The analysis of the possible movements of inference rules leads to the following considerations. First of all, being in a single-conclusion setting, two right rules are never in permutation position. Moreover, since a resource cannot have the form $!D$,

$$\frac{}{A, !\Gamma \vdash A}\ id' \qquad \frac{\Gamma_1, !\Gamma \vdash G_1 \quad \Gamma_2, !\Gamma \vdash \Gamma_2}{\Gamma_1, \Gamma_2, !\Gamma \vdash G_1 \otimes G_2}\ \otimes'_R \qquad \frac{\Gamma \vdash G}{G \multimap A, \Gamma \vdash A}\ \multimap'_L$$

$$\frac{R, \Gamma \vdash G}{\Gamma \vdash R \multimap G}\ \multimap_R \qquad \frac{A, \Gamma \vdash G}{A\&B, \Gamma \vdash G}\ \&_L \qquad \frac{B, \Gamma \vdash G}{A\&B, \Gamma \vdash G}\ \&_L$$

$$\frac{\Gamma \vdash G_1 \quad \Gamma \vdash G_2}{\Gamma \vdash G_1 \& G_2}\ \&_R \qquad \frac{\Gamma \vdash G_1}{\Gamma \vdash G_1 \oplus G_2}\ \oplus_R \qquad \frac{\Gamma \vdash G_2}{\Gamma \vdash G_1 \oplus G_2}\ \oplus_R$$

$$\frac{}{\Gamma \multimap A, \Gamma_1, \Gamma_2 \vdash \top}\ \top'_R \qquad \frac{A, !A, \Gamma \vdash G}{!A, \Gamma \vdash G}\ !'_L \qquad \frac{!\Gamma \vdash A}{!\Gamma \vdash !A}\ !_R$$

$$\frac{A[t/x], \Gamma \vdash G}{\forall x A, \Gamma \vdash G}\ \forall_L \qquad \frac{\Gamma \vdash G}{\Gamma \vdash \forall x G}\ \forall_R \qquad \frac{\Gamma \vdash A[t/x]}{\Gamma \vdash \exists x A}\ \exists_R$$

Fig. 1. The uniform proof system for Lolli (Hodas and Miller 1994).

then the inference of type $!_R$ is here always permutable with inference of type $\multimap_L, \&_L$ and $\forall_L$. Thus, we end up in the following set of inferences that can be moved down $I_\downarrow = \{\multimap_R, \&_R, \forall_R, \exists_R, \oplus_R, c!_L\}$, or that can be moved up $I_\uparrow = \{\otimes_R, \oplus_R, \exists_R, \multimap_L, \&_L, !_L, w!_L, \forall_L\}$. Let us remark that the inferences of $!_R$ type can be moved down by jumps but not because of permutability results. Note that, if we want to fix a particular order in the application of the rules, we still have different choices (e.g. for $\oplus_R$). However, except the $c!_L$ and $\otimes_R$ inferences, it appears that the right (resp. left) rules can easily be moved down (resp. up). Having fixed such specific movements it is now possible to move down the $\otimes_R$ inferences without any problem, because of the permutability with the left rules. Such an order correspond to the idea of uniform provability defined at the beginning of this section. In (Hodas and Miller 1994), it is proved that uniform provability (in the considered fragment) is complete with respect to provability in ILL. Let us mention that the permutability results are also used to define a variant of the initial system, in which all proofs are necessarily uniform (see Fig. 1). The new sequent calculus incorporates the considered operational interpretation of the proofs. The reader may refer to (**?**), for an accurate analysis of the maximal fragments of LL complete with respect to uniform provability.

As remarked in (Hodas and Miller 1994), the design of Lolli was guided by operational considerations, i.e., to extend hereditary harrop formulas with resource management (i.e. with the use of $\multimap$ and $\otimes$ in goals), while preserving uniform provability. However, as remarked in the Section **??**, multi-conclusion calculi provide for a natural view of concurrency at the logical level. We will discuss this point in the following section.

3.3.2. *Multi-conclusion sequents.* According to the logic programming interpretation of sequents and proofs, being in a multi-conclusion setting provides a new dimension that increases the expressiveness of traditional logic programming languages. Let us consider a two-sided sequent $\Gamma \vdash \Delta$. In this setting the formulae of Γ represent the current program, whereas the formulae of Δ represent a *multiset* of goals to be proved. Thus, in the extended setting we can encode the process-view of Section 2, as suggested in (Andreoli and Pareschi 1991; Miller 1993).

Being in a multi-conclusion setting leads to an extended notion of goal-directed provability (Andreoli 92; Miller 1994). The problem encountered by such an extension is well explained in (Miller 1994), presenting the higher order specification logic Forum.

Forum (Miller 1994; Miller 1996) is based on a simply typed lambda calculus presentation of LL. The connectives are constants with functional type, e.g., $\& : o \rightarrow o \rightarrow o$, with o representing the type of formulae. Higher-order quantification is defined through a family of constants $\forall_\tau$ having type $(\tau \rightarrow o) \rightarrow o$ for each type τ. A quantified formulas is then expressed by the expression $\forall_\tau(\lambda x.F)$. These aspects can be hidden when considering a proof-theoretic presentation of the logic. However, it is important to recall that λterms in normal form can occur inside Forum sequents and also as substitution terms in the $\forall_r$ rule.
The language is based on a fragment of LL with the following set of connectives: $\multimap$, $\Rightarrow$, $\&$, $\invamp$, $\perp$, $\top$, ? and the universal quantification. For the sake of simplicity, we shall omit the type in the quantification if it is clear from the context. The intuitionistic implication $A \Rightarrow B$ is defined here as $(!A \multimap B)$. In the following we shall also use $A \multimapinv B$ and $A \Leftarrow B$ as an alternative notation for $B \multimap A$ and $B \Rightarrow A$, respectively. Using the logical equivalences in Appendix A it is easy to see that by proper combinations of the considered set of connectives it is possible to express all the remaining ones.

Forum sequents are the multi-conclusion version of Lolli-ones. They have the following form (Miller 1994): $\Sigma : \Gamma; \Delta \vdash \mathcal{A}, F, \Omega; \Upsilon$, where, Σ is a signature containing all the constants appearing in the sequent. The two sides are divided by ";" in two parts in order to distinguish between the re-usable formulae and bounded-use formulas. In fact, the formulas in the multiset Γ are implicitly prefixed by "!", whereas, the formulas in the multiset Υ are implicitly prefixed by "?". Γ is a multiset of formulas. Finally, the bounded context on the right consists of a *list* of atomic formulae $\mathcal{A}$, a compound formula F and a multi-set of formulae Ω. Intuitively, $\mathcal{A}$ is the result of the simplification of the connectives in the right-hand side of sequents. It is possible to consider an equivalent formulation in which the right hand side is a *multi-set* of formulae, as we shall do in the rest of the paper. The LL interpretation of such sequents is the following: $!(\&_i C_i) \otimes (\otimes_i D_i) \rightarrow (\invamp_i A_i) \invamp F \invamp (\invamp_i G_i) \invamp (\oplus_i H_i)$, where $C_i \in \Gamma$, $D_i \in \Delta$, $A_i \in \mathcal{A}$, $G_i \in \Omega$, $H_i \in \Upsilon$. The methodology described in the previous section can also be applied here to analyze permutability of the rules for the considered connectives. Thus, a specialized proof system, see Figure 2, can be defined by considering uniformity of proofs as main requirement, which, in this setting, can be reformulated as follows:

A cut-free proof δ is *uniform* if for every subproof γ of δ and for every non-atomic occurrence B in the right-hand side of the end-sequent of δ, there is a proof ϵ that is equal to γ up to a permutation of inference rules and is such that the last inference rule in ϵ introduces the top-level logical connective of B.

According to this definition of uniformity, Forum can be considered an Abstract Logic Programming Language (Miller *et al.* 1991). Some examples of applications of Forum as high-level specification language for different programming paradigms can be found in (Chirimar 1995; Delzanno and Martelli 1995; Guglielmi 95; Hodas and Polakow 1996). As remarked in (Hodas and Polakow 1996), further refinement seem necessary to consider Forum as a logic programming language. The main problem of this approach is that

$$\frac{\Gamma;\Delta \xrightarrow{B} \mathcal{A};\Upsilon}{\Gamma;\Delta,B \longrightarrow \mathcal{A};\Upsilon}\ (decide_{\Delta}) \qquad \frac{\Gamma,B;\Delta \xrightarrow{B} \mathcal{A};\Upsilon}{\Gamma,B;\Delta \longrightarrow \mathcal{A};\Upsilon}\ (decide_{\Gamma}) \qquad \frac{\Gamma;\Delta \longrightarrow \mathcal{A},B;B,\Upsilon}{\Gamma;\Delta \longrightarrow \mathcal{A};B,\Upsilon}\ (decide?)$$

$$\frac{}{\Gamma;\emptyset \xrightarrow{A} A;\Upsilon}\ (initial_1) \qquad \frac{}{\Gamma;\Delta \longrightarrow \mathcal{A},\top,\Omega;\Upsilon}\ (\top_r) \qquad \frac{}{\Gamma;\emptyset \xrightarrow{A} \emptyset;A,\Upsilon}\ (initial_2)$$

$$\frac{}{\Gamma;\emptyset \xrightarrow{\bot} \emptyset;\Upsilon}\ (\bot_l) \qquad \frac{\Gamma;\Delta \longrightarrow \mathcal{A},\Omega;\Upsilon}{\Gamma;\Delta \longrightarrow \mathcal{A},\bot,\Omega;\Upsilon}\ (\bot_r)$$

$$\frac{\Gamma;B \longrightarrow \emptyset;\Upsilon}{\Gamma;\emptyset \xrightarrow{?B} \emptyset;\Upsilon}\ (?_l) \qquad \frac{\Gamma;\Delta \longrightarrow \mathcal{A},\Omega;B,\Upsilon}{\Gamma;\Delta \longrightarrow \mathcal{A},?B,\Omega;\Upsilon}\ (?_r)$$

$$\frac{\Gamma;\Delta \xrightarrow{B_i} \Omega;\Upsilon \quad i \in \{1,2\}}{\Gamma;\Delta \xrightarrow{B_1\&B_2} \mathcal{A};\Upsilon}\ (\&_l) \qquad \frac{\Gamma;\Delta \longrightarrow \mathcal{A},B,\Omega;\Upsilon \quad \Gamma;\Delta \longrightarrow \mathcal{A},C,\Omega;\Upsilon}{\Gamma;\Delta \longrightarrow \mathcal{A},B\&C,\Omega;\Upsilon}\ (\&_r)$$

