Proof Theory in Linear Logic: A Specification of Concurrent Objects (Draft)

Giorgio Delzanno
Dipartimento di Informatica e Scienze dell'Informazione, Università di Genova,
Via Dodecaneso, 35, I-16146 Genova, Italy,
phone:+39-10-3536638
email:giorgio@disi.unige.it

August 30, 1997

1 Introduction

Linear Logic, introduced by Girard [35] can be considered as a refinement of Classical Logic in that it introduces the notion of *resource* or, better, of *formulas* as *resources*. This notion results by considering a subset of the structural rules for Classical sequent calculus which does not contain *weakening* and *contraction*.

To evaluate the impact of Girard's intuition on the theoretical research in Computer Science, it is enough to look at the connections between Linear Logic and other frameworks and computation models such as the Petri nets [25, 34], the Chemical Abstract machine [2],the π-calculus [13], functional paradigms [50].

The Curry-Howard isomorphism, where formulas are interpreted as types, proofs as terms (programs) and cut-elimination as λ-conversion (computation), has been extended to this new setting yielding definitions of more complex (and more refined) term-calculi, e.g., [2, 51]. Proof theoretic investigations allowed to study, as in the Classical case, both Intuitionistic and Classical formulations of Linear Logic [68]; results were obtained on cut-elimination with different fragments of connectives and on a systematic study of the permutability of their rules [9, 33, 51].

Proof search paradigms, based on the interpretation of formulas as programs and proofs as computations, have been investigated with a particular attention to the impact of LL in the definition of Logic Programming languages as shown, for instance, in [9, 39, 42, 60].

1.1 Proofs as computations in LL: an overview

The proofs as computations paradigm has been investigated from many different point-of-views. The common denominator of these approaches can be found in the methodology followed to analyze the underlying logic system. Here we can distinguish three main steps which can be instantiated to a specific application:

- Selecting a fragment of Logic capable of expressing properties sensible to our aims;
- Isolating a cut-free subsystem complete w.r.t. the whole logic;
- Refinining the notion of proofs and sequents in order to interpret them as description of computation in the target computational model.

As described below the main objectives of this approach are: the development of new logic programming languages; the study of linear logic as a logical model for concurrency (the reader can refer to [7, 61] to have an extensive bibliography and survey on the topic).

1.1.1 Concurrency

Among the seminal works in the field it is possible to find Kobayashi's and Yonezawa's ACL [48]

The former is a presentation of Classical Linear Logic in terms of a model for concurrency. It is based on a one-sided sequent calculus, i.e., sequents have the form $\longrightarrow \Delta$, where Δ is a multi-set of linear formulas describing a collection of processes. A proof corresponds to the description of the evolution of such processes. Thus, the semantics of the operator of the concurrent language is given in terms of the semantics of a corresponding logical connective: for instance, the parallel composition is mapped into $⅋$ and message passing is implemented using $\otimes$. ACL and its corresponding Higher-Order formulation, HACL [49], provide remarkable intuitions behind the study of concurrent logic languages in LL.

A similar approach has been considered in CPL, [67], and FILL, [32], in which two-sided sequents are considered in the context of a single- and a multiple-conclusion formulation of Intuitionistic Linear Logic respectively. Here a sequent $\Gamma \longrightarrow \Delta$ represents a transition of the processes defined in Γ to the final configuration described in Δ (a singleton in CPL, a multi-set in FILL).

Many aspects of these work can be rephrased in presentations of LL closer to the logic programming perspective, as shown in the description of some aspects of π-calculus in Linear Logic in [58], using transformations guided by LL equivalences.

1.1.2 LP languages

The language LO [10] is based on a simple fragment of Linear Logic consisting of $\&$, $⅋$, $\forall$, $\top$. Thus, it is suitable to encode a process interpretation of Logic Programs in which $\&$-nodes in a proof search tree correspond to split a computation into two independent branches (global concurrency) and $⅋$-nodes to a local notion of concurrency between components of a single process. Under this perspective, a sequent can represent the state of a single process (or object as shown in [12]). LO formulas are defined as linear implications with multiple-head: $A_1 ⅋ \ldots ⅋ A_n \circ\!- G$ where G allows occurrences of $\&$ and $⅋$. Some problems arise when trying to model communication between processes in such a setting which, according to the previous point of view, requires information to flow between different branches of a proof, see [11]. For the same reason it is difficult to take snapshots of the global state of the system simulated by the LO-programs.

LinLog [9], which contains LO as a sub-language, is a (one-sided sequent calculus) presentation of Linear Logic in terms of *focusing proofs*. The form of a focusing proof forces the application of asynchronous rules (e.g., rules for $⅋$, $\&$) before the application of synchronous ones (e.g., $\otimes$, $\oplus$). LinLog provides the expressiveness of full Linear Logic, thus, many problems arise in its use as a programming language (in particular in a higher-order setting). For instance, the non-determinism in the construction of a proof is due here to: choice of clauses ($\&$ on the left), splitting of context due to the use of $\otimes$ (on the right) or $\multimap$ (left) and higher-order unifications.

Other approaches reveal the same trade off between expressivity and complexity of the proof search procedure. Lygon [39], for example, is focused on the definition of a resolution rule for LL. However, the structure of the considered formulas is quite complex allowing occurrences of $⅋$, $\otimes$, $\&$, $\oplus$ in goals and of $\otimes$, $⅋$ and $\&$ in clause definitions and, thus, affecting the usability of the language. Actually, in the examples in [76], particular subclasses of formulas are considered reducing to other languages like LinLog or Lolli (introduced below).

Lolli [44] seems to provide the right balance between expressivity of the underlying fragment of Intuitionistic Linear Logic ($\forall$, $\&$, $\multimap$, $\Rightarrow$) and usability as a programming language. This language, based on first-order intuitionistic LL, provides clauses of the form $A \multimap B$ and $A \Rightarrow B$ and goals of the form $G\&G'$, $G \multimap G$ and $G \Rightarrow G'$. Notice that B can itself be an implication, thus, $\otimes$ can be used in the goals.

In particular, Lolli provides all the means to handle resources stored as facts in the program: sequents are of the form $!\Gamma, \Delta \rightarrow A$ where Δ is a multi-set of atomic formulas. Right occurrences of $\otimes$ allow to split Δ, of & to copy it, of $\Rightarrow$ to depart auxiliary computations with Δ forced to be empty.

An intelligent management of resources has been studied in order to define an implementation of the language (the I/O model in [44]). Lolli does not provide what Andreoli and Pareschi called *local* parallelism, i.e., the connective $\parr$ cannot occur in a goal.

On the other hand multi-conclusion sequent-calculi, as shown in [48, 31, 59, 67], allow to model concurrency apects by exploiting multi-set rewriting (see also SMR [37]).

In the following we will study the connections between the approaches based on Intuitionistic Linear Logic and the ones based on Classical Linear Logic, showing different criteria suitable for developing well-founded proof as computation paradigms in both systems. As representative of the second class we have chosen Forum.

1.2 Forum: specifications in Linear Logic

Forum [60] can be considered as the point of convergence of Miller's previous works on both Higher-Order and Linear Logic languages. This language is the result of the analysis of LL in terms of an extended notion of Uniformity coping with multi-conclusion sequents. This framework is a *specification* language in which it is possible to define the semantics of many aspects of Programming Languages. The language is based on a fragment of linear logic with the following set of connectives: $\multimap$, $\Rightarrow$, &, $\parr$ the neutral element of $\parr$, *anti* ($\perp$), and the neutral element of &, *all* ($\top$), the universal quantification over variables of type $\forall$. For the sake of simplicity, we will omit the type in the quantification if it is clear from the context.

The intuitionistic implication is not a proper linear connective but it is defined by the following logical equivalence (i.e. provable in linear logic) $(A \Rightarrow B) \equiv (!A \multimap B)$.

Uniformity of proofs is encoded in the proof system, since the left introduction rules are applicable if and only if the right-hand side of a sequent consists of atomic formulas. According to this definition of uniformity, Forum can be considered an ALPL formulation of LL. In [62] Miller has proved the equivalence of Forum with LinLog [9] and thus with full Higher-Order Linear Logic. The previous cited result of completeness between Forum and higher-order Linear Logic suggests to consider Forum as a proof theoretical foundation for Logic programming languages defined over particular classes of formulas, as in the case of First Order Logic for Horn Clauses, and Intuitionistic Logic for hereditary Harrop formulas. In our work [21] we have tried to isolate a particular subset of Forum enjoying the good properties that the subset of higher-order hereditary Harrop formulas does [63]. We called the considered class of formulas $\mathcal{E}_{hhf}$ for "extended hhf". Restricting the class of formulas (providing formulas corresponding to program clauses and to goal formulas as in standard logic programming) allows to consider a subclass of proofs which enjoys good properties from an operational perspective. For instance, in the selected language we restrict non-determinism to clause selection and instantiation (unification) as in classical logic programming. Even if this could seem a strong restriction, being in a multi-conclusion setting (and thus enjoying multi set rewriting) provides a great number of different interpretations of sequents and proofs, suitable to encode many different kinds of applications.

2 Towards Executable Specifications

To avoid the high level of non-determinism of proof searching in Forum it is necessary to consider a subclass of Forum formulas, in particular, rsstricting the occurences of $\multimap$ in program clauses. In the following we will consider the class of Linear Logic formulas $\mathcal{E}_{hhf}$ defined in [21] and inspired by the class of hereditary Harrop formulas of Intuitionistic Logic. This study lies at the basis of an experimental specification system discussed later.

