Proofs, concurrent objects and computations in a FILL framework*

D. Galmiche & E. Boudinet
CRIN-CNRS & INRIA Lorraine
Campus Scientifique - B.P. 239
54506 Vandœuvre-lès-Nancy Cedex
France
e-mail: galmiche{boudinet}@loria.fr

Abstract. There are several major approaches to model concurrent computations using logic. In this context, one aim can be to achieve different forms of programming as logic, object-oriented or concurrent ones in a same logical language. Linear logic seems to be well-suited to describe computations that are concurrent and based on state transitions. In this paper, we propose and analyze a framework based on Full Intuitionistic Linear Logic (FILL), logical fragment with potentialities for non-determinisms management, as foundation of concurrent object-oriented programming, following the two paradigms *proof-search as computation* and *proofs as computations*.

1 Introduction

There exists several approaches to model concurrent computations using logic or object-oriented programming. Logic programming allows to write executable specifications while object-oriented programming allows to build up complex structures and a possible challenge consists in finding an approach mixing in a same logical framework such various forms of high-level reasoning. Moreover a representation of based-on object mechanisms within a suitable logical language can provide them with a clearer semantics [7]. Trying to use, in this context, an adequate logic to represent or specify concurrency and also some interesting non-determinisms can complete the challenge. Linear logic (LL) [12] seems to be well-suited to describe concurrent or sequential computations based on state transitions [29], combining a constructive character with a finer control on resource management.
Let us recall the different ways to model concurrent computations using fragments of linear logic. A first one is based on *formulas-as-types* and *proofs-as-programs* paradigms in which propositions are interpreted as types, proofs as programs and proof normalization processes as computations. Here, main works have been devoted to term assignment for intuitionistic linear logic (ILL) [6, 23] or classical LL [1] including proposals of concurrent functional programming languages. A second way is based on the *formulas-as-states* and *proofs-as-computations* paradigms [27] where the connections with Petri nets and linear logic have been investigated. The correspondence between ILL and Petri nets [22, 27] illustrates the interest of proof

* Proceedings of the Workshop on Object-based Parallel and Distributed Computation, OBPDC'95, Tokyo, Japan, 1995, LNCS 1107, pp 148-167, Springer Verlag

search methods for proving specifications (and synthesizing programs) of distributed systems. A third complementary way is based on the *proof-search as computation* paradigm and leads to works on logic programming [3], concurrent logic programming [5, 18, 20] in fragments of LL. As in the previous approach, proof construction is essential and we need to have efficient proof search methods for the logic we consider, for example based on specific proof schemas as uniform proofs in [14, 33] or canonical proofs [8, 11]. Other proposals are based on rewriting logic as a unified model of concurrency and as a theory of concurrent objects [25].
In fact we have two main goals in this paper. The first one is to motivate the interest of the use of fragments of LL for concurrent object programming, for example with a process calculus where the basic mechanisms are at the logical level. This last point leads to the question of using logic for representing the main programming mechanisms, its interest (as a clear semantics) or difficulty (how to insert a new operator dedicated to a mechanism into the logic and the impact on proof search). The second and main one is to show that the Full Intuitionistic Linear Logic ($FILL$) [16] seems adequate as logical foundation for a framework to involve object-oriented, concurrent and logic programming (following the previously mentioned paradigms) and that to illustrate its ability to represent non-determinisms at different levels (messages, process, proof search) in the context of concurrent systems specification.

In section 2, we recall the concepts about concurrent computations in linear logic fragments, focusing on the basic ideas inside our computation model based on multiple processes. In section 3, the $FILL$ fragment is presented [16]. Unlike intuitionistic linear logic (ILL), it includes the multiplication disjunction *par*, denoted here $\Box$ and it is formalized in a sequent calculus with multiple conclusions and a familiar notation for some kind of parallel process. In section 4, we analyze the dynamics of $FILL$ through proof search and also its potentialities for non-determinisms managements and thus concurrent programming. After this analysis of the *proofs-as-computations* approach, we consider it, in section 5, in a rather different way, i.e., *proof-search as computation*. Thus we propose a new framework, based on $FILL$, that can be viewed as an extension of CPL [31] having some common features with ACL [18]. The concurrent computations are described in terms of proof construction from a process calculus, with the logical representation of concepts as communication, parallel composition. The logical system, corresponding to this calculus, can be obtained after applying our proof normalization process [11]. In section 6, we illustrate one main point, compared to other proposals, that is the introduction at the logical level of non-determinisms with the $\Box$ operator. Within this system, we are able to modelize some important concepts of concurrent object programming. The framework allows the representation of rich mechanisms for concurrent object programming that are described in a pure logical form and exhibit attractive features as a concurrent programming languages providing mechanisms for object-oriented concurrent computing (inheritance, information sharing, identifier creation). In section 7, we show that our framework allows also to model systems that can dynamically modify their state, in the perspective of *proofs-as-computations* and we also illustrate the ability to represent objects and methods in a similar way as in other frameworks [7]. To complete our work and to emphasize its interest for concurrent object programming, we consider this paradigm with extended non-deterministic Petri nets and prove that the reachability problem between two states in such a net corresponds to the provability in a fragment of $FILL$.