$$\frac{\Gamma;\Delta_1 \xrightarrow{B} \mathcal{A}_1;\Upsilon \quad \Gamma;\Delta_2 \xrightarrow{C} \mathcal{A}_2;\Upsilon}{\Gamma;\Delta_1,\Delta_2 \xrightarrow{B \parr C} \mathcal{A}_1+\mathcal{A}_2;\Upsilon}\ (\parr_l) \qquad \frac{\Gamma;\Delta \longrightarrow \mathcal{A},B,C,\Omega;\Upsilon}{\Gamma;\Delta \longrightarrow \mathcal{A},B \parr C,\Omega;\Upsilon}\ (\parr_r)$$

$$\frac{\Gamma;\Delta_1 \xrightarrow{B} \mathcal{A}_1;\Upsilon \quad \Gamma;\Delta_2 \xrightarrow{C} \mathcal{A}_2;\Upsilon}{\Gamma;\Delta_1,\Delta_2 \xrightarrow{B \multimap C} \mathcal{A}_1+\mathcal{A}_2;\Upsilon}\ (\multimap_l) \qquad \frac{\Gamma;\Delta,B \longrightarrow \mathcal{A},C,\Omega;\Upsilon}{\Gamma;\Delta \longrightarrow \mathcal{A},B \multimap C,\Omega;\Upsilon}\ (\multimap)$$

$$\frac{\Gamma;\emptyset \longrightarrow B \quad \Gamma;\Delta \xrightarrow{C} \mathcal{A};\Upsilon}{\Gamma;\Delta \xrightarrow{B\Rightarrow C} \mathcal{A};\Upsilon}\ (\Rightarrow_l) \qquad \frac{\Gamma,B;\Delta \longrightarrow \mathcal{A},C,\Omega;\Upsilon}{\Gamma;\Delta \longrightarrow \mathcal{A},B \Rightarrow C,\Omega;\Upsilon}\ (\Rightarrow)$$

$$\frac{t{:}\tau \text{ is a } \Sigma\text{-term} \quad \Gamma;\Delta \xrightarrow{B[t/x]} \mathcal{A};\Upsilon}{\Gamma;\Delta \xrightarrow{\forall_\tau x:\tau.B} \mathcal{A};\Upsilon}\ (\forall_l) \qquad \frac{\Gamma;\Delta \longrightarrow_{y:\tau,\Sigma} \mathcal{A},B[y/x],\Omega;\Upsilon}{\Gamma;\Delta \longrightarrow \mathcal{A},\forall x_\tau.B,\Omega;\Upsilon}\ (\forall_r)$$

Fig. 2. The Forum proof system: Υ denotes a multi-set of *atomic formulae*
.

the extended notion of uniformity seems too coarse to obtain readable and predictable proofs. Thus, the question on how to program with such a specification logic naturally arises. To answer it, in the following section we restrict ourselves to a special fragment of Forum.

4. Towards Executable Specifications

In this section we describe the main features of the logic of *extended hereditary Harrop formulae* (Delzanno 1997; Delzanno and Martelli 1998), a fragment of Forum that can be used as an *executable* specification language.

$$\frac{}{\Gamma;\ \xrightarrow{D}_{\Sigma} \mathcal{A}}\ initial$$

$$(A_1 \mathbin{⅋} \ldots \mathbin{⅋} A_n) \in \langle D \rangle,\ \{A_1, \ldots, A_n\} \equiv \mathcal{A}.$$

$$\frac{\Gamma, D; \Delta \xrightarrow{D}_{\Sigma} \mathcal{A}}{\Gamma, D; \Delta \longrightarrow_{\Sigma} \mathcal{A}}\ decide_1 \qquad \frac{\Gamma; \Delta \xrightarrow{D}_{\Sigma} \mathcal{A}}{\Gamma; \Delta, D \longrightarrow_{\Sigma} \mathcal{A}}\ decide_2$$

$$\frac{}{\Gamma; \Delta \longrightarrow_{\Sigma} \top, \Omega}\ \top_r \qquad \frac{\Gamma; \Delta \longrightarrow_{\Sigma} \Omega}{\Gamma; \Delta \longrightarrow_{\Sigma} \bot, \Omega}\ \bot_r \qquad \frac{\Gamma; \Delta \longrightarrow_{\Sigma'} A[y/x], \Omega}{\Gamma; \Delta \longrightarrow_{\Sigma} \forall_\tau x.A, \Omega}\ \forall_r$$

$$\frac{\Gamma; \Delta \longrightarrow_{\Sigma} A_1, A_2, \Omega}{\Gamma; \Delta \longrightarrow_{\Sigma} A_1 \mathbin{⅋} A_2, \Omega}\ ⅋_r \qquad \frac{B, \Gamma; \Delta \longrightarrow_{\Sigma} A, \Omega}{\Gamma; \Delta \longrightarrow_{\Sigma} B \Rightarrow A, \Omega}\ \Rightarrow_r \qquad \frac{\Gamma; B, \Delta \longrightarrow_{\Sigma} A, \Omega}{\Gamma; \Delta \longrightarrow_{\Sigma} B \multimap A, \Omega}\ \multimap_r$$

$$\frac{\Gamma; \Delta \longrightarrow_{\Sigma} A_1, \Omega \quad \Gamma; \Delta \longrightarrow_{\Sigma} A_2, \Omega}{\Gamma; \Delta \longrightarrow_{\Sigma} A_1 \ \&\ A_2, \Omega}\ \&_r$$

$$\frac{\Gamma;\ \longrightarrow_{\Sigma} G \quad \Gamma; \Delta \longrightarrow_{\Sigma} B, \mathcal{A}'}{\Gamma; \Delta \xrightarrow{D}_{\Sigma} \mathcal{A}, \mathcal{A}'}\ bc$$

$$G \Rightarrow (A_1 \mathbin{⅋} \ldots \mathbin{⅋} A_n \mathrel{\circ\!\!-} B) \in \langle D \rangle,\ \{A_1, \ldots, A_n\} = \mathcal{A}.$$

Fig. 3. $\mathcal{E}_{hhf}$-derived rules. In $(\forall_r)$, $y : \tau \notin \Sigma$ and $\Sigma' = y : \tau, \Sigma$.

4.1. *Syntax of the Language $\mathcal{E}_{hhf}$*

The logic of $\mathcal{E}_{hhf}$ is based on the fragment of linear logic connectives $\multimap$, $\Rightarrow$, $\&$, $⅋$, $\bot$, $\top$, and the universal quantification. Forum formulae are further restricted as given by the production $\mathcal{D}$ (clauses) and $\mathcal{G}$ (goals) of the following grammar:

$$\begin{array}{lcl}
\mathcal{D} & ::= & \mathcal{D} \,\&\, \mathcal{D} \mid \forall x.\mathcal{D} \mid \mathcal{H} \mathrel{\circ\!\!-} \mathcal{G} \mid \mathcal{G} \Rightarrow \mathcal{D} \mid \mathcal{H}. \\
\mathcal{G} & ::= & \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{G} \mathbin{⅋} \mathcal{G} \mid \forall x.\mathcal{G} \mid \mathcal{D} \Rightarrow \mathcal{G} \mid \mathcal{D} \multimap \mathcal{G} \mid A_r \mid \bot \mid \top. \\
\mathcal{H} & ::= & \mathcal{H} \mathbin{⅋} \mathcal{H} \mid A_r.
\end{array}$$

Here, A_r is any *rigid* atomic formula with type o (i.e., $A_r = (f\ t_1\ \ldots\ t_n)$ with $f \in \Sigma$ a constant symbol). $\mathcal{E}_{hhf}$-sequents have the following form: $\Gamma; \Delta \longrightarrow_{\Sigma} \Omega$ or $\Gamma; \Delta \xrightarrow{D}_{\Sigma} \mathcal{A}$, where Γ is a set of $\mathcal{D}$-formulae, Δ is a multiset of $\mathcal{D}$-formulae, Ω is a multiset of $\mathcal{G}$-formulae, D is a $\mathcal{D}$-formula and $\mathcal{A}$ a multiset of atomic formulae. All the formulae are defined over the signature Σ. Intuitively, a sequent denotes an instantaneous configuration of the computation denoted by its proof. The left-hand side is the current program (with unbounded and bounded-use clauses) and the right-hand side is the current multiset of goals to be executed.

The set of Forum rules can be specialized to the considered type of sequents as shown in Fig. 4.1. All the rules are modulo λ-conversion. Note that the left rules collapse into one single backchaining rule, namely *bc*. In such a rule $\langle D \rangle$ represents the set of instances of a multiset Δ of $\mathcal{D}$-clauses over a signature Σ. Precisely, $\langle \Delta \rangle$ is the smallest set of D-clauses such that: $\langle \Delta \rangle = \bigcup_{D \in \Delta} \langle D \rangle$; $\langle D_1 \& D_2 \rangle = \langle D_1 \rangle \cup \langle D_2 \rangle$; $\langle \forall_\tau x.D \rangle = \langle \{D[t/x] \mid t : \tau \text{ is a } \Sigma\text{-term}\} \rangle$; and $\langle A \rangle = \{A\}$, otherwise.

4.2. *Expressiveness of the Language*

Horn clauses and hereditary Harrop formulae are a particular case of $\mathcal{E}_{hhf}$, i.e., they can be embedded into $\mathcal{E}_{hhf}$ preserving *provability.* Horn clauses can be expressed by $\mathcal{D}$-clauses of the form $G \Rightarrow A$, where A is a rigid atomic formulae and G a goal formula in one of the following forms: $\top$, $B_1 \,\&\ldots\&\, B_n$ where the B_i's are atomic formulae.
Hereditary Harrop formulae are obtained from the grammar in Section 4.1 by removing the productions for the connectives $\invamp$, $\multimap$ and $\bot$.

4.3. *Computational View of Proofs*

Clauses can be rewritten as conjunctions $D_1\&\ldots\&D_m$ of formulae in the following form:

$$\forall\, G_1 \,\&\ldots\&\, G_n \quad\Rightarrow\quad (A_1 \invamp \ldots \invamp A_n \circ\!\!- G),\ n \geq 1$$

where G and $G_1,\ldots,G_n$ are goal formulae. Such a clause has the following intuitive operational meaning: *if the goals $G_1,\ldots,G_n$ can be proved, then the multiset $A_1,\ldots,A_n$ in the right-hand side of the current sequent can be rewritten into G.* This idea is formalized by the backchaining rule (bc) in 4.1. Note that, given a sequent s, there can be several different ways of applying a rule D in a *bc* step to s, depending on the set $\langle D\rangle$. In practice, the non-determinism induced by the *bc* rule is managed by the use of *backtracking* (i.e., the possibility to roll back to a choice point).

As specified by the rule *initial*, a backchaining step over a clause D of the form $A_1 \invamp \ldots \invamp A_n$ (i.e. without body w.r.t. the nested implication $\circ\!\!-$) corresponds to the conclusion of the computation w.r.t. to a given property of the current state we want to observe. In fact, in this stage we *observe* the final state of the computation by *matching* D with the right-hand side of the current sequent. The rule $decide_1$ allows one to use an unlimited number of times the clauses in a program Γ, whereas the rule $decide_2$ removes the selected clause from the current context.