Terms and Formulas. In order to define the terms of our language, let us fix a signature Σ of non-logical symbols, and a set of atomic predicates *constructors* denoted by $\Sigma_{\mathcal{R}} \subseteq \Sigma$ ($\Pi_{\Sigma_{\mathcal{R}}}$ will denote the corresponding set of atoms). The set of considered formulas is defined over simply typed λterms to provide a higher-order syntax. Hence terms are given by constant symbol, variables, applications $(t_1\ t_2)$, also written as $f\ t_1\ \ldots\ t_n$ when f is an n-ary constant, and abstractions $\lambda x.t$. Formulas are the λterms with special type o generated by the following grammar:

$$
\begin{array}{lcl}
\mathcal{D} & ::= & \mathcal{D} \,\&\, \mathcal{D} \mid \forall x.\mathcal{D} \mid \mathcal{H} \mathrel{\circ\!-} \mathcal{G} \mid \mathcal{G} \Rightarrow \mathcal{D} \mid \mathcal{H}. \\
\mathcal{G} & ::= & \mathcal{G} \,\&\, \mathcal{G} \mid \mathcal{G} \parr \mathcal{G} \mid \forall x.\mathcal{G} \mid \mathcal{D} \multimap \mathcal{G} \mid \mathcal{D} \Rightarrow \mathcal{G} \mid A \mid \bot \mid \top. \\
\mathcal{H} & ::= & \mathcal{H} \parr \mathcal{H} \mid A_r. \\
\mathcal{R} & ::= & \mathcal{R} \,\&\, \mathcal{R} \mid \mathcal{R} \parr \mathcal{R} \mid \forall x.\mathcal{R} \mid \bot \mid A_r \in \Pi_{\Sigma_{\mathcal{R}}}.
\end{array}
$$

Here A_r stands for a generic rigid atomic formula (i.e., the top level predicate symbol is a constant). Furthermore, we state the following restriction over $\mathcal{D}$-formulas.

Definition 2.1 ($\mathcal{D}$- $\mathcal{G}$ and $\mathcal{R}$-formulas) *The set of $\mathcal{D}$, $\mathcal{G}$ and $\mathcal{R}$-formulas results by considering all the closed formula generated by the previous grammar, with the further constraint that in each $\mathcal{D}$-formula having form $A_1 \parr \ldots \parr A_n$ there exists i s.t. $A_i \notin \Pi_{\Sigma_{\mathcal{R}}}$.*

We consider here a variation of Forum sequents [60], in fact we introduce a further partition of the right hand side: $\Gamma; \Delta \longrightarrow_{\Sigma} \Omega \parallel \Theta$ where Γ is a set of $\mathcal{D}$- and $\mathcal{R}$-formulas, Δ is a multi-set of $\mathcal{D}$-formulas; Ω is a multi-set of $\mathcal{G}$-formulas, and Θ is a multi set of atomic $\mathcal{R}$-formulas. Σ contains all the non-logical symbols occurring in the sequent. As shown in the previous formulation, the partition of the right-hand side is just a convention to isolate formulas belonging to $\Pi_{\Sigma_{\mathcal{R}}}$. Intuitively, $\mathcal{D}$-formulas correspond to program clauses, $\mathcal{G}$-formulas to goal-formulas, atomic $\mathcal{R}$-formulas to data representations using constructors defined in $\Sigma_{\mathcal{R}}$ and compound $\mathcal{R}$-formulas to assertions on state configurations. The Forum proof system can be specialized to this set of formulas and enriched with a backchaining rule over multiple-headed clauses. As a consequence we have the following right-introduction rules:

Search Rules

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

$$
\frac{}{\Gamma; \Delta \longrightarrow_{\Sigma} \top, \Omega \parallel \Theta}\ \top_r
\qquad
\frac{\Gamma; \Delta \longrightarrow_{\Sigma} \Omega \parallel \Theta}{\Gamma; \Delta \longrightarrow_{\Sigma} \bot, \Omega \parallel \Theta}\ \bot_r
\qquad
\frac{\Gamma; \Delta \longrightarrow_{y:\tau,\Sigma} A[y/x], \Omega \parallel \Theta}{\Gamma; \Delta \longrightarrow_{\Sigma} \forall_{\tau} x.A, \Omega \parallel \Theta}\ \forall_r\ (y \notin \Sigma)
$$

$$
\frac{\Gamma; \Delta \longrightarrow_{\Sigma} A_1, A_2, \Omega \parallel \Theta}{\Gamma; \Delta \longrightarrow_{\Sigma} A_1 \parr A_2, \Omega \parallel \Theta}\ \parr_r
\qquad
\frac{B, \Gamma; \Delta \longrightarrow_{\Sigma} A, \Omega \parallel \Theta}{\Gamma; \Delta \longrightarrow_{\Sigma} B \Rightarrow A, \Omega \parallel \Theta}\ \Rightarrow_r
\qquad
\frac{\Gamma; B, \Delta \longrightarrow_{\Sigma} A, \Omega \parallel \Theta}{\Gamma; \Delta \longrightarrow_{\Sigma} B \multimap A, \Omega \parallel \Theta}\ \multimap_r
$$

Furthermore, given a multi-set Δ of $\mathcal{D}$-clauses over a signature Σ, let us define $\langle \Delta \rangle$ as $\bigcup_{D \in \Delta} \langle D \rangle$, where $\langle D \rangle$ is the minimum set such that: $\langle D_1 \& D_2 \rangle = \langle D_1 \rangle \cup \langle D_2 \rangle$, $\langle \forall_{\tau} x.D \rangle = \bigcup_{t \in \mathcal{T}_{\tau}} \langle D[t/x] \rangle$, where $\mathcal{T}_{\tau}$ is the set of Σ-terms of type τ, and $\langle D \rangle = \{D\}$ otherwise. We will use such a definition in order to define a backchaining step over a $\mathcal{D}$-clause D. Let us consider $\mathcal{D}$-clauses occurring in the bounded context, then we have the following rules:

Backchaining (bc)

$$
\frac{\Gamma; \Lambda \longrightarrow_{\Sigma} \Omega \parallel \Xi}{\Gamma; \Delta, D \longrightarrow_{\Sigma} \Upsilon \parallel \Theta}
\qquad
\begin{array}{l}
H \mathrel{\circ\!-} B \in \langle D \rangle, \\
H = A_1 \parr \ldots \parr A_n, \\
\Omega \uplus \Theta \equiv ((\Upsilon \uplus \Theta) \setminus \{A_1, \ldots, A_n\}) \uplus \{B\}.
\end{array}
$$

Guard rule (gr)

$$
\frac{\Gamma; \emptyset \longrightarrow_{\Sigma} G \parallel \emptyset \qquad \Gamma; \Delta, E \longrightarrow_{\Sigma} \Upsilon \parallel \Theta}{\Gamma; \Delta, D \longrightarrow_{\Sigma} \Upsilon \parallel \Theta}
\qquad G \Rightarrow E \in \langle D \rangle.
$$

The rule bc represents the application of a multi-set rewriting step (whose behaviour is described by the linear implication). The rule gr is the usual rule for $\Rightarrow$ occurring in the left-hand side of sequents. Similar rules must be considered if D belongs to the unbounded context: in that case D is not removed from Γ. There are two other cases to consider: backchaining over $\mathcal{D}$-facts (i.e., in $\mathcal{H}$-form) and selection of $\mathcal{R}$-formulas from Γ as shown in the figure below:

$$\frac{\textbf{Initial}}{\Gamma; D \longrightarrow_{\Sigma} \Upsilon \parallel \Theta} \quad \begin{array}{l} A \in \langle D \rangle, \\ \{A\} \equiv \Upsilon \uplus \Theta. \end{array} \qquad \frac{\textbf{Verify}}{\Gamma; \emptyset \longrightarrow_{\Sigma} \emptyset \parallel \Theta} \quad \begin{array}{l} \textit{If there exists } \Phi \in \Gamma \textit{ s.t. } \Phi \textit{ is an} \\ \mathcal{R}\textit{-formula and } \Phi \textit{ matches } \Theta \textit{ .} \end{array}$$

Here Φ *matches* Θ stands for $\Phi \rightarrow \Theta$ is cut-free provable (i.e., the proof is a simple sequence of $\forall_l$, $⅋_l$, $\&_l$ applications). Both cases are an extension of the Forum initial axiom $\Gamma; A \rightarrow A$ (with A atomic) since $\mathcal{H}$ and $\mathcal{R}$-formulas do not contain any occurences of the two forms of implication.

Furthermore, notice that only one Φ-formula can be selected and applied in any branch of a proof; for this reason we will use the notation $\Gamma[\Phi]; \Delta \longrightarrow_{\Sigma} \Omega \parallel \Theta$ to isolate the (guessed) Φ-formula which will be applied as last step of a proof. Assuming similar hypothesis to those in [63] it is possible to prove the following theorem.

Theorem 2.2 (ALPL [21]) *The language $\mathcal{E}_{hhf}$ together with the provability relation $\vdash_e$, induced by the above described rules, is an Abstract Logic Programming Language according to the definition of uniformity for multi-conclusion sequent calculi given in [60].*

2.1 An interpreter for $\mathcal{E}_{hhf}$: notation

As mentioned in the previous section, in $\mathcal{E}_{hhf}$ it is possible to enrich linear logic programs à-la-Lolli with multi-set rewriting capabilities. By exploiting this aspect, we have developed an $\mathcal{E}_{hhf}$-prototype in λProlog [63] (a higher-order intuitionistic logic programming language) to write high-level executable specifications of concurrent systems. In the following we will resort to the notation adopted in this prototype in order to discuss examples of $\mathcal{E}_{hhf}$-specifications. In order to do this we will reformulate the proof-system in terms of an interpreter whose inferencial engine enjoys the λProlog one, i.e., use of universal quantification and intuitionistic implication in a goal, and higher-order unification. We will adopt a syntax slightly different from the one in the previous section, in order to be faithful with the code of the examples of our prototype. Thus, the syntax for writing $\mathcal{E}_{hhf}$-programs is given now by the signature in Fig. 1. Here λterms are written as `f t1 ... tn` for $(f\ t1\ \ldots\ tn)$ and `x \ t` for $\lambda x.t$. Then, the notation `pi x \ F` will denote the universally quantified formula $\forall x.F$. In this context $\mathcal{D}$-formulas are written:

$Cond \Rightarrow (Head \multimapinv Body)$ as `Cond >> Head || State --> Goal || NewState.`;
$Cond \Rightarrow (Head)$ as `Cond >> Ops || State.`;
$Head \multimapinv Body$ as `Ops || State --> Goal || NewState.`;
$(Head)$ as `Ops || State`.

Here `Ops`, `State`, and `NewState` are $⅋$-disjunction of atomic formulas written `anti`, `A` (an atomic formula), $\texttt{A}_1$ | ...| $\texttt{A}_\texttt{n}$; `State` is the multiset of state atoms of $Head$;s `Goal` is a $\mathcal{G}$-formula with the connectives `|`, `&`, `=>`, `-o` and `pi`, i.e., respectively, $⅋$, $\&$, $\Rightarrow$, $\multimap$, and $\forall$.

Notice tha Finally, identifiers with capital letters must be considered as universally quantified variables, and comments are prefixed by `%` .