4.4. *New Types of Goals*

Goal formulae have a richer structure than goals in classical logic programming. For instance, goals of the form $G_1 \invamp G_2$ express in a natural way the concurrency in the execution of the subgoals G_1 and G_2. The two subgoals are, in fact, simultaneously rewritten in the current multiset of goals to be proved. Linear implications of the form $D \multimap G$ allow one to enrich the current multiset of bounded-use clauses with a new one, see rule $\multimap_r$. The newly introduced resource must then be consumed during the rest of the proof, i.e., it will be applied in a successive backchaining step. Here, we require D to be a clause at the moment of the execution of such a goal. In (Delzanno and Martelli 1998), the authors study a variation on the language in which *variables* of type o can occur in $\mathcal{D}$-position within the scope of a goal and they prove that the resulting language is closed under instantiation. We shall use this feature in the rest of the paper. The logical constant $\bot$ can be used to *remove* resources from the right hand side of sequents, e.g., by using a clause of the form $A \circ\!\!- \bot$. $\top$ makes a proof to succeed under any bounded context. The other connectives have the operational interpretation inherited by the logic of hereditary Harrop formulae (Nadathur and Miller 1988). For instance, (Miller 1989b) shows how to employ the side condition of the rule $\forall_r$ to introduce a notion of data abstraction over components of the current configuration.

4.5. *Refinement of the Syntax*

In order to emphasize the computational view of proofs, we introduce a refined syntax for sequents and clauses.

Sequents assume the following form: $\Gamma; \Delta \longrightarrow_\Sigma \Omega \;\|\; \Theta$, where the right hand side is partitioned into two multisets: the *operations*, Ω, and the *state atoms*, Θ. State atoms are nothing but atoms built upon a signature $\Sigma_{\mathcal{R}}$ of state constructors fixed a priori. By default, $\Sigma_{\mathcal{R}}$ is contained in any signature Σ mentioned in the rest of the paper. We assume that each atomic formula occurring in the right hand side of a sequent is implicitly *moved* into the current Θ. According to it, we shall adopt a slightly different syntax for clauses, namely

$$\underbrace{C_1 \;\&\; \ldots \;\&\; C_n}_{Cond} \texttt{ >> } \underbrace{A_1 \mid \ldots \mid A_m}_{Ops} \texttt{ || } \underbrace{S_1 \mid \ldots \mid S_k}_{State} \texttt{ --> Body || } \underbrace{Z_1 \mid \ldots \mid Z_p}_{New\ \ State}$$

Such an expression will denote the corresponing $\mathcal{E}_{hhf}$-clause:

$$C_1 \;\&\; \ldots \;\&\; C_n \Rightarrow (A_1 \parr \ldots \parr A_m \parr S_1 \parr \ldots \parr S_k) \;\circ\!\!-\; (Body \;\parr\; Z_1 \parr \ldots \parr Z_p).$$

The symbol `||` in the clauses is used to distinguish state-atoms from operations in accord with the notation used in the sequents. The free variables of this expression (denoted by identifiers beginning with a capital letter) are implicitly universally quantified. When necessary, universally quantified formulae $\forall x.F$ will be explicitly represented as `pi X / F` (according to the λProlog notation (Nadathur and Miller 1988)). If a clause has no conditional part, i.e., *Cond* in the previous figure, then we use the notation: OPS || STATE --> BODY || NEW STATE.
Facts without body are written: [COND >>] OPS || STATE. The constant $\perp$, written **anti**, is used to denote an empty BODY, STATE or NEW STATE component. The constant $\top$ will be written as `all`. The previous notation puts in relief the similarities between $\mathcal{E}_{hhf}$-clauses and rewriting rules, between $\mathcal{E}_{hhf}$-sequents and instantaneous configurations, and between $\mathcal{E}_{hhf}$-proofs and state-based computations.

4.6. *Final state of a computation*

The final state of a computation corresponds to a sequent of the form: $\Gamma; \emptyset \longrightarrow_\Sigma \emptyset \;\|\; \Theta$ where Θ is a multiset of $\Pi_{\Sigma_{\mathcal{R}}}$ atoms. Such a sequent is not an axiom unless there exists a formula Φ in Γ which *matches* the facts in Θ. Such a formula Φ can be easily *re-constructed* after having built a partial proof tree with leaves of the form $\Gamma; \emptyset \longrightarrow_\Sigma \emptyset \;\|\; \Theta$. For this reason, we single out Φ from the unbounded context and adopt the following entended syntax for the sequents: $\Gamma[\Phi]; \emptyset \vdash_\Sigma \Omega \;\|\; \Theta$. We omit Φ when we are not interested in the final state of a proof (e.g. when the proof is terminated by $\top$).

Given an $\mathcal{E}_{hhf}$-theory (program) Γ and a collection of goals Ω, a query (initial state) will assume the form $\Gamma[\Phi]; \emptyset \vdash_\Sigma \Omega \;\|\; \emptyset$, where Φ can be considered as a variable instantiated with the possible final states of the execution of Ω.

$$
\dfrac{\dfrac{\begin{matrix}\vdots\ bc\\ \Gamma;\ \vdash_\Sigma (add\ 0\ 1\ 1)\ \|\end{matrix} \qquad \dfrac{\begin{matrix}\vdots\ bc\\ \Gamma;\ \vdash_\Sigma (add\ 1\ 3\ 4)\ \|\end{matrix} \qquad \dfrac{}{\Gamma[\Phi];\ \vdash_\Sigma\ \|\ (counter\ 4)}\ verify}{\Gamma[\Phi];\ \vdash_\Sigma \underline{inc\ 3}\ \|\ \underline{(counter\ 1)}}\ bc+gr}{\Gamma[\Phi];\ \vdash_\Sigma \underline{inc\ 1}, inc\ 3\ \|\ \underline{(counter\ 0)}}\ bc+gr}{\Gamma[\Phi];\ \vdash_\Sigma \underline{new}, inc\ 1, inc\ 3\ \|}\ bc
$$

Fig. 4. A sample derivation

4.7. *An Example.*

Let us consider the following informal specification: *We want to specify the behavior of a set of counters which can be modified randomly by using an operation (inc Z).*

Let Σ be a signature with the symbols $new, inc, counter, add$ with adequate types and such that $(countert) \in \Pi_{\Sigma_\mathcal{R}}$, for each t. The following clauses, which form Γ, implement the specification:

```
new --> counter 0.
add X Z Y >> inc Z || counter X --> counter Y.
```

the former simply rewrites the atomic goal *new* into a new *counter* initialized to 0 (there is no guard in this clause), whereas the latter *synchronizes* an *inc*-message and a *counter*-formula consuming and rewriting them into the updated counter (the guard simply executes the increment). The *add* predicate is defined by the following Horn-clauses:

```
all >> add X 0 X.
add X Y Z >> add (s X) Y (s Z).
```

In the following we shall use $0, 1, etc$ for $0, (s\ 0), etc.$
In Figure 4 we show a computation (proof) derived by using the previous program and the rules of Fig. 4.1. Notice that the two *inc* operations could be executed in any order, since the Ω component of a sequent is a multi-set: in this case their execution yields always the same final state.

The main thread of the computation corresponds to the right branch which models the evolution of the state of the counter, whereas, the left branches are used here to test the conditional parts of the rules. A final observation about the structure of the proof: the left branches are terminated by *initial* axioms matching facts in Γ with atomic goals in Ω; the right branch is terminated by an instance of the *initial* axiom which matches a particular formula consisting only of $\Pi_{\Sigma_\mathcal{R}}$ predicates with the final state of the computation. According to our previous notation, the formula Φ in Figure 4 corresponds to *counter* 4.

In the above sketched scheme the non-determinism is due to the choice of the clause to apply (which induces a choice on the multiset of atoms in the right-hand side that must be rewritten), and, in the effective implementation, to higher-order unification. In this sense the chosen class of formulae can be considered as an extension of hereditary Harrop formulae (λ-Prolog) with the possibility to handle resources and concurrent actions.

5. Concurrent Object-Oriented Programming

Programs provide for automated management of data. From the users perspective it is important to know which are the effects of executing a program, i.e., its functionalities, but not how they are actually performed, or in which form the data are internally stored. On the other side, a programmer needs to adapt the software to a given specification, which, in turn, depends on the requirements of the real problem, which can change in the time. Thus, it is important to have the possibility to modularly modify programs. Object-orientation provides with an interpretation of programs, which can ease both the task of users and developers.

In this setting new concepts like *class* and *object* (Wegner 1987), are added to the traditional programming ones. A *class* corresponds to a data type describing which are the functionalities of a given part of a program (data and operations). The users can only access the external interface of operations and data. An *object* corresponds to a given instance of a class; it encapsulates (Snyder 1986) data and operations (called *methods*) which handle them and it has an internal *state*, whose components can be changed by executing the methods. Objects can receive *messages* accepting only the ones declared in their classes. A message (method invocation) corresponds to the activation of one method in the object. Methods can refer to the current object by using a *self* pointer (e.g. in their definition they can invoke other methods belonging to the same object by referring to self). Classes can be composed by defining hierarchies which allow to share code among different classes. The *inheritance* relations can be defined in different ways, see again (Snyder 1986; Wegner 1987) (e.g. single- and multiple-inheritance, overriding, etc.). As said before objects are created by instantiating classes. Usually, objects have unique identifiers and store a private copy of data, operations, with links to shared ones. Object-based languages are directly based on the notion of object, providing operations to create them and to define clones. In this setting *delegation* is used to share code and data among objects (i.e., an object delegates another object to perform a given operation).

Object-Orientation can fit both in a sequential and in a concurrent setting. In a sequential model objects are passive except when they must react to a given message, i.e., the current operation in the execution thread of the program. On the other hand, in a concurrent model, where the unit of parallelism can be a process as in CSP (Hoare 1978), a task as in Ada, a statement as in Occam, a goal as in Concurrent Prolog (Shapiro 1989), objects can play different roles. For instance, objects can be considered as *active* entities having their own execution threads activated at the moment of creation. An active object can be considered as an evolute form of process. Otherwise, it is possible to employ asynchronous communication, e.g., the Actor model (Agha 1986), or to add processes as an orthogonal concept to the object oriented dimension. Active objects are particularly suitable to safely share information in a distributed system.

In the last years, interesting works based on LL, have been also proposed in order to deal with all these aspects by exploiting its richer set of logical features provided.

6. Specification of Concurrent Objects

In this section we introduce a linear logic formalization of several different features of object-oriented programming. Such a specification is based on the view of *objects* as *atomic formulae*. In a higher-order setting this view allows us to capture the key point of the *object-orientation*, i.e., *encapsulation*. Before getting into the detailed encoding of our *object-based* model, we shall briefly discuss the problems related with *class-based* models.