In the following, enjoying the λProlog syntax of the $\mathcal{E}_{hhf}$-interpreter, $\mathcal{E}_{hhf}$-clauses can have a further format:

```
(Cond >> Head || State --> Body):-  Constraints},
```

where the unification `Contraints` are used to simplify the presentation of the $\mathcal{E}_{hhf}$-clause (which ends before the ":-" symbol).

```
module sig.
type     ´-->´  o -> o -> o.
infixr   ´-->´  5.  % standing for o-
type     ´>>´  o -> o -> o.
infixr   ´>>´  6.   % standing for =>
type     ´||´ o  -> o -> o.
infixr   ´||´ 7.    % standing for ´par´ (to distinguish state atoms)
type     ´|´  o -> o -> o.
infixr   ´|´  8.    % standing for ´par´
type     ´-o´  o -> o -> o.
infixr   ´-o´  9.
type   anti    o.
type   all     o.
```

Figure 1: Signature for $\mathcal{E}_{hhf}$ formulas: `=>`, `pi`, `&` are already defined in λProlog.

By definition of $\mathcal{E}_{hhf}$-formulas, the only source of non-determinism in building $\mathcal{E}_{hhf}$-proofs (i.e., describing computations) is given by the choice of clause to apply in a bc-step. Actually, passing to the implementation in λProlog, this step requires higher-order unification and selection of atoms from the current goals (a multiset of atomic formulas) matching the head of the clauses. Thus the bc-rule is much more complex with respect to bc-rules in Lolli and Prolog. Thus we can rephrase the proof-theoretical operational semantics in terms of a proof-search procedure, inspired by the λProlog interpreter, as follows. A goal corresponds here to the right-hand side of a sequent, i.e., `?- Goal || CurrentState`, a program `P` and a bounded context `C` correspond to a set and a multiset of $\mathcal{D}$-clauses, respectively, with the same conventions adopted in Fig. 1. Substitutions are defined as in [63] and they associate variables with λterms. In this setting there can exist many different unifiers which are not related each other.

FORALL. `?- (pi G) | Gs || CState.` succeeds in `P` and `C` if `?- (G d) | Gs || CState.` succeeds in `P` and `C` where `d` is a new constant.

WITH. `?- G & H | Gs || CState.` succeeds in `P` and `C` if `?- G | Gs || CState.` succeeds in `P` and `C` and `?- H | Gs || CState.` succeeds in `P` and `C`.

ANTI. `?- anti | Gs || CState.` succeeds in `P` and `C` if `?- Gs || CState.` succeeds in `P` and `C`.

LOLLI. `?- D -o G | Gs || CState.` succeeds in `P` and `C` if `?- G | Gs || CState.` succeeds in `P` and `C,D`.

INTUI. `?- D => G | Gs || CState.` succeeds in `P` and `C` succeeds if `?- G | Gs || CState.` succeeds in `P,D` and `C`.

BC. `?- AGs || CState.` succeeds with answer η in `P` and `C` if `?- (G´ | RG´) || (NState´ | RState´).` succeeds with answer ρ in `P` and `C´` where

- `cl= Cond >> Head || State --> G || NState.` belongs to `C` or `P` (`Cond=true` if it is not specified);
- `Head,State` unifies with a submultiset of `AGs,CState` with residue `RG,RState` and unifier θ;
- `?- Cond`θ `||.` succeeds with answer γ in `P` and $\emptyset$;
- `(G´| RG´) || (NState´ | RState´) = (G | RG) || (NState | RState)`$\theta\gamma$;
- `C´=(C \ cl)`$\theta\gamma$;
- finally, $\eta = \theta\gamma\rho$.

INITIAL. `?- AGs || CState.` succeeds with answer η n `P` and `C` iff

- `cl= Cond || Head || State.` belongs to `C={cl}` or `P`; `Head,State` unifies with `AGs,CState` with no residue and unifier θ;
- `?- Cond`θ `|| anti.` succeeds in `P` and `emptyset` with answer γ and $\eta = \theta\gamma$.

VERIFY. `?- anti || CState` suceeds with empty answer.

Here we have assumed the commutativity of `|`, i.e., of $⅋$, while always reducing the leftmost formula in a goal. Also, we have not specified the notion of *anwer* substitution for the rules that just inherit it from their premises (i.e., the search rules). Since higher-order unification does not provide for a notion of *most* general unifier, a further level of non-determinism is introduced by the choice of one of the possible unifiers in the **BC** and **INITIAL** rules. Notice that **BC** and bf INITIAL result from a synthesis of the bc and initial rule introduced in the previous section. This simplification is guided by the following observation over $\mathcal{D}$-clauses: since $(A \Leftarrow B) \Leftarrow C)$ is equivalent to $A \Leftarrow (B \,\&\, C)$ and in $\mathcal{D}$-clauses of the form $H \multimapinv B$, H is a disjunction of atomic formulas, it follows that each $\mathcal{D}$-formula can be reduced to a clause having form $G \Rightarrow (H \multimapinv G')$, where G is a & of $\mathcal{G}$-formulas.

2.2 Examples

A simple example is given by the following $\mathcal{E}_{hhf}$-specification of a automatic-booking system. Let us fix the following signature:

```
module spec.

type request A -> o.
type store    list A -> o.
type checkp  A -> list A -> o -> o.
type go   o.
type in   o.
type notin o.
```

The rules describing the behaviour of the system are given below:

```
checkp Pren Log notin >>
 request Pren || store Log --> anti  || store (Pren::Log) .

checkp Pren Log in  >>
 request Pren || store Log --> anti  || store Log .
```

The idea is that a request can be satisfied if a *log* is active in the current state and the request has not been satisfied yet, thus the first rule must be read as: *if Pren is notin Log then consume the request Pren and update the state Log into (Pren::Log).* The second clause treats the case in which the request cannot be satisfied. Recall that the linear logic interpretation of such clauses is like the following:

$\forall\, Pren. \forall\, Log.\ checkp\ Pren\ Log\ notin \Rightarrow ((request\ Pren)\ ⅋\ (store\ Log)) \multimapinv store\ (Pren :: Log))$. Thus we have introduced the separator `||` just to simplify the lay out of programs, distinguishing between state-atoms (i.e., consumed and added to Θ in $\mathcal{E}_{hhf}$-sequents) and goals.

The clauses defining `checkp` can be defined in the usual Prolog-style (or better λProlog style) since they are supposed to be executed in the condition-part of an $\mathcal{E}_{hhf}$-sequent. For intance, if `Pren` must not be present in the current log to be executed:

```
checkp Pren (Pren::Log) in.
checkp Pren  nil        notin.
checkp Pren (P::Log)    Flag :-
  not (Pren = P),
  checkp Pren Log Flag.
```

Executing the goal `?-request "X" | request "X" | request "y" | store nil` yields to the final state `store ("X"::"Y"::nil)` with empty answer.

Since this kind of specification allows to deal with multiset rewriting it is possible here to specify the behaviour of concurrent agents/objects/processes by considering rewriting rules as the following one:

```
request Pren | request Pren || anti --> request Pren || anti .
```

this allows to reduce two request of the same prenotation synchronizing them. The linear logic interpretation is:

$\forall\, Pren.\;\; (requestPren) \parr (request\ Pren)) \multimap\; request\ Pren$

Thus, | still corresponds to $\parr$, but it is used here to compose operations and state atoms. Based on this framework we shall see how to specify concurrent-object oriented system by exploiting linear resources and higher-order.

3 Concurrent Object-Oriented Programming

Logic and Object-Oriented languages are vehicles to achieve different forms of high-level programming. Logic programming allows to write executable specifications providing both a declarative and a procedural reading, while OO programming allows to incrementally build up complex data structures *hiding* the internal representation of data and the implementation of the operations over them.

The challenge is to find a way of enjoying these two different forms of high-level reasoning inside the same framework. Furthermore, a faithful representation of features typical of OO within a logic language can provide them with a clear semantics, induced by their logical reading. In this second part of the thesis we will show that Linear Logic and the interpretation of proofs fixed in the first part can be used as a Logical foundations of extensions of LP based on different form of object-orientations. In this Section we will briefly introduce the main aspects of Object-Orientation, presenting also a survey on integrations with Logic Programming. The approaches based on Linear Logic will be discussed in the next section. First, in the following section we will summarize the basic features of object-oriented programming, fixing the terminology used in the rest of the thesis.

3.1 Terminology

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 the 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 of modularly modify programs. Object-orientation provides with an interpretation of programs, which can ease both the task of users and developers. Language like SmallTalk [36], Modula 3 [38], Ada [73], Self [74], C++ [24], POOL [8], and Beta [66], are based on such programming paradigm. In this setting new concepts like **class** and **object**, [75], are added to the traditional programming ones. **Class-based** languages provide both the notion of class and object. A class corresponds to a data type describing which are the functionalities of a given part of a program, i.e., description of data and operations. The users should only access the external interface of the operations and of the data, i.e., the implementation of the operations and the structure of data are hidden inside the class (so that their modifications do not affect the way the users use the class). An object corresponds to a given instance of a class, i.e., it encapsulates, [71], both data and operations (**methods**) which handle them. Thus, objects provide an internal **state**, whose components (**instance variables**), can be changed by executing its 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). In some respects, the object-oriented paradigm is based on message-passing, replacing the usual notion of procedure or function call in imperative and functional programming [72]. Classes can be composed by defining hierarchies which allow to share code among different classes. A class which inherits from another class is a **subclass** of the class it inherits from. A class is a subclass of its **superclass**. There are many ways to define *inheritance* relationships, see again [71, 75]: a new class can be created which inherits all of the functionality and data of some other class (i.e. instances of the new class understand all the messages that its superclass understands); the new class can choose to respond to any given message in a different way than the superclass does and it can also define new methods; the new class can modify the data used to implement the superclass and it can add new ones. When a subclass responds to a message in a different way that its superclass does, the subclass is said to have **overridden** its superclass method. 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 an operations over its data to another objects).

3.2 Objects, processes and concurrency

So far we have not associated any specific model of computations to the notion of object. In fact, Object-Orientation can fit both in a sequential and in a concurrent setting. As outlined in the previous section an objects can be considered as independent entities which, as response to a method call, can execute operations over their data. 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 [41], a task as in Ada [73], a statement as in Occam [52], a goal as in Concurrent Prolog [69], objects can play different roles.

For instance, objects can be considered as *active* entities having own execution threads which are activated at the moment of creation. An active object can be considered as an evolute form of process. As alternative ways, it is possible to employ asynchronous communication, e.g., the Actor model [3, 40], or to add processes as an orthogonal concept to the object oriented dimension.

Merging concurrency and OO aspects are important issues in distributed programming, the granularity in the representation of the information can be defined according to an OO definition of data. In particular, the notion of active objects is suitable to naturally distribute the information in a multi-sited systems, using the objects as elementary pieces of information.

The flexibility revealed by the Object-Orientation notions, induced the development of a variety of Object-Oriented- versions of already existing programming languages. Here, we are interested in studying the relationships with Logic Programming.

3.3 Logic Programming and Object-Orientation

The central point in combining OO and LP lays in finding a logical counterpart to the notion of object, and, in particular, object with an internal state. In the following, we will present some of the approaches in the literature, see [14, 16] for a more complete survey, according to different ways of representing objects.

Objects as typed variables. Among the early works in this field we mention Login [4], (its extensions are LeFun [5] and LIFE [6]), an extension of Prolog with order-sorted type structures. Instead of using usual terms as data constructors, the authors introduced a sort of record terms (called ψ-terms), which express the *name* of the structure and its *features* (i.e. attributes). Typed variables can be treated as a special form of objects with a structure induced by the type. The language defined over this signature is equivalent to a constraint language over the type attributes. In this way switching to the constraints language, the authors used an instantiation of the general CLP schema to define the semantics of the entire language. Similar work can be found in Smolka's feature logics, [70].

Objects as Theories: methods as clauses. In [53] McCabe proposed a different way of integrating OO features in Logic Programming. Here, an object is viewed as a named Horn-Clause theory (i.e., Prolog-program), where its internal data structure is defined by a set of facts and its methods by a set of clause definitions. Actually, this interpretation is closer to a modular definition of logic programs. In order to support message passing the language allows to explicit address the name of objects in a goal (i.e., special *switching* goals $\mathsf{u{:}G}$ are considered here, where u is an object(theory) name and G a goal), to switch from an object to another during the computation. Following this idea a number of improvements have been proposed. Among these we mention the language SelfLog, [15], where programs are divided into collections of units organized into inheritance hierarchies, whose semantics is defined, by overriding. Thus, a unit u inherits only the predicate definitions in its super classes, which are not defined in u itself.

It is also important to mention, Contextual Logic Programming, [64, 65]. In this setting, the semantic of switching goals like $\mathsf{u{:}G}$ is more complex: intuitively, the module u is loaded into a stack (context) of units, defining the current program in which to prove G. By studying different ways of managing the context, different form of inheritance have been investigated.

Related to these works, we recall the approach in Intuitionistic Logic, e.g. [56], discussed in Section **??**, which allows to define modules with hidden structures, by using logical features (intuitionistic implication and universal quantification in goal formulas).

For further details, the reader can refer to the above mentioned survey in [16]. Notice that, due to the chosen interpretation of objects as programs, all the previous works must resort to extra-logical features, e.g. primitives to modify the program clauses during the execution (see again Section **??**) in order to implement a notion of object with an internal update able state. The interpretation of methods as logic program clauses has also been adopted in other approaches.

Objects as variables. In order to overcome this problem, in [17], Prolog was extended by considering a class of non-logical variables, whose value can be modify by invoking a sort of assignment construct. Dynamic predicates, i.e., predicates that can modify the state of a variable, have a particular semantics based on Kripke-like models. Their denotation is the set of possible state transitions induced by the modification of the corresponding non-logical variables. These special variables can be considered as a primitive notion of objects with state, and the assignment construct as a special method. This idea has been reconsidered in other approaches, as we will discuss in the sequel of this Section.

Objects as atomic goals. Another approach to model state updates has been shown in [19]. In this approach, close to the concurrent interpretation of Prolog as we will describe later, objects are denoted by particular atomic goals. In order to define methods for this representation of objects, Horn Clauses have been extended by considering heads with multiple heads, as in the stack class below:

```
class stack.
    stack(Id,[]).
    stack(Id,[X|L]) :- integer(X),stack(Id,L).
    empty(Id) ∧ stack(Id,[]) :- stack(Id,[]).
    push(X,Id) ∧ stack(Id,S) :- integer(X),stack(Id,[X| L]).
    pop(X,Id) ∧ stack(Id,[X|S]) :- stack(Id,S).
    top(X,Id) ∧ stack(Id,[X|S]) :- stack(Id,[X|S]).
```

The evolution of an object is determined by executing a clause, i.e., rewriting the object and the message into a new object. The semantics of this approach was a little bit confused, while, it finds a natural counterpart in the Linear Logic setting, as we will discuss in the next section.

Meta-programming approaches. Here, we can mention the experimental languages SPOOL [29], and POL [30]. In particular, in the latter, it is possible to declare hierarchies of classes, where each declaration consists of a set of clause, considered as the methods, as in the following example:

```
% person
person with X:is_aged(Y) :- askuser(age,X,Y).
person with X:diplomlevel(Y) :- askuser(diplomlevel,X,Y).
% researcher
researcher isa person.
researcher with X:topic(Y) :- X:in_team(Z),topics(Z,Y).
franz instance researcher.
```

where X represents the identifier on an object, self in usual object-oriented languages. In this approach the instances have a state that is represented by a set of facts, like is_aged.

There are some interesting aspects about the methods management. In particular the crucial point is how to fit backtracking into such a scheme. Indeed, let us suppose to invoke a method that fails, should we allow the system to backtrack over the class hierarchy? It is possible to find examples in which it is necessary (the method is defined only in the superclass) and example in which it is not (the method is overridden in the subclass). A possible solution is to consider deterministic methods, using Prolog *cut*, which prevent the call of other methods with the same name in the hierarchy: local backtracking. This problem does not arise if objects have own copy of the definitions for which the backtracking is allowed.

Concurrent Objects. 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. Vulcan [46] is *presentation* of Concurrent Prolog providing a syntax close to usual OO languages, for instance, providing an explicit constructs to change the attributes of objects. The semantics is given in terms of a mapping into Concurrent Prolog: as in the *object* as *atoms* approach, objects are represented by goal atoms(processes); updates are expressed by clauses representing methods. For instance, an example of Vulcan syntax is:

```
class(person,[Name,Knows]).
person::name(Name).
person::acquaintance(AName) :- Knows:name(AName).
person::putn(new Name).
person::putk(new Knows).
```

where, Name and Knows are attributes names, and class::def. is a method. A new instances is created by using the construct Class : make(val,id) , where val is a list of values associated to the attributes and id is the name of the creating instance; a sequence of messages can be sent to an object using the construct: **id** : m_1 : ... : m_n. The special operator **new** corresponds to an assignment operator. The possible translation of the method **putn** in Concurrent Prolog is:

```
person(id,[name(Name),knows(Knows)],[putn(Name1)|M]) :-
        person(id,[name(Name1),knows(Knows)],M).
```

Here, an object defined by an atom **person(id,attrs,calls)**, specifying its identifier, its attributes, and the messages that it can receive, respectively. Thus, an invocation like, id : putn(john), is mapped into a goal like: person(id,As,[putn(john)]), which is solved using the clause above. Similarly to the Vulcan case-study, an OO presentation of Parlog [18], Polka, has been developed in [20]. These examples share the common idea of considering program clauses as rewriting rules, and of modeling state updates by rewriting the atoms representing *state* components. This is the same perspective adopted in the specification of OO paradigms in the setting of term rewriting systems, for instance, the Conditional Rewriting Logic [54]. Finally, we recall the following other proposal: Distributed Logic Programming, DLP [23], another extension of Prolog with special constructs for dealing with non-logical variables, object creation and rendez-vous communication; MultiLog [47], and Orient84/K [45].

In the last year, interesting works based on Linear Logic, have been also proposed in order to deal with all these aspects by exploiting its richer set of logical features provided. We will discuss them in the following section.

4 Specification of Concurrent Objects

Based on the experience of pioneering languages in the field like the examples in [10, 48], Higher-Order Linear Logic seems a suitable tool to define languages integrating Object-Oriented and Logic Programming constructs. In this section we will describe different possible representations of objects and of their features, fitting them into the interpretation of Linear Logic adopted in the previous sections.

The object-oriented scheme adopted in [22] aimed at describing complex systems by giving a distinct specification of global and local knowledge. The local knowledge consists of the specification of the behavior of a single object which can be considered as an entity residing on a given site. On the other hand, the global knowledge consists of the specification of the interactions among objects, i.e., a meta-level specification. By a logical viewpoint, it seemed reasonable to consider the set of existing objects as the *state* denotation inside a $\mathcal{E}_{hhf}$-sequent. Thus, each object can be given an atomic representation providing the corresponding local information, i.e, the set of methods. Furthermore, meta-level statements about the objects interaction is explicitly encoded by logical assertions. In this way it is possible to give a logical account of a variety of object-oriented features and to define different languages by modifying the formulation of the meta-level rules. Higher-Order Linear Logic is essential here in order to naturally amalgamate the meta- and object-level while describing operational aspects proper of the object-oriented paradigm.

The main difficulty in formally describing the behavior of object-based languages is due to the self-application mechanism used to solve a method call. Many approaches are focused on capturing this aspect in very simple representation of classes and object. Among the foundational works in this field, it is interesting to briefly introduce the extensions of λ-calculus with objects, [1, 27, 28, 55]. For instance, in [1] objects are represented by a λ-term having form $\lambda self.e$, where e is a record consisting of a list of method definitions. The body of methods may contain occurrences of $self$, as, for instance, in the case of the identity method $e = [id = self]$. An object is now obtained as **fix-point** of such a definition: $\mathbf{Y}(\lambda self.e)$ (where $\mathbf{Y}$ is the fixpoint operator). Intuitively, the behavior of an object, as response to a given message call, is obtained by *unfolding* its class definition (w.r.t. to $self$) and applying consequently the method corresponding to the invocation. Another possible approach [26] is based on an object calculus similar to the above mentioned one, with the main difference that $self$ is abstracted out inside each method. Thus, an object is defined here directly by a record of method definitions of the form: $m = e$ where $e = \lambda self.e'$. A method invocation is performed directly over an object structure $\langle \ldots m = e \ldots \rangle.m$. This term is rewritten into $(e\ \langle \ldots m = e \ldots \rangle)$ by a newly added reduction rule. This second representation can give more precise type information when considered together with overriding and inheritance operations, however in this paper we will not take them into consideration.