6.1. *Class-based Representation*

In classical logic programming a direct way to design logical specification with an object-oriented flavor is the following: We interpret a set of clauses defining a predicate `p` as the class representing the set of objects of the form `p(t)`, see, e.g., (McCabe 1992). By exploiting the additional features of linear logic, we can refine such an idea as follows: Each class can be represented as a program defining a given predicate `classname` such that each single method synchronizes an object of the form `classname State` and a message `message`:

```
cond >> message || classname State --> msg1 |... | msgn || classname State'.
```

Let us give an example.

Example 6.1. We want to specify the class `rpoly` of *regular polygons.* To represent a regular polygon we just need the number of faces (the rank) and the length of one of the faces. We assume here that they are natural numbers. An object of type `rpoly` is defined as an atom `(rpoly face rank perimeter)`. Furthermore, we define two methods: `set`, to correctly initialize the data fields, and `get`, to retrieve the data values. Formally, we introduce the following signature:

`rpoly`: $\text{int} \to \text{int} \to \text{int} \to \text{o}$.
`set`: $\text{int} \to \text{int} \to \text{o}$.
`perimeter`: $\text{int} \to \text{o}$.

The methods are defined as follows:

```
Perim is Face*Rank  >>
   set Face Rank  || rpoly F R P   --> anti || rpoly Face Rank Perim.   (a)

   get F R P      || rpoly  F R P  --> anti || rpoly F R P.             (b)
```

A possible proof for the query $\Gamma[\Phi]; \emptyset \vdash_\Sigma (set\ 5\ 4), (get\ F\ R\ P) \parallel (rpoly\ v\ w\ z)$ is shown in Fig. 5. To complete the proof, Φ can be set to $(rpoly\ 5\ 4\ 20)$, i.e., to the *final state* of the computation. Another possible proof-tree can be obtained in *backtracking* by selecting the method *get* first.

$$
\dfrac{
  \dfrac{\dots}{\Gamma;\emptyset \longrightarrow_{\Sigma} P' \ is\ 5*4}
  \qquad
  \dfrac{
    \dfrac{
      \dfrac{}{\Gamma[\Phi];\emptyset \longrightarrow_{\Sigma} \emptyset \ \| \ (rpoly\ 5\ 4\ P')}\ initial
    }{
      \Gamma[\Phi];\emptyset \stackrel{(b)}{\longrightarrow}_{\Sigma} (get\ F\ R\ P) \ \| \ (rpoly\ 5\ 4\ P')
    }\ bc
  }{
    \Gamma[\Phi];\emptyset \longrightarrow_{\Sigma} (get\ F\ R\ P) \ \| \ (rpoly\ 5\ 4\ P')
  }\ decide_1
}{
  \dfrac{
    \Gamma[\Phi];\emptyset \stackrel{(a)}{\longrightarrow}_{\Sigma} (set\ 5\ 4), (get\ F\ R\ P) \ \| \ (rpoly\ v\ w\ z)
  }{
    \Gamma[\Phi];\emptyset \longrightarrow_{\Sigma} (set\ 5\ 4), (get\ F\ R\ P) \ \| \ (rpoly\ v\ w\ z)
  }\ decide_1
}\ bc
$$

Fig. 5. A sample proof: $P = P' = 20,\ F = 5,\ R = 4$

Note that the *conditions* of a $\mathcal{D}$-clause, e.g., `Perim is Face*Rank`, are solved on an *auxiliary* branch for which we do not need to know the *final state*. In fact, for the sake of this paper, the predicates invoked in the conditions are just *Horn logic programs*. e.g., the predicate that defines `''is''`. The result of such a program is communicated to the main thread of the computation via shared variables.

Inheritance can be achieved by using the approach of Andreoli and Pareschi (Andreoli and Pareschi 1991), i.e., employing disjunctions ($\parr$) to represent composite objects. For instance, the subclass of *colored polygons* can be defined by just adding a further component (atom) with the `color` data field to the class of polygons. More precisely, given the signature

`crpoly`: int $\rightarrow$ int $\rightarrow$ string $\rightarrow$ o.
`color`: string $\rightarrow$ o,

we add the clauses:

```
anti  || crpoly Face Rank Color  --> anti || rpoly Face Rank 0 | color Color.

setcolor Color || color C        --> anti ||  color Color.
```

Thus, `crpoly`-objects have the form $(rpoly\ F\ R\ P) \parr (color\ C)$. Note that the methods of the superclass (e.g. `set` and `get`) still apply to them. This approach is followed in LO (Andreoli and Pareschi 1991). Here, objects of the previous form are spread on different branches of a proof. Communication is achieved through a common blackboard. This model does not provide *overriding* of methods. For instance, let us now define the class *circle* as a subclass of *rpoly*, the limit case, indeed, in which the number of faces is infinite. We add then the following clauses:

```
  anti   ||  circle Ray Perim --> anti ||  rpoly Ray infinity Perim.

Perim is 2*Ray*pi_greek >>
  set  Ray  || rpoly R infinity P --> anti || rpoly Ray infinity Perim.
```

In such a hierarchy, however, nothing prevents the method `set` of the superclass `rpoly` from being selected after *opening* a `circle`-object. This might lead to run-time errors, e.g., multiplication of `Ray` by `infinity` which, however, is just a *dummy* constant.

As outlined in (Abadi and Cardelli 1994; Fisher *et al.* 1994), a better understanding of the basic concepts of object-orientation can be achieved by focusing on *object-based*

% Objects and Messages

`object`:	Identifier $\to$ o $\to$ o $\to$ o.
`frozen`:	Identifier $\to$ o $\to$ o $\to$ o.
`send`:	Identifier $\to$ Msg $\to$ o.
`call`:	Identifier $\to$ Msg $\to$ o.

Fig. 6. The signature $\Sigma_{\mathcal{R}}$ consists of `object` and `frozen`.

languages. Classes can be viewed as particular *meta*-objects. This is the goal (and the main novelty with respect to Andreoli and Pareschi's approach) of the following section.

6.2. *An Object-based Representation*

Encapsulation is a central point in the objects metaphor: *objects are black boxes, with unique identity, containing data and operations and exchanging messages in order to interact with the external world.* According to it, in our setting an object will be defined by an atom of the following form:

```
object id state methods,
```

where `state` is the set of attributes, and `methods` are *clauses* defining the behavior of the object (see the signature in Fig. 6). A message (method invocation) is defined as an atom `call id msg` (see again Figure 6). Based on it, we shall consider a sequent $\Gamma;\Delta \longrightarrow_{\Sigma} \Omega \parallel \Theta$ as a configuration of an object-based system, where Γ (the program) specifies the *external* behavior of objects, Θ (the global state) the set of active objects, and Ω the set of active messages. We shall clarify later the role of Δ in such a representation.

In the following we shall consider a model of computation in which the messages are delivered without any order (e.g. in the style of actors (Agha 1986)) and executed in interleaving. The methods of a given object `(object id state methods)` have the following form, m_1 `&` ... `&` m_n where each conjunct is a clause of the form:

```
cond >> call id Mgs || object id St Ms --> msg1 |...| msgn || object id St' Ms'.
```

Such a clause rewrites the object `id` and a message into a (possibly modified) copy of the object and a multiset of new messages. The newly produced messages correspond to the body of the invoked method. To handle such a recursive definition (an object contains its methods defined in terms of the object itself) we have to abstract from some of the components of the object-representation. To clarify, let us consider the following simple method:

```
call id (get V) || object id V M --> anti || object id V M.
```

it is supposed to retrieve the value stored in the object. On the other hand, the methods `M` must be unfolded only at the time of execution. For this purpose, we rewrite the method as follows:

```
pi V \ pi M \ (call id (get V) || object id V M --> anti || object id V M),
```

using quantification over variables of type o to leave unspecified the methods in the nested object-atoms. Thus, we construct the object as follows:

```
object id v
  (pi V \ pi M \
     (call id (get V) || object id V M --> anti || object id V M)
  ).
```

The third parameter of the `object` predicate is a (universally quantified) $\mathcal{D}$-clause. For simplicity we assume that the state denotation, `v` in the example, has type o, as well. For instance, a tuple can be represented by a constructor `(st A B ...)`: o.

6.3. *Method Invocation*

In order to execute a method it is necessary to:

i) synchronize the object and the message;
ii) fire the methods, i.e., move them at the top level in the left hand side of sequents (methods are indeed clauses);
ii) execute a backchaining step over the $\mathcal{D}$-clauses defining the methods.

To prevent anomalies we need to enrich the syntax of our representation (see Fig. 6). The new predicate `send` is used to represent messages before step i, i.e., messages that require the corresponding definition to be fired. The predicate `call` will instead be used to represent messages during steps ii and iii, i.e., messages for which their definition has already been fired. Finally, to prevent the execution of a `call` from being delayed after step ii, we inhibit other receptions by *freezing* the considered object after step i. Methods assume then the following form:

```
call id Msg || frozen id V Ms --> msg1 |...| msgn || object id V' Ms'.
```

Note that, if we don't freeze objects before steps ii and iii, an interleaved execution of another message could change the structure of the original object while a copy of its old methods exists. In synthesis, `send` will be used as *external* primitive to invoke methods, whereas `call` will serve to internally dispatch the message. Formally, such semantics is formalized through a single $\mathcal{E}_{hhf}$-rule.

Definition 6.1. (Self-application) The semantics of method-application (steps i, ii and iii of Section 6.2) is defined by the following clause:

```
send Id Msg || object Id St Ms --> (Ms -o call Id Msg) || frozen Id St Ms.
```

Step i is expressed by the head of the previous clause. Note that such a rule can be applied in *any* context, i.e., with any number of messages and objects in the global state, provided the object `Id` exists. Step ii is achieved by the goal `(Ms -o call Id Msg)`: the methods are fired and they are *consumed* right after their execution. Finally, by the adding `call Id Msg` to the current operations and by freezing the object we start up the execution of the method. In fact, by hypothesis the methods in `Ms` *define* the predicate `call` for the considered object `Id`. Note that `Ms` is a &-conjunction of methods. The nondeterminism in the choice of one of its conjuncts (see rule bc) guarantees the selection of the right clause.