In the following we will reconsider such an approach in $\mathcal{E}_{hhf}$in order to integrate such a representation of objects with logic programming aspecs. In the following, we will start from a *class* representation and then we will focus on an *object-based* representation.

4.1 Class-based representation

A simple way of interpreting logic programs under the OO perspective, is to consider a program defining a predicate `p` as the definition (i.e. the et of methods) of the class `p`. Thus, an object is an *instance* of `p` of the form `p(t)`. This interpretation can be chosen also in a Linear Logic Language as shown in LO [10]. In accord to this idea, it is possible to define a quite simple **class-based** language. Each class is just a program defining a given predicate `classname` (in our setting a set of $\mathcal{D}$-clauses), such that each single method synchronizes an object of the form `classname State` and a message term `message` as follows:

```
cond >>  message || classname State -->
         messag_1 |... | messag_n || classname State´.
```

Here we assume that objects and methods are concurently activated during a computation. Thus, there is no order between them. For instance, a class defining a STACK can be defined as follows:

```
module obj.

type stack list Element -> o.
type pop o.
type push Element -> o.
type top  Element -> o.

  pop     || stack (X::S)  --> anti || stack S.
  push Y  || stack S       --> anti || stack (Y::S).
  top  X  || stack (X::S)  --> anti || stack (X::S).
```

Such a representation allow to define an inheritance hierarchy just declaring how the subclass is related to the superclass, see [12], as in the following case:

```
kind specifier type.
type point   specifier -> int -> o.
type point2D int       -> int -> o.
type move    specifier -> int -> o.   type x   specifier.
type get     specifier -> int -> o.   type y  specifier.

  (Nx is X + Dx) >>
     move Spec Dx || point Spec X --> anti || point Spec Nx.
  write X   >>
     get  Spec X  || point  Spec X --> anti || point Spec X.
  anti         || point2D X Y   --> anti || point x X | point y Y.
```

The class `point`, parameterized over a specifier `x,y,etc.`, has two methods: `get` which writes the current value of the point-coordinate, and `move` which updates the value of the point. The class `point2D` is defined in terms of the class `point`. As soon as one of the `point2D`-methods is invoked the corresponding object is *opened* and methods of the superclass can be used to handle the attributes *inherited* by the superclass. The open/closed representation allow to delete and to hide parts of an object just manipulating the atom representing it, while providing the above described notion of inheritance inspired by the one used in LO [12]. Differently from that approach we consider here an object as an atomic formula instead of a collection of atomic formulas (i.e., a ⅋ of attributes). This choice will be motivated when we will pass to an object-base representation in order to describe many OO features not considered in [12]. If we want add specific methods for objects of the class `point2D`, then we must use a more careful encoding to still enjoy inheritance from class `point`. For instance, we can write the following specification:

```
compdist D >>
  distance D  || point2D X Y --> anti || point2D X Y.
move Spec D || point2D X Y --> move Spec D || point x X | point y Y.
get  Spec D || point2D X Y --> get  Spec D || point x X | point y Y.
anti        || pointx X | point y  Y  --> anti || point2D X Y.
```

Here a `point2D` object is opened if and only if an inherited method must be invoked, a further operation has been introduced to close it after such messages. In the following sections we will study other representations in order to analyze a more general notion of objects in the logic programming which is more flexible in that it allow to study inheritance and overriding at the object-level.

4.2 An object-based representation

Encapsulation is a central point of the OO metaphor. In fact, objects are meant as boxes containing data and operations and exchanging messages in order to interact with the external world. In order to capture such feature in our setting we will chooose an atomic representation of objects in conformity with the previous examples. In [12] the authors adopt a different encoding which however is not suitable for our aims, as we will discuss later.

In order to define logical-objects, i.e., objects whose behavior is defined by a logical theory, each methods should be associated with a collection of clauses, applied when a given message occurs at some moment in the thread of the computation. In the following we shall consider objects labeled by identifiers which serve as references for message passing. Thus, an object is defined by a predicate `object` having type `Identifier -> o -> o -> o` and messages by predicate `send` having type: `Identifier -> Msg -> o` Methods belonging to a given object must be labeled with the corresponding identifier. The simplest way to achieve this is to define a method as a clause:

```
cond >> call id Mgs || object id St Ms --> Msg1 |..| Msgn || object id St´ Ms´.
```

Such a clause rewrites an object *id* and a message (sent to *id*) into the object itself and a multiset of new messages. These messages correspond to the body of the invoked method. Given the recursive nature of such a definition it is necessary to handle with care the quantification over each method definition. Let us consider a siple `get` method which retrieves the entire state `St` of an object `(object id v Ms)`, where `v` is a fixed value:

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

Here `M` must be a universally quantified variable to avoid cyclic definitions. In fact, such quantification allows to unfold the method-definition only at execution time, as it will be clearer later. For this reason, we will use terms like:

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

Thus, the complete object will be the term:

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

i.e., the third parameter is a $\mathcal{D}$-clause.

Given such an object, in order to execute its methods it is necessary to: synchronize it with a send-message; to fire the methods, i.e., move them at the top level in the left hand side of sequents; and to execute a backchaining step over the $\mathcal{D}$-clauses defining the methods.

In order to distinguish such steps, it is necessary to use the predicate `call` and `send` for method invocations. Method invocations which require the corresponding definitionto be fired are defined using `semd`, whereas invocations which assume that the definition has already been fired use `call`.

4.3 Self application

As outlined above, `send` should be considered as the principal primitive for invoking methods, while `call` serves to dispatch the message.

Definition 4.1 (Self-application rule) *The definition of the previously described* **meta**-*rule is given as follows:*

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

Once fixed the syntax above it is possible to specialize the proof system obtaining a sort of operational semantics for the object-based model. In fact, restricting the syntax s.t. goal formulas are $\parr$-disjunctions of objects and message atoms, and considering sequents in which the self-application rule is the unique clause occuring in the unbounded context, all the possible proofs have the following common pattern:

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma[\Phi]; \Delta \longrightarrow_{\Sigma} \mathtt{m}_1, \ldots, \mathtt{m}_n,\ \Omega \parallel (\mathtt{object\ id\ st}'\ \mathtt{ms}'),\ \Theta}{\Gamma[\Phi]; \mathtt{ms},\ \Delta \longrightarrow_{\Sigma} (\mathtt{call\ id\ m}),\ \Omega \parallel (\mathtt{object\ id\ st\ ms}),\ \Theta}\ \vdots\ bc + \parr_r}{\Gamma[\Phi]; \mathtt{ms}, \Delta \longrightarrow_{\Sigma} \mathtt{object\ id\ st\ ms} \parr \mathtt{call\ id\ m},\ \Omega \parallel \Theta}\ \parr_r}{\Gamma[\Phi]; \Delta \longrightarrow_{\Sigma} \mathtt{ms} \multimap (\mathtt{object\ id\ st\ ms} \parr \mathtt{call\ id\ m}), \Omega \parallel \Theta}\ \multimap_r}{\Gamma[\Phi]; \Delta \longrightarrow_{\Sigma} (\mathtt{send\ id\ m}\ ), \Omega \parallel (\mathtt{object\ id\ st\ ms}), \Theta}\ bc$$

Here the message m is supposed to be defined in a clause of `ms` with body

$\mathtt{m}_1 \mid \ldots \mid \mathtt{m}_n$ `|| object id st´ ms´`,

which can be successfully applied in a backchaining step.

The previous figure gives rise to a number of important observation:

1. A method has a complete control over the structure of the object to whom it belongs. Playing with quantification, it is possible to modify the object state and methods (in the figure denoted by `st´,mt´`). In particular it is possible to destroy an object `id`, by invoking the following method (defined in `id` itself):

   ```
   kill Id  ||  object Id St Ms --> anti ||  anti.
   ```

2. Messages are received without any order, as $\parr_r$ decomposes the body of methods into a multi-set of messages.

3. If `ms` is a &-conjunction of methods the non-determinism in the choice of one of its conjuncts guarantees the selection of the right clause matching with the current message to be executed.

There is a problem hidden in such a mechanism which can introduce anomalies: the execution of a `call` could be delayed, even if the methods of an objects have been already fired. Since messages arrive without any order the execution of another message could change the structure of the original object while a copy of its old methods is still in the bounded context. To prevent this anomaly, it is enough to distinguish between objects which can receive messages and objects which are executing methods. This distinction is possible rephrasing the self-application rule as follows:

Definition 4.2 (Self-application revisited) *Let us rewrite the rule in Def. 4.1 as follows:*

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

and forcing methods to have the following format:

```
(call id Msg || frozen id V Ms --> Msgs || object id V′ Ms′).
```

This trick allows to frozen an object during the execution of one of its methods. In the following, a state will be denoted by a multiset of `object` and `frozen` atoms, whereas messages will be of the form `send` and `call`. In the following we will clarify the operational behavior of this object calculus by an analysis of the proof-theoretical properties of this encoding.

Definition 4.3 (Consistent state) *A state Θ is consistent iff the identifiers of the object-terms occuring in Θ are different from each other.*

In the sequel, *send* will denote the rule scheme derived by the following sequence of rules:

$$\dfrac{\dfrac{\dfrac{\Gamma[\Phi];\texttt{ms},\Delta \longrightarrow_\Sigma \texttt{call id m},\Omega \parallel \texttt{object id st ms},\Theta}{\Gamma[\Phi];\texttt{ms},\Delta \longrightarrow_\Sigma \texttt{frozen id st ms} \mathbin{⅋} \texttt{call id m},\Omega \parallel \Theta}\,⅋_r}{\Gamma[\Phi];\Delta \longrightarrow_\Sigma \texttt{ms} \multimap (\texttt{frozen id st ms} \mathbin{⅋} \texttt{call id m}),\Omega \parallel \Theta}\,\multimap_r}{\Gamma[\Phi];\Delta \longrightarrow_\Sigma \texttt{send id m},\Omega \parallel \texttt{frozen id st ms},\Theta}\,bc$$

Notice that, after the bc application, the sequences of steps can be forced by permuting the right rules that can be applied to Ω in the proper way. In the following lemma, given a proof δ all the sequences of rules matching with the above defined pattern, will be considered as a single applications of the smtt send rule.