Example 6.2. Let us reformulate the `polygon` objects of Section 6.1 in the new setting.

$$\begin{array}{cl}
\dfrac{\ldots}{\Gamma;\emptyset \longrightarrow_{\Sigma} X\ is\ 2*3} \qquad \begin{array}{c}\vdots\\ \Gamma[\Phi];\emptyset \longrightarrow_{\Sigma} \Omega \parallel o_{2,3,X},\Theta\end{array} & \\ \hline
\Gamma[\Phi];\emptyset \overset{Ms}{\longrightarrow}_{\Sigma} (call\ id\ (set\ 2\ 3)),\Omega \parallel F_{5,4,w},\Theta & bc\ over\ (b) \\ \hline
\Gamma[\Phi];Ms \longrightarrow_{\Sigma} (call\ id\ (set\ 2\ 3)),\Omega \parallel F_{5,4,w},\Theta & decide_2 \\ \hline
\Gamma[\Phi];\emptyset \longrightarrow_{\Sigma} Ms \multimap (call\ id\ (set\ 2\ 3)) \parr F_{5,4,w},\Omega \parallel \Theta & \parr_r + \multimap_r \\ \hline
\Gamma[\Phi];\emptyset \overset{(send)}{\longrightarrow}_{\Sigma} (send\ id\ (set\ 2\ 3)),\Omega \parallel o_{5,4,w},\Theta & bc \\ \hline
\Gamma[\Phi];\emptyset \longrightarrow_{\Sigma} (send\ id\ (set\ 2\ 3)),\Omega \parallel o_{5,4,w},\Theta & decide_1
\end{array}$$

Fig. 7. A sample (piece of) proof: $X = 6$.

```
object id (s face rank perim) Ms
 where Ms =
  pi Face \ pi Rank \ pi Perim \ pi F \ pi R \ pi P \
   (Perim is Face*Rank  >>
     (call id (set Face Rank)  || object id (s F R P)  -->                  (a)
                       anti    || object id (s Face Rank Perim)))
        &
     (call id (get Face Rank Perim)) || object  id (s Face Rank Perim)  -->  (b)
                               anti || object  id (s Face Rank Perim)
```

Note that the methods incorporate the identifier of the object to which they belong. In the following, we denote such an object by $o_{f,r,p}$ and the corresponding frozen object by $F_{f,r,p}$, where f, r, p are the values of the attributes `Face, Rank, Perimeter`. Then, the query $\Gamma[\Phi]; \emptyset \vdash_{\Sigma} (send\ id\ (set\ 2\ 3)) \parallel o_{5,4,w}$ has the (partial) $\mathcal{E}_{hhf}$-proof tree of Fig. 7. Note that the method (b) (contained in Ms) is first fired by the *send* rule and after *removed* by the $decide_2$ rule. Since the *right* instance of b is in $\langle Ms \rangle$, the *bc* step over Ms succeeds and it has the desired effect.

In the next section we shall study the proof-theoretical properties of the object-calculus resulting from the above described encoding.

6.4. *Proof-theoretical Analysis*

Throughout the rest of the section, we shall consider Γ as the singleton consisting of the clause of Def. 6.1, Θ as a multiset of `object` and `frozen` atoms, and Ω as a multiset of `send` and `call` atoms. Furthermore, we say that a global state Θ is consistent if the identifiers of the object occurring in Θ are different from each other. We introduce then the following derived rule.

Definition 6.2. (The *send* schema) The *send* scheme consists of the following sequence of rules:

$$\begin{array}{cl}
\Gamma[\Phi]; Meths, \Delta \vdash_{\Sigma} \texttt{call id m}, \Omega \parallel \texttt{frozen id st Meths}, \Theta & \\ \hline
\Gamma[\Phi]; Meths, \Delta \vdash_{\Sigma} \texttt{frozen id st Meths} \parr \texttt{call id m}, \Omega \parallel \Theta & \parr_r \\ \hline
\Gamma[\Phi]; \Delta \vdash_{\Sigma} Meths \multimap (\texttt{frozen id st Meths} \parr \texttt{call id m}), \Omega \parallel \Theta & \multimap_r \\ \hline
\Gamma[\Phi]; \Delta \overset{(send)}{\longrightarrow}_{\Sigma} \texttt{send id m}, \Omega \parallel \texttt{object id st Meths}, \Theta & bc \\ \hline
\Gamma[\Phi]; \Delta \vdash_{\Sigma} \texttt{send id m}, \Omega \parallel \texttt{object id st Meths}, \Theta & decide_1
\end{array}$$

We observe that the right rules occurring immediately above the *bc* can always be permuted so as to obtain the previous configuration. Thus, without loss of generality, given a sequent $\Gamma[\Phi]; \Delta \vdash_\Sigma \Omega \parallel \Theta$ we can restrict our attention to proofs in which each applications of a backchaining step (over the clause of Def. 6.1) matches the previous scheme. We denote such a derived rule as *send* rule.

Lemma 6.3. (Permutability of the *send* rule) Let Γ consists of the rule 6.1, Θ_o and Ω be multisets of objects and messages. Let δ be an $\mathcal{E}_{hhf}$-proof for $s = \Gamma[\Phi]; \Delta \vdash_\Sigma \Omega \parallel \Theta$, s.t. the last rule of δ is a *send* rule, applied to an object named *id* and introducing $Meths_{id}$ on the left hand side of its premise. Finally, let us assume that Δ does not contain any other methods labeled by *id* and that each state Θ_i occurring in δ is consistent. Then, there exists a proof γ for s, where, $Meths_{id}$ is consumed immediately after the application of the *send* rule which introduces it.

Proof. Let $send_{id}$ the last occurrence of *send* applied to the object denoted by *id*. The proof is by induction on the number of rule applications between the $send_{id}$ rule and the corresponding *consumption* of the methods $Meths_{id}$. The base case is trivial. If the distance is greater than zero, then the proof is by cases on the rule R that occurs immediately above the considered instance of the $send_{id}$ rule. There are two cases.

— R is an instance $send_{id'}$ of the *send* rule, where, $id' \neq id$, since id is frozen by $send_{id}$. It is easy to see that two instances of the *send* rule (over different objects) permute. Therefore, after permuting $send_{id'}$ and $send_{id}$, we conclude by applying the inductive hypothesis.

— R is an instance of the *bc* rule over the methods $Meths_{id'}$ of the object $\texttt{frozen id}'$, where, by hypothesis, $id' \neq id$, followed by *deterministic* application of the $\parr_r$ rule, as shown in the following picture:

$$\dfrac{\begin{array}{c}\eta \\ \Gamma[\Phi]; Meths_{id}, \Delta \vdash_\Sigma \texttt{call id}, m_1 \ldots m_n, \Omega \parallel \texttt{frozen id}, \texttt{object id}', \Theta \\ \vdots\ R \\ \Gamma[\Phi]; Meths_{id}, Meths_{id'}, \Delta \vdash_\Sigma \texttt{call id}, \texttt{call id}', \Omega \parallel \texttt{frozen id}, \texttt{frozen id}', \Theta\end{array}}{\Gamma[\Phi]; Meths_{id'}, \Delta \vdash_\Sigma \texttt{send id}, \texttt{call id}', \Omega \parallel \texttt{object id}, \texttt{frozen id}', \Theta}\ send\ ,$$

where, $\texttt{frozen id}' \parr \texttt{call id}' \multimapinv \texttt{object id}' \parr m_1 \ldots m_n \in \langle Meths_{id'} \rangle$. In this case, we can re-arrange the proof as follows:

$$\begin{array}{c}\dfrac{\begin{array}{c}\eta \\ \Gamma[\Phi]; Meths_{id}, \Delta \vdash_\Sigma \texttt{call id}, m_1, \ldots, m_n, \Omega \parallel \texttt{frozen id}, \texttt{object id}', \Theta\end{array}}{\Gamma[\Phi]; \Delta \vdash_\Sigma \texttt{send id}, m_1 \ldots m_n, \Omega \parallel \texttt{object id}, \texttt{object id}', \Theta}\ send \\ \vdots\ R \\ \Gamma[\Phi]; Meths_{id'}, \Delta \vdash_\Sigma \texttt{send id}, \texttt{call id}', \Omega \parallel \texttt{object id}, \texttt{frozen id}', \Theta.\end{array}$$

We conclude, by applying the inductive hypothesis on the subproof η, that there exists γ satisfying the thesis.

Notice that in each case the length of the resulting proof γ is equal to the length of δ. $\square$

In order to emphasize the operational aspects of the $\mathcal{E}_{hhf}$-theory introduced before we define the following operational semantics:

Definition 6.4. (Operational Semantics) Let $\Omega, \Omega', \Theta, \Theta'$ be multisets of formulae. The relation $\rightsquigarrow_o$ is defined as follows:

$$\langle \Omega \mid \Theta \rangle \quad \rightsquigarrow_o \quad \langle \Omega' \mid \Theta' \rangle$$

iff `(send id n)`$\in \Omega$ and `(object id st ms)`$\in \Theta$,
`(call id m || object id st mts --> msgs|| object id st' mts')`$\in \langle$`mts`$\rangle_\Sigma$,
`msgs=`m_1 `| .. | n` m_n,
$\Omega' = \Omega \backslash \{$`(send id n)`$\} \cup \{m_1, \ldots, m_n\}$,
$\Theta' = \Theta \backslash \{$`(object id st ms)`$\} \cup \{$`(object id st' ms')`$\}$.

Let $\rightsquigarrow_o^\star$ be the transitive closure of $\rightsquigarrow_o$. The following result holds.

Proposition 6.5. (Provability and Operational Semantics) Let Γ consist of clause 6.1, Θ_o be a multiset of `object`-atoms, and Ω be a multiset of `send`-atoms. Then, $\Gamma[\Phi]; \emptyset \vdash_\Sigma \Omega \parallel \Theta_o$ has an $\mathcal{E}_{hhf}$-proof iff $\langle \Omega \mid \Theta_o \rangle \rightsquigarrow_o^\star \langle \emptyset \mid \Theta_f \rangle$ and $\Phi \multimap \Theta_f$ is provable in $\mathcal{E}_{hhf}$.

Proof. By induction on length of a proof and a case analysis based on lemma 6.3 □

We shall use this semantics (and its extensions, as explained later) to simplify the description of the object behavior.

6.4.1. *Classes.* Classes can be viewed as templates for the creation of *clones* of a given object. We can enrich the $\mathcal{E}_{hhf}$-theory Γ with a very simple type of *class* declaration. We introduce, for this purpose, the predicate `new` with type `Class -> o -> Identifier -> o`. A class declaration assumes the following form:

```
new classname InitState Id || anti --> anti || object Id InitState Mts.
```

Here, `InitState` represents the initial values of the attributes of the object being created. An invocation of the *primitive*, i.e. visible to any object, `new c s id` generates a new instance of the class named c, with an initial state `s`, and an identifier `id`. In the following we enrich the possible types of messages with such a `new`-atom.

Example 6.3. The `rpoly` class can be defined as:

```
new rpoly (st F R P) Id || anti --> anti || object Id (st F R P) Ms.
```

where `Ms` is defined as in the Ex. 6.2.

The considered model captures many other features typical of object-oriented programming. We shall discuss them next.

6.5. *Hiding*

As shown in (Hodas and Miller 1989; Miller 1989b), universal quantification can be used to create *local* data structures. In our setting there are three possible ways of using universal quantifications:

— before an object declaration to *hide data fields*;
— before an object declaration to provide *private methods*;
— before the body of a method to provide creation of *private objects*.

The first two points require the following modification of the previously introduced `new` definition:

```
new classname InitState Id || anti -->
      pi X \( anti || object Id InitState' Mts ).
```

The following examples will illustrate how to exploit such a feature.

6.5.1. *Hiding of Data Fields.* In the scheme presented above any object may freely access to the attributes of the other objects. Let us consider the following rule:

```
break State || object Id State Mts --> all || anti.
```

Through unification, the predicate `break State` may break the encapsulation of any of the objects in the current state. In order to prevent such a violation, we shall employ the universal quantifier as in the example below.

Example 6.4. The `rpoly` class of Ex. 6.3 can be re-defined as:

```
new rpoly (st F R P) Id || anti -->
         pi st \ (anti || object Id (st F R P) Ms).
```

where `Ms` is defined as in Ex. 6.2. Note that, the quantification is defined over a predicate name, namely `st`. This is allowed by the syntax of $\mathcal{E}_{hhf}$. Such a higher-order quantification hides the occurrences of such a constructor in the definition of the object. For instance, the `break` predicate cannot be applied to such objects since constants introduced with $\forall_r$ (e.g. the one associated with `st` when the body of `new` is executed) cannot be bound to free variables outside the scope of the quantifier, see (Miller 1989b; Hodas and Miller 1989).

6.5.2. *Hiding of Methods.* The same idea can be applied in order to define private methods. A method is private in o if it can be invoked only by other methods of o. To accomplish this it is sufficient to quantify over the names of the private methods. For instance, in the following declaration the method `pub` (with type `o`) is public while `m` (with type `o`) can be invoked only by other methods of the same object.

```
(new private St Id || anti -->
         (pi m \ (anti || object Id St (Ms m))))

where (Ms m) =
   pi S \ pi M \
    (call Id m    || frozen Id S M  --> ... || ...)
     &
    (call Id pub  || frozen Id S M  --> send Id m || object Id S M).
```

The third way to use quantification has effect at the execution of a method. We shall illustrate this point in the following section.

6.5.3. *Creation of Local Objects.* Universal quantification in the body of a method can be used to create data structures local to that method. In particular we can employ this feature to create *local objects*. Consider the following general pattern:

```
cond >> call id msg || frozen id st ms --> pi x \ Body.
```

Now, if x is associated to the identifier of a newly created object, the object will automatically become local to the method. The meta-rule for the self-application is not affected by this modification. However, the *send* scheme of Def. 6.2 must be enriched so as to account for possible applications of the $\forall_r$ rule. We redefine the relation $\leadsto_o$ according to this idea. The new relation is denoted by $\overset{\forall}{\leadsto}_o$. By a case analysis similar to the one in Proposition 6.5, it is easy to state the correspondence between $\overset{\forall^*}{\leadsto}_o$-transitions and $\mathcal{E}_{hhf}$-proofs. We shall consider this feature in the example given in the next section.

6.6. *Expressiveness of the Calculus*

To understand the power of the resulting *object* calculus just note that a method can modify the object to which it belongs. For instance, the method:

```
call kill Id  ||  frozen Id St Ms --> anti ||  anti.
```

has the effect of deleting the object that receives it. Similarly, we can think of methods that *dynamically* modify the attributes or even the methods of the object. We shall clarify this point with some examples.

Example 6.5. (Factorial objects) Let factclass be a class of objects which, upon reception of the message (fact N Obj) (N is an integer and O is an object identifier), send the factorial of N (N!=N*N-1*... and !0=!1=1) to Obj. In the following we shall present an implementation in which the same object can manage many requests at the same time. To implement such type of objects it is necessary to define slave-objects which take in charge the actual computation of the factorial, allowing their masters to process the other possible incoming requests. Let us consider the following signature

factclass,slave, receiver:	o.
fact,st:	int $\rightarrow$ Identifier $\rightarrow$ o.
eval:	int $\rightarrow$ o.

Based on it, the definition of factclass-objects is given as follows:

```
% Class of "factorial objects"
new factclass St Id || anti --> anti || object Id St Ms.

where Ms =

 (pi S \ pi M \ pi N\ pi N1 \ pi O\
         (((N =< 2) >>
               call Id (fact N O) || frozen Id S M -->
                       send O  (eval N)   || object Id S M)
          &
          ((N > 2, N1 is N - 1) >>
             call Id (fact N O) || frozen Id S M  -->
              (pi x \(new slave (st N O) x | send Id (fact  N1 x)))
                                                 || object Id S M
          ))).
```

These objects have only one method (fact N O) defined by two clauses: if N is less or equal to 2, then (eval N), the result of the computation, is immediately sent to the object `O`; in the other cases a new `slave` object `x` is created and the message `fact N-1 x` is sent to `x`. Slave and receiver objects are defined as follows:

```
% Class of "slave objects"
new slave St Id || anti --> anti || object Id St Ms

where Ms =

   (pi N \ pi N1 \ pi N2 \ pi Ms \ pi O \
         ((N2 is N1 * N) >>
                 call Id (eval N1) || frozen Id (st N O) Ms -->
                         send O  (eval N2) || anti)).

% Class "receiver"
new receiver St Id || anti --> anti || object Id St Ms

where  Ms =

   (pi N \ pi S \ pi M \
         call Id (eval N) || frozen Id S M  --> anti || frozen Id (eval N) M).
```

They both expect `(eval N)` messages: a slave object executes a part of the computation of `N!`, it sends the result to another slave object and, then, it disappears, whereas a receiver simply stores the result.
Some abbreviations are needed in order to describe possible derivations based on the previous definitions. Let $\mathrm{O}_{id}[\mathrm{V}, id']$ be a `slave`-object with identifier id and state (V, id'). Now, let $\Theta = \{\mathrm{R}_{id_r}, \mathrm{F}_{id_f}\}$ where R_{id_r} is a *receiver* and F_{id_f} is a *factclass* object. The following picture describes a $\overset{\forall}{\leadsto}_o$-derivation which starts from the state $\langle \Omega \mid \Theta \rangle$, where $\Omega = \{\mathtt{send\ id_f\ (fact\ 4\ id_r)}\}$, i.e., we ask the object id_f to compute 4! and to send the result to id_r:

$$
\begin{gathered}
\langle \Omega \mid \Theta \rangle \overset{\forall}{\leadsto}_o \langle \mathtt{new\ \ slave\ (st\ 4\ id_r)\ x}, \mathtt{send\ id_f\ (fact\ 3\ x)} \mid \Theta \rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \mathtt{send\ id_f\ (fact\ 3\ x)} \mid \mathrm{O}_x[4, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \mathtt{new\ \ slave\ (st\ 3\ x)\ y)}, \mathtt{send\ id_f\ (fact\ 2\ y)} \mid \mathrm{O}_x[4, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \mathtt{send\ y\ (eval\ 2)} \mid \mathrm{O}_y[3, x], \mathrm{O}_x[4, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o \langle \mathtt{send\ x\ (eval\ 6)} \mid \mathrm{O}_x[4, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \mathtt{send\ id_r\ (eval\ 24)} \mid \Theta \rangle \overset{\forall}{\leadsto}_o \langle \emptyset \mid \Theta' \rangle.
\end{gathered}
$$

The final state Θ' is obtained by storing 24 in the *receiver*-object. x and y are newly introduced names for the local *slave*-objects. A `factclass`-object can process many $fact$-messages at the same time since it demands the computation to its `slave`. For instance, let Θ be as before and add a new message $(\mathtt{fact\ 3\ id_r})$ to Ω. We obtain computation as the following one:

$$
\begin{gathered}
\langle \mathtt{send\ id_f\ (fact\ 4\ id_r)}, \mathtt{send\ id_f\ (fact\ 3\ id_r)} \mid \Theta \rangle \overset{\forall}{\leadsto}_o \\
\langle \mathtt{new\ \ slave\ (st\ 4\ id_r)\ x}, \mathtt{send\ id_f\ (fact\ 3\ x)}, \mathtt{send\ id_f\ (fact\ 3\ id_r)} \mid \Theta \rangle \overset{\forall *}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \mathtt{send\ id_f\ (fact\ 3\ x)}, \mathtt{new\ \ slave\ (st\ 3\ id_r)\ x'}, \mathtt{send\ id_f\ (fact\ 2\ x')} \mid \mathrm{o}_x[4, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \mathtt{new}\ \ slave\ (st\ 3\ x)\ y), \mathtt{send\ id_f\ (fact\ 2\ y)}, \mathtt{send\ id_f\ (fact\ 2\ x')} \mid \mathrm{o}_x[4, id_r], \mathrm{o}_{x'}[3, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \mathtt{send\ x'\ (eval\ 2)}, \mathtt{send\ y\ (eval\ 2)} \mid \mathrm{o}_y[3, x], \mathrm{o}_x[4, id_r], \mathrm{o}_{x'}[3, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o \\
\langle \mathtt{send\ id_r\ (eval\ 6)}, \mathtt{send\ x\ (eval\ 6)} \mid \mathrm{o}_x[4, id_r], \Theta \rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \mathtt{send\ id_r\ (eval\ 24)}, \mathtt{send\ id_r\ (eval\ 6)} \mid \Theta \rangle.
\end{gathered}
$$

Here, the identifiers x, y, x', y' are new names for the local objects of the two concurrent computations. Note that two messages are executed in interleaving. To conclude, we want to return on the problem of inheritance sketched at the beginning of this section.

6.7. *Inheritance at the Object Level*

Being in an object-based setting allows us to capture a very basic notion of inheritance. Let us extend the formalism above with two new *primitive* operations `extend` and `override` defined as follows:

```
extend Id NMs || object Id St Ms --> anti || object Id St (Ms & NMs).

override Id Ms OMs || object Id St Ms --> anti || object Id St OMs.
```

The meaning of the first clause should be clear: it extends the current set of methods with new ones. On the other hand, the second clause *overrides* the methods specified through the parameter `Ms` with the new definitions given in `OMs`.

Based on such definitions, it is now possible to define hierarchies of classes on a delegation-based mechanism. In the following example we assume that methods are paired with their name and kept into a list $(name_1, Def_1) :: \ldots :: (name_n, Def_n) :: nil$. It is not difficult to figure out a modification of the clause of Def. 6.1, in order to handle the new representation. We can rewrite the `rpoly` and `circle` classes of the example at the end of Section 6.1 as follows:

```
 new rpoly  (st Face Rank Perim) Id || anti -->
         anti || object Id (st Face Rank Perim) Ms.

where Ms is the list (set,M_set)::(get,M_get)::nil

 new circle (st Ray Perim) Id || anti -->
         new rpoly (st Ray infinity Perim) Id | override Id rms RMs || anti

      RMS=(set,M1)::(get,M2)::nil

      OMs=(set,M_set_circle)::(get,M2)::nil
```

where for the sake of brevity, we omit the definitions of `M_set`, `M_get` and `M_set_circle`, they can easily written by following the examples shown in the previous sections. Note that `M1, M2` are variables, in fact, we can distinguish the two methods by looking at the name, and this is enough to replace their definitions.