Lemma 4.4 (Permutation of object logical rules) *Let Γ consists only of rule for message-passing, Θ_o be a consistent state and Ω a multiset of message-terms.*
Let $s = \Gamma[\Phi];\Delta \longrightarrow_\Sigma \Omega \parallel \Theta$ and δ s.t. $\delta \vdash_e \Gamma[\Phi];\Delta \longrightarrow_\Sigma \Omega \parallel \Theta$, 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. Furthermore, assume that Δ does not contain any other methods labeled by "id" and each state Θ_i occuring 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 $send_{id}$ and the corresponding *consumption* of METHS$_{id}$. The base case is trivial. If the distance is greater than zero, then the proof is by cases on the rule r which immediately follows the $send_{id}$. There are two possibilities: r can be $send_{id'}$ ($id' \neq id$, since id is frozen by a $send_{id}$, i.e., it cannot receive other messages), then r can be permuted with *send* and the thesis follows by applying the inductive hypothesis.

If the last rule is the *bc* one, involving $\text{METHS}_{id'}$, `frozen id'`, and `call id'` (again $id' \neq id$ by hypothesis), then it has the following pattern:

$$\cfrac{\cfrac{\cfrac{\eta}{\Gamma[\Phi];\text{METHS}_{id},\Delta \longrightarrow_\Sigma \texttt{call id,m}_1\ldots\texttt{m}_n,\Omega \parallel \texttt{frozen id,object id}',\Theta}}{\Gamma[\Phi];\text{METHS}_{id},\text{METHS}_{id'},\Delta \longrightarrow_\Sigma \texttt{call id,call id}',\Omega \parallel \texttt{frozen id,frozen id}',\Theta}\ R}{\Gamma[\Phi];\text{METHS}_{id'},\Delta \longrightarrow_\Sigma \texttt{send id,call id}',\Omega \parallel \texttt{object id,frozen id}',\Theta}\ send$$

where
$\texttt{call id}' \mid\mid \texttt{frozenid}' \;--> \; \texttt{m}_1\ldots\texttt{m}_n \mid\mid \texttt{object id}'$ belongs to $\langle\text{METHS}_{id'}\rangle$, and $R = bc{+}\,\invamp_r$. Notice that the sequence of $\invamp_r$ after the application of $\text{METHS}_{id'}$ is again forced by the definition of the rules of the $\mathcal{E}_{hhf}$-proof system. Thus it is possible to re-arrange the proof as follows:

$$\cfrac{\cfrac{\cfrac{\eta}{\Gamma[\Phi];\text{METHS}_{id},\Delta \longrightarrow_\Sigma \texttt{call id,m}_1,\ldots,\texttt{m}_n,\Omega \parallel \texttt{frozen id,object id}',\Theta}}{\Gamma[\Phi];\Delta \longrightarrow_\Sigma \texttt{send id,m}_1\ldots\texttt{m}_n,\Omega \parallel \texttt{object id,object id}',\Theta}\ send}{\Gamma[\Phi];\text{METHS}_{id'},\Delta \longrightarrow_\Sigma \texttt{send id,call id}',\Omega \parallel \texttt{object id,frozen id}',\Theta.}\ R$$

Then applying the inductive hypothesis on the subproof terminated by *send* yields 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 role of backchaining over a *send* let us define the following operational rule:

Definition 4.5 (Objects-rewriting rule) *Let $\rightsquigarrow_o$ be the relation over pairs $\langle\Omega \mid \Theta\rangle$ of multiset of message- and object-terms, respectively, given by:*
$\langle(\texttt{send id m}),\Omega \mid (\texttt{object id st mts}),\Theta\rangle \rightsquigarrow_o \langle\texttt{m1},\ldots,\texttt{mn},\Omega \mid \texttt{object id st}'\ \texttt{mts}'),\Theta\rangle$
where

```
call id m || object id st mts --> m1 | ... |mn || object id st´ mts´
```

belongs to $\langle\texttt{mts}\rangle_\Sigma$, i.e., it is an instance of a clause in `mts`.

Let $\rightsquigarrow_o^\star$ be the transitive closure of $\rightsquigarrow_o$. It is possible to state the following result, where only **consistent** states will be considered both in proofs and in sequences of $\rightsquigarrow_o$ steps.

Proposition 4.6 (Deduction and rewriting) *Let Γ consists only of the rule for message-passing, Θ_o a multiset of* `object`*-terms, and Ω a multiset of* `send`*-terms, then the sequent $\Gamma[\Phi];\emptyset \longrightarrow_\Sigma \Omega \parallel \Theta_o$ has a proof iff $\langle\Omega \mid \Theta_o\rangle \rightsquigarrow_o^\star \langle\emptyset \mid \Theta_f\rangle$, where $\Phi \multimap \Theta_f$ is provable in Forum.*

Proof : The *only-if* part is immediate from the definition. The *if*-part follows by induction on the length of δ and using lemma 4.4 (which does not modify the length of proofs). $\square$

Some examples will clarify this point.

4.4 Examples

Before showing some examples of object definition, let us give some hints about the adopted notation. First of all, it is esay to figure out a sort of class-definition or, better, a template for objects with the same structure, by defining clauses for a predicate `new` which creates new instances. It has the following format:

```
type new  A -> Values -> Identifier  -> o.

        new classname InitValues Id || anti --> anti || object Id St Mts.
```

Here `InitValues` stands for the initial values of the attributes of the object being created. This clause simply asserts that, given some class name `c`, initial values `v`, and an identifier `id`, the message `new c v id` must be rewritten into an `object`-term `object id v mts`, where `mts` are defined accordingly to the specification of the class.

Based on it, let us present some examples.

Integers. It is possible to define objects behaving like integers, i.e., with methods `succ, pred` and `zero`,as shown in the following $\mathcal{E}_{hhf}$-specification:

```
type zero? o.           type succ  o.
type pred  o.           type intclass o.

(true || new intclass anti Id || anti --> anti || object Id St Ms):-
  N = (M \ (pi S \ pi P \ pi Q \        % N is a lambda-abst. over M
          (call Id pred || frozen Id S (P & Q) -->
                                anti || object Id S (M & Q))
      )),
  Ms = (                                % two methods: zero? and succ
        (pi S \ pi M \ pi T \
          ((write "ZERO!" >> call Id zero? || frozen Id S M -->
                                        anti || object Id S M)
           &
           (call Id succ  || frozen Id S    (M  & T)  -->
                              anti || object Id S ((N M) & T)))).
```

The idea is that the objects-integers are supposed to have just two methods and no state. The method `succ` is always present, and, when invoked, it overrides the other one (either `zero?` or `pred`, with a new `pred`-definition which, in turn, returns the current definition of the object. In the picture above the an integer-object is created by invoking the predicate `new intclass id` which simply rewrites itself into an object term with the method zero? and `succ`. For instance, the goal `?- new intclass 1 | send 1 succ.` creates an object `1` which corresponds to *zero* and then it tranforms it into *one*. A further message `send 1 pred` will recover the old state of the object, i.e., it turns into *zero*.

4.5 Concurrent Objects

In the presented model objecs and messages float in the sequents, simulating a completely concurrent execution. In this section we will show how to define objects working correctly in such a setting using the factorial as example.

This example is useful to show how to exploit universal quantification on the right-hand side of sequents to create private objects (s.t. their identifiers are visible only to the creating object). The idea is to enrich the general pattern of methods as follows:

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

The innermost quantifier `pi` allows to *hide* the element `X` occuring in the body of methods. Notice that the meta-rule for the self-application, is not affected by this modification. However the logical deduction of a *send* message implies a further step, i.e., an application of the $\forall_r$ rule, which allows to simplify the quantification introducing new constants.

Definition 4.7 *Let* $\overset{\forall}{\rightsquigarrow}_o$ *be the relation over pairs of message and objects terms given by:*
$\langle \texttt{send id m}, \Omega \,|\, (\texttt{object id st mts}), \Theta \rangle \overset{\forall}{\rightsquigarrow}_o \langle \texttt{m1}', \ldots, \texttt{mn}', \Omega \,|\, (\texttt{object id st}'' \; \texttt{mts}''), \Theta \rangle$. *where*

```
call id m || frozen id st mts -->  pi V \ m1| ... | mn || object id st´ mts´
```

belongs to $\langle \texttt{mts}\rangle_\Sigma$*, and smtt st", mts´´, mi$'$ result by substituting new constants for* V *(a vector of variables) in* st´*,* mts´ *and* $\texttt{m}'_\texttt{i}$*,* $i : 1, \ldots, n$.

It is easy to check that $\overset{\forall}{\leadsto}_o$-transition corresponds to an $\mathcal{E}_{hhf}$-proofs, according to proposition 4.6. In the following, $\overset{\forall *}{\leadsto}_o$ will denote its transitive closure. The following example is useful to see how to program with concurrent objects. It also embeds some of the new features introduced in this section.

Factorial objects. Let `factclass` be a class of objects which, upon reception of the message `(fact N O)` (`N` is an integer and `O` is an object identifier), send `N!` to `O`. In order to implement them in a concurrent setting, it is also necessary to define `slave`-objects which take in charge the actual computation of the factorial, allowing their masters to process more messages at a time.

```
type factclass o.           type slave     o.
type receiver  o.           type fact      int -> Identifier -> o.
type eval      int -> o.    type st        int -> Identifier -> o.

% Class of "slave objects"
(new slave St Id || anti --> anti || object Id St Ms):-
  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 of "factorial objects"
(new factclass St Id || anti --> anti || object Id St Ms):-
  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
          ))).

% Class "receiver"
(new receiver St Id || anti --> anti || object Id St Ms):-
  Ms =
   (pi N \ pi S \ pi M \
      write N >> call Id (eval N) || frozen Id S M  --> anti || anti).
```