To summarize, in this section we have shown, through many examples, how to define a rich object-based concurrent calculus in higher order linear logic. We just mention, without entering into a detailed description, that other features such as *classes* as *special meta-objects* can also be derived. Note, finally, that the presented encoding integrates in a natural way logical features peculiar of linear logic (e.g., state-based computations) with features introduced in previous works in higher order logic programming (e.g., the use of universal quantification).

7. Conclusions

To conclude, we can compare our work to previous works relating logic and object-oriented programming.

7.1. *Related works*

The central point in the approach integrating OO and LP is the definition of a logical counterpart of objects, and, in particular, of objects with internal state. The different approaches and the different ways of representing objects are presented in (Brogi *et al.* 1991). We recall here some of them. In Login (Aït-Kaci and Nasr 1986), an extension of Prolog with order-sorted type structures, object are represented as *typed variables.* In (McCabe 1992), objects are viewed as logical *theories* and methods as clauses. Based on the sameidea, in (Miller 1989a), an extension of logic programming with embedded implication is used to define modules with hidden structures. In (Chen and Warren 1988), objects are viewed as non-logical variables, whose value can be modify by means of an assignment construct. In (Conery 1988), objects are viewed as atomic formulas. In this approach Horn Clauses are extended to multiple-headed clauses. The evolution of an object (an atomic formula) is determined by executing a clause, i.e., rewriting the object and the message into a new object. The semantics of this approach finds a natural counterpart in the linear logic setting, as we discussed in the course of the paper. In Concurrent Prolog, goals are considered as a collection of concurrent processes, each represented by an atomic formula, and shared variables are used as communication channels. In Vulcan (Kahn *et al.* 1987), a concurrent object-oriented language based on Concurrent Prolog program clauses are seen as rewriting rules. State updates are modeled by rewriting the atoms representing *state* components. This is the same idea used in Polka, an OO presentation of Parlog (Clark and Gregory 1986).

A similar approach has been followed in the specification of OO paradigms in the setting of term rewriting systems, as, for instance, in the Conditional Rewriting Logic (Meseguer 1990).

Natural extensions to the previous mentioned ideas have been developed in the setting of linear logic programming. As we discussed in the course of the paper, LO, the linear logic language proposed by Andreoli and Pareschi in (Andreoli and Pareschi 1991), is one of the most significant approaches in the field. LO extends in different ways previous

work relating traditional logic programming and object-orientation, e.g., (Conery 1988; McCabe 1992). Specifically, LO objects are aggregation of data fields (represented by disjunctions of atomic formulae), and LO classes are defined by collections of multiple-conclusion clauses synchronizing objects and messages. Here, a sequent represents a single object and a proof represents the evolution of a collection of objects (spread on different branches of a proof-tree). In our setting, objects are higher-order atomic formulae embedding their methods, defined, in turn, as multi-headed clauses. In addition, sequents represent collections of objects. Our view captures the essence of object-based languages, i.e., encapsulation and privacy of data (private methods and data fields), as well as dynamic overriding of methods. All these features were not captured by the LO object-model. From an abstract point-of-view, our approach shares similarities with the work in (Kobayashi and Yonezawa 1993). In that paper, the authors present a higher-order process calculus (i.e. where a process can be sent as a value to another process) modeled in classical linear logic. They have extended this idea in (Kobayashi and Yonezawa 1994b) to model a concurrrent object-based calculus and its type inference system. Differently from our approach, they modeled the basic computational mechanism of the calculus (i.e. method invocation) through the addition of a fixpoint operator to represent recursive object expressions. Object-expressions are unfolded at the moment of the method execution. In our approach, methods invocations are modeled by exploiting the combined power of the linear implication (to fire methods) and the higher-order syntax used in encoding objects (i.e. methods are viewed as terms or formulae depending on the context in which they occur).
To this regard, it is our opinion that our work gives a more complete view of the expressiveness of linear logic programming languages (and, in particular, of the uniform-proof based approach) and an extensive example on the use of all of its basic features. We believe that it is still worthwhile to further inspect the strength of based-on linear logic formalisms in order to find stronger and stronger connections with computer science. New important contributions in this sense could be given by research on non-commutative logic, in order to have sequentiality as a feature of the logic itself, logic of definitions to analyse at the meta-level significative properties of the object-logic (Miller 1993; Miller 1994).

Acknowledgments. The authors would like to thank the anonymous referees for their fruitful comments and recommandations.

References

M. Abadi and L. Cardelli. A Theory of Primitive Objects. In *Proceedings of Theoretical Aspect of Computer Software*, volume 789 of *Lecture Notes in Computer Science*, pages 296–320. Springer-Verlag, 1994.

S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111:3–57, 1993.

G. Agha. Actor: A model of Concurrent Computation in Distributed Systems, 1986. The MIT Press, Cambridge, MA.

H. Aït-Kaci and R. Nasr. LOGIN: A logic programming language with built-in inheritance. *Journal of Logic Programming*, 3(3):185–215, October 1986.

V. Alexiev. Applications of Linear Logic to Computation: An Overview. *Bulletin of the IGLP*, 2(1):77–107, 1994.

J.M. Andreoli. Logic Programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

J.M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-In Inheritance. *New Generation Computing*, 9:445–473, 1991.

A. Brogi, E. Lamma, and P. Mello. Objects in a Logic Programming Framework. In A. Voronkov, editor, *Proceedings of the 1st Russian Conference on Logic Programming*, volume 592 of *Lecture Notes in Computer Science*, pages 102–113. Springer-Verlag, Berlin, 1991.

W. Chen and D. H. Warren. Objects as Intensions. In R. A. Kowalski and K. A. Bowen, editors, *Proceedins of 5th International Conference on Logic Programming*, pages 404–419. The MIT Press, Cambridge, MA, 1988.

J. Chirimar. *Proof Theoretic Approach to Specification Languages.* PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.

K.L. Clark and S. Gregory. PARLOG: Parallel Programming in Logic. *ACM TOPLAS*, 8(1):1–49, 1986.

J.S. Conery. Logical objects. In R.A. Kowalski and K.A. Bowen, editors, *Proceedins of 5th International Conference on Logic Programming*, pages 420–434. The MIT Press, Cambridge, MA, 1988.

G. Delzanno. *Logic & Object-Oriented Programming in Linear Logic.* PhD thesis, Università of Pisa, Dipartimento di Informatica, March 1997.

G. Delzanno and M. Martelli. Objects in Forum. In *Proceedings of the International Logic Programming Symposium*, pages 115–129. The MIT Press, Cambridge, MA, 1995.

G. Delzanno and M. Martelli. Proofs as Computations in Linear Logic. Technical Report DISI-TR-98-12, DISI, Dipartimento di Informatica, Università di Genova, 1998.

K. Fisher, F. Honsell, and J. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

D. Galmiche. Canonical proofs for linear logic programming frameworks. In *Workshop on Proof-theoretical Extensions of Logic Programming*, Santa Margherita Ligure, Italy, June 1994.

D. Galmiche and E. Boudinet. Proofs, concurrent objects and computations in a FILL framework. In *Workshop on Object-based Parallel and Distributed Computation, OBPDC'95, LNCS 1107*, pages 148–167, Tokyo, Japan, 1996.

D. Galmiche and G. Perrier. Foundations of proof search strategies design in linear logic. In *Logic at St Petersburg '94, Symposium on Logical Foundations of Computer Science, LNCS 813*, pages 101–113, St Petersburg, Russia, July 1994.

D. Galmiche and G. Perrier. On proof normalization in linear logic. *Theoretical Computer Science*, 135(1):67–110, 1994.

J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

A. Guglielmi. *Abstract Logic Programming in Linear Logic - Independence and Causality in a First Order Calculus.* PhD thesis, Department of Computer Science, University of Pisa, 1995.

J. Harland. A Proof-theoretic Analysis of Goal-directed Provability. *Journal of Logic and Computation*, 4(1):69–88, 1994.

C.A.R. Hoare. Communicating Sequential Processes. *CACM*, 21(8):666–677, 1978.

J. Hodas. *Logic Programming in Intuitionistic Linear Logic.* PhD thesis, University of Pennsylvania, Department of Computer and Informaton Science, 1994.

J. Hodas and D. Miller. Representing Objects in a Logic Programming Language with Scoping Constructs. In D. H. Warren and P. Szeredi, editors, *Proceedins of 7th International Conference on Logic Programming*, pages 511–526. The MIT Press, Cambridge, MA, 1990.

J. Hodas and D. Miller. Logic Programming in a Fragment of Intuitionistic Linear Logic. *Information and Computation*, 110(2):327–365, 1994.

J. Hodas and J. Polakow. Forum as a Logic Programming Language (Preliminary Report). *Electronic Notes in Theoretical Computer Science*, 3, 1996.

M. Hyland and V. de Paiva. Full Intuitionistic Linear Logic (extended abstract). *Annals of Pure and Applied Logic*, 64:273–291, 1993.

K. Kahn, E. Tribble, M. Miller, and D. Bobrow. Vulcan: Logical concurrent objects. In P. Wegner B. Shriver, editor, *Research Directions in Object-Oriented Programming.* The MIT Press, Cambridge, MA, 1987.

N. Kobayashi and A. Yonezawa. ACL - a concurrent linear logic programming paradigm. In *Int. Symposium on Logic Programming*, pages 279–294, Vancouver, October 1993.

N. Kobayashi and A. Yonezawa. Asynchronous Communication Model based on Linear Logic. *Formal Aspects of Computing*, 3:279–294, 1994.

N. Kobayashi and A. Yonezawa. Higher-order concurrent linear logic programming. In *Proceedings of Theory and Practice of Parallel Programming (TPPP'94).* Lecture Notes in Computer Science, 1994.

Higher-order, linear, concurrent constraint programming. P. Lincoln and V. Saraswat. Unpublished manuscript, available at the URL file://parcftp.xerox.com/pub/ccp/lcc/hlcc.dvi. Xerox Co. January, 1993.

P. Lincoln and N. Shankar. Proof search in first-order linear logic and other cut-free sequent calculi. In *9th IEEE Symposium on Logic in Computer Science*, pages 282–291, Paris, France, July 1994.

F. G. McCabe. *Logic and Objects.* International Series in Computer Science. Prentice Hall, 1992.

J. Meseguer. A logical theory of concurrent objects. In *OOPSLA-ECOOP '90*, pages 101–115, Ottawa, October 1990. Sigplan Notices, (25), 10.

J. Meseguer and N. Marti-Oliet. From Petri nets to Linear Logic. *Math. Struct. in Comp. Science*, 1:69–101, 1991.

D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6:79–108, 1989.

D. Miller. Lexical Scoping as Universal Quantification. In *Proceedins of 6th International Conference on Logic Programming*, pages 268–283. The MIT Press, Cambridge, MA, 1989.

D. Miller. The π-calculus as a theory in linear logic: Preliminary results. In E. Lamma and P.Mello, editors, *Proceedings of the 1992 Workshop on Extension to Logic Programming*, volume 660 of *Lecture Notes in Computer Science*, pages 242–265. Springer-Verlag, Berlin, 1993.

D. Miller. A multiple-conclusion meta-logic. In *9th IEEE Symposium on Logic in Computer Science*, pages 272–281, Paris, France, July 1994.

D. Miller. Forum: A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform Proofs as a Foundation for Logic Programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

G. Nadathur and D. Miller. An Overview of λ-Prolog. *Fifth International Symposium on Logic Programming*, 1988.

G. Perrier. A model of concurrency based on linear logic. In *Proceedings of the Conference on*

Computer Science Logic 95, 1995.

D. Pym and J. Harland. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, 1994.

H. Schellinx. Some syntactical observations on linear logic. *Journal of Logic and Computation*, 1(4):537–559, 1991.

E.Y. Shapiro. The family of concurrent logic programming languages. *Computing Surveys*, 21(3):413–510, 1989.

A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA '86*, pages 38–44, 1986.

T. Tammet. Proof Strategies in Linear Logic. *Journal of Automated Reasoning*, 12:273–304, 1994.

L.A. Wallen. *Automated Proof search in Non-Classical Logics.* MIT Press, 1990.

P. Wegner. Dimension of object-based language design. In *Proceedings of OOPSLA '87*, pages 168–182, 1987.

Appendix A. Proof Systems for Linear Logic

The language considered here consists of a set of *terms* over a countable set of variables, a countable set of *atoms* and the following set of formulas built over the *logical operators* $\{0, 1, \bot, \top, ()^{\bot}, !, ?, \otimes, \parr, \&, \oplus, \forall, \exists\}$:

$$F ::= \mathbf{0}|\mathbf{1}| \bot |\top|a|a^{\bot}|?F|!F|F \otimes F|F \parr F|F\&F|F \oplus F|\forall xF|\exists xF.$$

The negation of formula is defined by the following equalities:
$A^{\bot\bot} = A$, $\mathbf{1}^{\bot} = \bot$, $\bot^{\bot} = \mathbf{1}$, $\top^{\bot} = \mathbf{0}$, $\mathbf{0}^{\bot} = \top$, $(A \otimes B)^{\bot} = A^{\bot} \parr B^{\bot}$ and $(A \parr B)^{\bot} = A^{\bot} \otimes B^{\bot}$, $(A\&B)^{\bot} = A^{\bot} \oplus B^{\bot}$ and $(A \oplus B)^{\bot} = A^{\bot}\&B^{\bot}$, $(\forall xA)^{\bot} = \exists xA^{\bot}$ and $(\exists xA)^{\bot} = \forall xA^{\bot}$, $(!A)^{\bot} = ?A^{\bot}$ and $(?A)^{\bot} = !A^{\bot}$. In the following we present two formulations of linear logic: a one-sided sequent calculus and a two-sided sequent calculus. The laws of negation can be used to pass from one to the other.

$$\frac{}{\vdash A, A^{\bot}}\ Id \qquad \frac{\vdash A, \Gamma \quad \vdash A^{\bot}, \Delta}{\vdash \Gamma, \Delta}\ Cut$$

$$\frac{\vdash \Delta}{\vdash ?A, \Delta}\ w? \qquad \frac{\vdash ?A, ?A, \Delta}{\vdash ?A, \Delta}\ c?$$

$$\frac{\vdash A, \Gamma_1 \quad \vdash B, \Gamma_2}{\vdash A \otimes B, \Gamma_1, \Gamma_2}\ \otimes \qquad \frac{\vdash A, B, \Gamma}{\vdash A \parr B, \Gamma}\ \parr \qquad \frac{}{\vdash \mathbf{1}}\ \mathbf{1} \qquad \frac{\vdash \Gamma}{\vdash \bot, \Gamma}\ \bot$$

$$\frac{\vdash A, \Gamma \quad \vdash B, \Gamma}{\vdash A\&B, \Gamma}\ \& \qquad \frac{\vdash A, \Gamma}{\vdash A \oplus B, \Gamma}\ \oplus_1 \qquad \frac{\vdash B, \Gamma}{\vdash A \oplus B, \Gamma}\ \oplus_2 \qquad \frac{}{\vdash \top, \Delta}\ \top$$

$$\frac{\vdash A, ?\Gamma}{\vdash !A, ?\Gamma}\ ! \qquad \frac{\vdash A, \Gamma}{\vdash ?A, \Gamma}\ ? \qquad \frac{\vdash A[y/x], \Gamma}{\vdash \forall xA, \Gamma}\ \forall \qquad \frac{\vdash A[t/x], \Gamma}{\vdash \exists xA, \Gamma}\ \exists$$

In $\forall$ rule, y is not free in Γ and in A if y is different of x.

Fig. 8. The one-sided LL sequent calculus

The inference system here is the classical linear sequent calculus with two-sided sequent $\Gamma \vdash \Delta$ where Γ and Δ are multisets of linear formulae, i.e, formulae built on the linear primitives $0, 1, \bot, \top, ()^{\bot}, \otimes, \parr, \&, \oplus, \forall, \exists, ?, !$.
For a complete presentation of this system we refer to (Girard 1987).

$$\frac{}{A \vdash A}\ id \qquad \frac{\Gamma \vdash A, \Delta \quad A, \Lambda \vdash \Theta}{\Gamma, \Lambda \vdash \Delta, \Theta}\ cut$$

$$\frac{\Gamma \vdash \Delta}{!A, \Gamma \vdash \Delta}\ !w_L \quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, ?A}\ ?w_R \quad \frac{!A, !A, \Gamma \vdash \Delta}{!A, \Gamma \vdash \Delta}\ !c_L \quad \frac{\Gamma \vdash \Delta, ?A, ?A}{\Gamma \vdash \Delta, ?A}\ ?c_R$$

$$\frac{A, B, \Gamma \vdash \Delta}{A \otimes B, \Gamma \vdash \Delta}\ \otimes_L \quad \frac{\Gamma \vdash \Delta, A \quad \Lambda \vdash \Theta, B}{\Gamma, \Lambda \vdash \Delta, \Theta, A \otimes B}\ \otimes_R \quad \frac{}{\vdash \mathbf{1}}\ 1_R \quad \frac{\Gamma \vdash \Delta}{\mathbf{1}, \Gamma \vdash \Delta}\ 1_L$$

$$\frac{A, \Gamma \vdash \Delta \quad B, \Lambda \vdash \Theta}{A \parr B, \Gamma, \Lambda \vdash \Delta, \Theta}\ \parr_L \quad \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \parr B}\ \parr_R \quad \frac{}{\bot \vdash}\ \bot_L \quad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \bot}\ \bot_R$$

$$\frac{\Gamma \vdash \Delta, A \quad B, \Lambda \vdash \Theta}{A \multimap B, \Gamma, \Lambda \vdash \Delta, \Theta}\ \multimap_L \quad \frac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \multimap B}\ \multimap_R$$

$$\frac{\Gamma \vdash \Delta, A}{A^\bot, \Gamma \vdash \Delta}\ \bot_L \quad \frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, A^\bot}\ \bot_R$$

$$\frac{}{\Gamma \vdash \Delta, \top}\ \top_R \quad \frac{}{\mathbf{0}, \Gamma \vdash \Delta}\ 0_L \quad \frac{A, \Gamma \vdash \Delta}{A \& B, \Gamma \vdash \Delta}\ \&_L \quad \frac{B, \Gamma \vdash \Delta}{A \& B, \Gamma \vdash \Delta}\ \&_L$$

$$\frac{\Gamma \vdash \Delta, A \quad \Gamma \vdash \Delta B}{\Gamma \vdash \Delta, A \& B}\ \&_R \quad \frac{A, \Gamma \vdash \Delta \quad B, \Gamma \vdash \Delta}{A \oplus B, \Gamma \vdash \Delta}\ \oplus_L \quad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \oplus B}\ \oplus_R \quad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \oplus B}\ \oplus_R$$

$$\frac{A, \Gamma \vdash \Delta}{!A, \Gamma \vdash \Delta}\ !_L \quad \frac{!\Gamma \vdash A, ?\Delta}{!\Gamma \vdash !A, ?\Delta}\ !_R \quad \frac{!\Gamma, A \vdash ?\Delta}{!\Gamma, ?A \vdash ?\Delta}\ ?_L \quad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, ?A}\ ?_R$$

$$\frac{A[t/x], \Gamma \vdash \Delta}{\forall x A, \Gamma \vdash \Delta}\ \forall_L \quad \frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, \forall x A}\ \forall_R \quad \frac{A, \Gamma \vdash \Delta}{\exists x A, \Gamma \vdash \Delta}\ \exists_L \quad \frac{\Gamma \vdash \Delta, A[t/x]}{\Gamma \vdash \Delta, \exists x A}\ \exists_R$$

Fig. 9. The two-sided LL sequent calculus

Appendix B. Permutability for the Linear Connectives

$t_2 \backslash t_1$	$\multimap_L$	$\otimes_R$	$\multimap_R$	$!_R$	$\&_R$	$\&_L$	$\oplus_R$	$!_L$	$w!_L$	$c!_L$	$\forall_R$	$\forall_L$	$\exists_R$
$\multimap_L$	p	p	p	p	p	p	p	p	p	p	p	p	p
$\otimes_R$	p	np	np	np	np	p	np	p	p	p	np	p	np
$\multimap_R$	np	np	np	np	np	p	np	p	p	p	np	p	np
$!_R$	np	np	np	np	np	np	np	np	p	p	np	np	np
$\&_R$	np	np	np	np	np	np	np	np	np	np	np	np	np
$\&_L$	p	p	p	p	p	p	p	p	p	p	p	p	p
$\oplus_R$	p	np	np	np	np	p	np	p	p	p	np	p	np
$!_L$	p	p	p	p	p	p	p	p	p	p	p	p	p
$w!_L$	p	p	p	p	p	p	p	p	p	p	p	p	p
$c!_L$	np	np	p	p	p	p	p	p	p	p	p	p	p
$\forall_R$	p	np	np	np	np	p	np	p	p	p	np	np	np
$\forall_L$	p	p	p	p	p	p	p	p	p	p	p	p	p
$\exists_R$	p	np	np	np	np	p	np	p	p	p	np	p	np

Fig. 10. Permutability properties for HM system