Here, `slave`-objects can receive a message `eval` Let $\textsc{o}_{id}[\textsc{v},\textsc{id}]$ be a `slave`-object with state ($\textsc{v}$,$\textsc{id}$) and identifier id. Now if a given object R send the request (`fact 4` $\texttt{id}_\texttt{r}$) to an object F (id_f) of class FACTORIAL, i.e., $\Theta = \{\textsc{r}_{id_r}, \textsc{f}_{id_f}\}$ and $\Omega = \{\texttt{send id}_\texttt{f}\ (\texttt{fact 4 id}_\texttt{r})\}$, then one of the possible transitions is:

$$\langle \Omega \mid \Theta\rangle \overset{\forall}{\leadsto}_o \langle \texttt{new slave (st 4 id}_\texttt{r}\texttt{) x4, send id}_\texttt{f}\ \texttt{(fact 3 x)} \mid \Theta\rangle \overset{\forall}{\leadsto}_o$$
$$\overset{\forall}{\leadsto}_o \langle \texttt{new slave 3 x y), send id}_\texttt{f}\ \texttt{(fact 2 y)} \mid \textsc{o}_x[4, id_r], \Theta\rangle \overset{\forall}{\leadsto}_o$$
$$\overset{\forall}{\leadsto}_o \langle \texttt{send y (eval 2)} \mid \textsc{o}_y[3, x], \textsc{o}_y[4, id_r], \Theta\rangle \overset{\forall}{\leadsto}_o \langle \texttt{send x (eval6)} \mid \textsc{o}_y[4, id_r], \Theta\rangle \overset{\forall}{\leadsto}_o$$
$$\overset{\forall}{\leadsto}_o \langle \texttt{send id}_\texttt{r}\ \texttt{(eval24)} \mid \Theta\rangle.$$

It is interesting to notice that the SLAVE objects disappear after sending the result of their own computation (see the definition of method $fact$), i.e., a sort of garbage collection is performed together with the computation. Furthermore, a FACTORIAL-object can process many $fact$-messages in parallel since it demands the computation to its `slave`. For instance, let Θ be as before and add a new message ($\texttt{fact3id}_\texttt{r}$) to Ω.

$$
\begin{array}{c}
\langle \texttt{send id}_\texttt{f}\ \texttt{(fact 4 id}_\texttt{r}\texttt{)}, \texttt{send id}_\texttt{f}\ \texttt{(fact 3 id}_\texttt{r}\texttt{)} \mid \Theta\rangle \overset{\forall}{\leadsto}_o \\
\langle \texttt{new\ \ slave (st 4 id}_\texttt{r}\texttt{) x}, \texttt{send id}_\texttt{f}\ \texttt{(fact 3 x)}, \texttt{send id}_\texttt{f}\ \texttt{(fact 3 id}_\texttt{r}\texttt{)} \mid \Theta\rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \texttt{send id}_\texttt{f}\ \texttt{(fact 3 x)}, \texttt{new\ \ slave (st 3 id}_\texttt{r}\texttt{) x}'\texttt{3}, \texttt{send id}_\texttt{f}\ \texttt{(fact 2 x}'\texttt{)} \mid \circ_x[4, id_r], \Theta\rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \texttt{new\ \ slave 3 x y)}, \texttt{send id}_\texttt{f}\ \texttt{(fact 2 y)}, \texttt{send id}_\texttt{f}\ \texttt{(fact 2 x}'\texttt{)} \mid \circ_\texttt{x}[4, \texttt{id}_\texttt{r}], \circ_{\texttt{x}'}[3, \texttt{id}_\texttt{r}], \texttt{0}\rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \texttt{send x}'\ \texttt{(eval 2)}, \texttt{send y (eval 2)} \mid \circ_y[3, x], \circ_x[4, id_r], \circ_{x'}[3, id_r], \Theta\rangle \overset{\forall}{\leadsto}_o \\
\langle \texttt{send id}_\texttt{r}\ \texttt{(eval 6)}, \texttt{send x (eval 6)} \mid \circ_x[4, id_r], \Theta\rangle \overset{\forall}{\leadsto}_o \\
\overset{\forall}{\leadsto}_o \langle \texttt{send id}_\texttt{r}\ \texttt{(fact 24)}, \texttt{send id}_\texttt{r}\ \texttt{(fact 6)} \mid \Theta\rangle.
\end{array}
$$

It is possible to go further in the refinement of the object representation in order to study other aspects of OO programming. For instance, objects are encoded so to encapsulate data and methods. it would be interesting to be able of hiding part of their structure. Using the above notations, it is possible to enhance the definition of the `new` meta-rule in order to exploit universal quantification at the moment of the *object creation*:

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

The variables `X` correspond to parts of the objects that cannot be accessed from other objects (unless the newly creating objects decides to make them public). As in the previous use of quantification, such a constraint is guaranteed by the side condition of the *bc* rule. Now in an object definition there are three level in which universal quantification is used:

- before an object declaration: it allows to hide parts of the object;
- before the methods declaration: it allows to abstract over their definition;
- before the body of a method: it allows to define private objects (in a given computation).

The following example illustrates the second point.

Private structures. In the scheme presented a above, methods can access the state values by using pattern matching or unification in their definitions (see for instance the definition of `slave`-objects). However, it is also possible for a user-defined clause like

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

to break encapsulation. In fact, when called, it unifies with any object currently active and returns its internal state.

In order to prevent this form of access to the internal state of of an object, it is possible to quantify over the state constructors. For instance, let us consider a stack-definition with hidden implementation:

```
type stack o.
type pop  Element  -> o.
type push Element -> o.

(new stack St Id || anti -->
        (pi a \ (anti || object Id (a nil) (Ms a)))):-
 Ms = a \ (pi S \ pi M \ pi El \
    ((call Id (pop  El) || frozen Id (a (El::S)) M  -->
                        anti || object Id (a S) M)
       &
     (call Id (push El) || frozen Id (a S) M  -->
                        anti || object Id (a (El::S)) M))).
```

The higher-order quantification over `a` hides all the occurrences of such a constructor in the state and methods formulas. The above `break` predicate cannot be applied to such objects since constants introduced with `pi` cannot be binded to free variables outside its scope, see [57, 43].

Private 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 enough to quantify over the names of the private methods. For instance, in the following declaration the method `pub` is public while `priv` can be invoked only by other methods of the same object.

```
type private o.
type pub     o.

(new private St Id || anti -->
         (pi m \ (anti || object Id St (Ms m)))):-
 Ms =  priv \
  (pi S \ pi M \
   (
    (call Id priv || frozen Id S M  --> ... || ...)
     &
    (call Id pub  || frozen Id S M  --> send Id priv || object Id S M)
   )
  ).
```

The introduction of universal quantification in the definitions of objects allows to exploit the underlying logics in a nice way, but makes difficult to compose objects (i.e., to define complex ones starting from elementary ones). This is the argument of the following section.

5 Operations over objects

In this section, the representation of objects and classes will be refined in order to define operations over objects such as *extension*, *delegation/inheritance*. In order to manipulate object without being constrained by the specific identifier associated to its methods, it is possible to use λabstractions fix the following new representation:

```
          object id State (id \ Methods),
```

i.e., the third parameter of the predicate `object` is a λ-abstraction over the object-identifier. According to this definition, we must modify the rule for message passing as follows:

```
type object   Identifier -> o -> (Identifier -> o) -> o.
type frozen   Identifier -> o -> (Identifier -> o) -> o.
type send     Identifier -> o -> o.
type call     Identifier -> o -> o.

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

As immediate consequence we can define the following operations to compose elemntary objects into complex ones and viceversa:

```
type extend   Identifier -> (Identifier -> o) -> o.
type clone    Identifier ->  Identifier -> o.
type clone    Identifier ->  Identifier -> o.

  extend Id M || object Id St Ms -->
```

```
              anti || object Id St (id \ ((Ms id) & (M id))).

  clone Id Id1 || object Id St Ms -->
              anti || object Id St Ms | object Id1 St Ms.

  merge Id Id1 || object Id St Ms | object Id1 St Ms1-->
               anti || object Id (St & St1) (id \ ((Ms id)& (Ms1 id))).
```

The extend operator allows to add a new method definition to an object (the method must be defined by a λabstraction over the object identifier); the clone operator makes a new copy of an object, with a new identifier; the merge operator merges two objects. Similar operations can be figured out by exploiting the previous compact representation.

Another interesting apsects concerns the study of inheritance or better delegation since we work here at the object-level. Actually, the operator *extend* already provides for object-based inheritance since it allows to create new objects on the basis of existing ones. However it is possible to create chains of linked objects which inherit methods from each other without duplication of code, by using a meta-rule called *link* as shown in the following example.

```
type obj      o.
type dobj      o.
type st       int -> o.
type get      o.

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

 new obj St Id || anti --> anti || object Id St Ms:-
  Ms = id \
       (pi M \ pi V \
        (write V || call id get || frozen id (st V) M -->
                           anti || object id (st V) M)).

 del dobj St Id Id1 || anti --> anti || object Id St Link:-
  Link = id \
         (pi St \ pi Ms \
          (true  || anti           || object Id1 St Ms -->
                 ((Ms id) -o anti) || object Id1 St Ms)).
```

The method link in the object dobj allows, for instance, to invoke the method get of an object Id1 of class obj over Id. In fact, this rule simply fires the methods of the delegating object Id1 when a message sent to Id must be executed. In this sense Id automatically inherits all the methods of its superobject Id1. Other examples can be depicted by studying similar meta-rule, enjoying the higher-order nature of the underlying logic, as shown in [21].

6 Object mobility

As last example we present here a simulation of code-mobility in a distributed system. The idea is to consider the different places of the systems and the processes executing over them, allowing primitives to move the execution from one place to another. Actually, we can consider the mobile entity as an agent or as an object according to our previous idea. In the following examle we will just sketch this idea without cosidering objects as complex as in the previous sections. We will base our example on the following signature:

```
module mobile.
type place    Place -> list Agent -> list Facts -> o.
type agent    Place -> Def -> Thread -> Res -> Agent.
```

```
type send     Agent -> Msg -> Res -> o.
type residue  Place -> list Agent -> o.
type exdb     Agent -> Op -> list Facts -> list Facts -> o.
type go       Place -> o.
type db       Op -> o.
```

The predicate `place` denotes a place where more agents can be executed working on a shared memory (a list of facts). An `agent` is a process with its own operations and a thread of execution: `agent P Agent Ops NewAgent` stands for an agent executing one of the operations `Ops` in the place `P`, yielding the new state `NewAgent`. We assume to have three types of messages internal operations, `go P` which forces an Agent to move to the place `Q`, and `(db Op)` which performs an operation over the shared memory of the place where the object is. Here we assume that an object in place `P` cannot access the memory of another place. We can describe the evolution of such a system as follows:

```
sel Agent Agents Others  >>
  anti || place P Agents DB -->  Agent || place P Others DB.

agent P Agent (go Q,Msgs) NewAgent  || place Q Agents DB  -->
  anti  || place Q ((agent Q Agent Msgs NewAgent)::Agents) DB.

exdb Agent Op DB NewDB >>
 agent P Agent (db Op) NewAgent  ||  place P Agents DB -->
   anti || place P Agents NewDB.

exdb Agent Op DB NewDB >>
 agent P Agent (db Op,Msgs) NewAgent ||  place P Agents DB -->
   agent P Agent Msgs NewAgent || place P Agents NewDB.

send Agent Msg NewAgent >>
 agent P Agent Msg NewAgent  || residue P Rs  -->
   anti  || residue P (NewAgent::Rs).

send Agent Msg IntAgent ||
 agent P Agent (Msg,Msgs) NewAgent || place P Agents DB -->
  anti || place P ((agent P IntAgent Msgs Res)::Agents) DB.
```

The operation `sel` selects the agents of a given place, in order to allow to concurrently execute them as parallel goals; in fact, it moves the agent form inside a place to the Ω-part of a sequent. The operations `exdb` and `send` correspond to the execution of the operation of the selected agent.

The operations of agents which retrieve/store information from/to the shared memory can be specified as follows:

```
type app Info -> o.
type store o.
type get Info -> o.

 exdb  Agent (get X) (X::DB) (X::DB).
 exdb  Agent (store) DB NewDB:- append Agent DB NewDB.

 send  Agent (app Y) (Y::Agent):-write Y.
```

Now we can write the code of an agent which, travelling from place to place, collects information from each distributed memory (assuming the places 1,2, and 3) and stores it at the starting place (i.e., place 1):

```
  (agent 1 nil (go 2,db (get X),app X,go 3,db (get Y),app Y,go 1,db store) Outa)
```

Providing more complex operations could allow to describe more sophisticated protocols of communications, and in particular to study security aspects of code-mobility by simulating all the dangerous configurations in a distributed system.

7 Comparisons and Conclusions

References

[1] M. Abadi and L. Cardelli. A Theory of Primitive Objects. In *Proceedings of Theoretical Aspect of Computer Software*, volume 789 of *LNCS*, pages 296–320. Springer-Verlag, 1994.

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

[3] G. Agha. Actor: A model of Concurrent Computation in Distributed Systems, 1986. MIT Press.

[4] 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.

[5] H. Aït-Kaci, R. Nasr, and P. Lincoln. LeFun: Logic, equations, and Functions. In J.Maluszynski and M.Wirsing, editors, *Proceedins of the Symposium on Logic Programming*, pages 17–23. IEEE, Computer Society Press, 1987.

[6] H. Aït-Kaci and A. Podelski. Towards a Meaning of LIFE. *Journal of Logic Programming*, 16:255–274, August 1993.

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

[8] P. America and J.J.M.M. Rutten. *A Parallel Object-Oriented Language: Design and Foundations.* PhD thesis, Free University Amsterdam, 1989.

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

[10] J.M. Andreoli and R. Pareschi. Linear Objects: Logical Processes with Built-In Inheritance. In D.H. Warren and P.Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming*, pages 495–510. The MIT Press, Cambridge, MA, 1990.

[11] J.M. Andreoli and R. Pareschi. Communication as Fair Distribution of Knowledge. In *Proceedings of OOPSLA '91*, 1991.

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

[13] G. Bellin and Philip J. Scott. On the π-calculus and linear logic. Manuscript, 1992.

[14] 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, LNCS Vol.592*, pages 102–113. Springer-Verlag, Berlin, 1991.

[15] M. Bugliesi. A Declarative View of Inheritance in Logic Programming. In *Joint International Conference and Symposium on Logic Programming*, pages 112–127, 1992.

[16] M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. *Journal of Logic Programming*, 19:443–501, 1994.

[17] 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.

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

[19] 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.

[20] A. Davison. Polka: A Parlog Object Oriented Language, 1989. PhD Thesis, Dept. of Computing, Imperial College London.

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

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

[23] A. Eliëns. Extending Prolog to a Parallel Object-Oriented Language. In *Proceedings of IFIP W.G. 10.3 Working Conference on Decentralized Systems*, 1989.

[24] E. Ellis and B. Stroustrop. *The Annotated C^{++} Reference Manual*. ACM Press, 1990.

[25] U. Engberg and G. Winskel. Petri nets as models of linear logic. In A. Arnold, editor, *Proceedings of Colloquium on Trees in Algebra and Programming*, pages 147–161, Copenhagen, Denmark, 1990. Springer-Verlag LNCS 389.

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

[27] K. Fisher and J.C. Michell. Notes on typed object-oriented programming. In *Proceedings of Theoretical Aspect of Computer Software*, volume 789 of *LNCS*, pages 844–885. Springer-Verlag, 1994.

[28] K. Fisher and J. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proceedings of FCT-95*, Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.

[29] K. Fukunaga and S. Hirose. An experience with a prolog-based object-oriented language. In *Proc. OOPSLA'86*, pages 224–231. ACM, September 1986.

[30] H. Gallaire. Merging Objects and logic programming: Relational Semantics. In *AAAI 86*, pages 754–758, 1986.

[31] D. Galmiche and E. Boudinet. Proof search for programming in intuitionistic linear logic. In D. Galmiche and L. Wallen, editors, *CADE-12 Workshop on Proof Search in Type-Theoretic Languages*, pages 24–30, Nancy, France, June 1994.

[32] D. Galmiche and E. Boudinet. Proofs, Concurrent Objects and Computations in a FILL Framework. In *Proceedings of the Workshop on Object-Based Parallel and Distributed Computation (OBPD 95), Tokyo, Japan*, volume 1107 of *Lecture Notes in Computer Science*, pages 148–167, 1995.

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

[34] V. Gehlot and C.A. Gunter. Nets as tensor theories. In G. De Michelis, editor, *Proceedings of the tenth International Conference on Application and Theory of Petri Nets*, pages 174–191, Bonn, Germany, 1989. Extended and revised version available as Technical Report MS-CIS-89-68, University of Pennsylvania.

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

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

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

[38] P. Harbison. Modula-3, 1992. Prentice Hall, ISBN 0-13-596396-6.

[39] J.A. Harland and D. J. Pym. The Uniform Proof-Theoretical Foundation of Linear Logic Programming (Extended Abstract). In *Proceedings of the International Logic Programming Symposium*, pages 304–318. The MIT Press, 1991.

[40] C. Hewitt. Viewing Control Structures as Patterns of Passing Messages. *Artificial Intelligence*, pages 323–364, 1977.

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

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

[43] 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.

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

[45] Y. Ishikawa and M. Tokoro. A concurrent object oriented knowledge representation language orient84/k: its features and implementation. *OOPSLA 86 SIGPLAN Notices*, 21(11):232–241, 1986.

[46] 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, 1987.

[47] G.M. Karam. Prototyping Concurrent Systems with MultiLog. Technical report, Dept. of Systems and Computer Engineering, Carleton University Ottawa, 1988.

[48] N. Kobayashi and A. Yonezawa. Asynchronous Communication Model based on Linear Logic. *Formal Aspects of Computing*, 3:279–294, 1994. Short version appeared in Joint International Conference and Symposium on Logic Programming, Washington, DC, November 1992, Workshop on Linear Logic and Logic Programming.

[49] N. Kobayashi and A. Yonezawa. Typed Higher-Order Concurrent Linear Logic Programming. Technical report, Department of Information Science the University of Tokyo, Report No. 94-12, 1994.

[50] Y. Lafont. Functional programming and linear logic. Lecture notes for the Summer School on Functional Programming and Constructive Logic, Glasgow, United Kingdom, 1989.

[51] P. Lincoln. *Computational Aspects of Linear Logic.* PhD thesis, Stanford Univeristy, 1992.

[52] Inmos Ltd. The Occam Programming Language, 1984.

[53] F. G. McCabe. *Logic and Objects.* PhD thesis, Imperial College of Science and Technology, Dept. of Computing, London, UK, 1988.

[54] J. Meseguer. Conditional rewriting logic, technical report. Technical report, SRI International, Stanford University, 1993.

[55] J. Michell. Toward a Typed Foundation for Method Specialization and Inheritance. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 109–124. ACM, 1990.

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

[57] 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.

[58] D. Miller. Unification Under a Mixed Prefix. *Journal of Symbolic Computation*, 14:321–358, 1992.

[59] 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 *LNCS*, pages 242–265. Springer-Verlag, Berlin, 1993.

[60] D. Miller. A Multiple-Conclusion Meta-Logic. In *Proceedins of the 1994 Symposium on Logics in Computer Science, Paris*, pages 272–281, 1994.

[61] D. Miller. Survey of Linear Logic Programming. *Computational Logic: The Newsletter of the European Network of Excellence in Computational Logic*, 2(2):63–67, 1995.

[62] D. Miller. Forum: A Multiple-Conclusion Specification Logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[63] 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.

[64] L. Monteiro and A. Porto. Contextual logic programming. In M. Martelli G. Levi, editor, *Proc. Sixth Int. Conf. on Logic Programming*, Cambridge, MA, 1989. The MIT Press.

[65] L. Monteiro and A. Porto. A transformational view of inheritance in logic programming. In *Proc. Seventh Int. Conf. on Logic Programming*, pages 481–494, Cambridge, MA, 1990. The MIT Press.

[66] O. Lehrmann Madsen and B. Möller-Pedersen and K. Nygaard. *Object Oriented Programming in the BETA Programming Language.* Addison Wesley, 1993. ISBN 0-201-62430-3.

[67] G. Perrier. A model of concurrency based on linear logic. In *Proceedings of the Conference on Computer Science Logic 95, Paderborn, Germany*, 1995.

[68] H. Schellinx. Some Syntactical Observations on Linear Logic. *Journal of Logic and Computation*, 1(4):537–559, 1991.

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

[70] G. Smolka. A feature logic with subsorts, lilog report 33. Technical report, IWBS, IBM Deutscland, Stuttgard, Germany, 1988.

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

[72] R. Ström. A comparison of the object-oriented and process paradigm. *SIGPLAN Notices*, 10:88–96, 1986.

[73] J. Tokar. Ada 95: The Language for the 90's and Beyond. *Ada User Journal*, 16(3), 1995.

[74] D. Ungar and R.B. Smith. Self: the Power of Simplicity. In *Proceedings of ACM Symp. on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–241. ACM Press, 1987.

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

[76] M. Winikoff and J. A. Harland. Some Applications of the Linear Logic Programming Language Lygon. In *Proocedings of the 19th Australasian Computer Science Conference, Melbourne, Australia.* The MIT Press, 1996.