2 Linear logic and concurrent computations

Linear Logic (LL) is recognized as relevant for computation issues, especially concurrency and state change [2]. We briefly outline the main characteristics of LL, the formal system of which being in appendix A. From the sequent calculus point of view, LL can be seen as a substructural logic where the *contraction* and *weakening* rules are discarded in order that each assumption must be spent exactly once and thus considered as computational *resource*. With the lack of these rules we have two conjunctions $\otimes$ (*times*) and & (*with*) and two disjunctions $\wp$ (*par*) and $\oplus$ (*plus*).

2.1 Modeling concurrent computations in LL

Let us analyze the operators from the viewpoint of concurrent computation. The $\otimes$ rule means that a process connected with $\otimes$ can be decomposed into two parallel processes the environments of which are different from one another. Moreover, the & rule means that a process connected with & can be executed alternatively. LL is also characterized by a new implication $\multimap$, called *linear implication* such that $A \multimap B$ means that if we have A we can obtain B by consuming A. The modalities ! and ? allow to reintroduce the notions of weakening and contraction, but in a finer way. The cut-elimination in the *cut* rule can be seen as a communication between a process with A and a process with $A^{\perp}$. Thus, we can have in mind that linear logic (and its fragments) can naturally deal with properties in concurrent and distributed systems that other logical frameworks cannot represent as computational resources, communication, state transitions and concurrent execution.

There are several major approaches to model concurrent computation using fragments of linear logic. One is the approach based on *formulas-as-types* and *proofs-as-programs* paradigms in which propositions are interpreted as types, proofs as programs and proof normalization process as computation [1, 6, 23] with proposals of concurrent functional programming languages. Another approach is the one based on *formulas-as-states* and *proofs-as-computations* [22, 27] where the connections with Petri nets and Linear Logic have been investigated. It is also investigated in a rather different way in the context of logic programming [3]. Let us mention also works on linear logic programming [14], concurrent linear logic programming [18, 20] and object representation in Forum [7] that are based on *proof-search as computation* programming paradigm. Our proposal focuses an the two latter approaches for modeling concurrent computation.

2.2 Main concepts for concurrent programming

Concurrent programming languages based on asynchronous communication would become more important for massively parallel processing environments and programmers need formal frameworks for concurrent computation that can help them

to transform, verify and prove concurrent programs. In such a framework, computation is described in terms of proof construction in LL, knowing that inference rules and formulas in LL are restricted so that restricted rules have a proof power equivalent to the original rules for the restricted formulas. Moreover, the mechanisms for concurrent computations are mainly described in a purely logical form.
Let us recall some simple points about linear logic and concurrent programming. More details can be obtained in [20]. We consider a computational model where multiple processes perform computation and communicate with one another via asynchronous message passing. A message disappears after having been read by a process and becomes then consumable resource. That is why it is natural to represent a message by a formula (atomic) of linear logic. Let us consider now the processes. If A is a process waiting for a message M and behaves like B after reception, A consumes M and produces B and hence A is interpreted by a linear logic implication $M \multimap B$. Then a process is represented by a formula of linear logic. Consumption of a message M by a process $M \multimap B$ is represented by the following deduction $M, M \multimap P, C \vdash P, C$ where C can be considered as other processes, messages or as an environment.

Remark. We cannot interpret it with connectives of classical logic. For example, in classical logic, we can deduce $M, M \Rightarrow B \vdash B$ but also $M, M \Rightarrow B \vdash M, M \Rightarrow B, B$ which implies that the original message M and the process $M \Rightarrow B$ may remain after the message reception. Another point here is that M is undependent of any sender and thus the communication is asynchronous. In fact, if a message has been synchronous it is consumed when it is produced.

Let us try to interpret another connectives of LL. $P \otimes Q$ means that we have both a process P and a process Q at the same time, i.e., two concurrent processes and thus $\otimes$ represents a concurrent computation. If we consider $M_1 \multimap A_1 \& M_2 \multimap A_2$, we have one of the $M_i \multimap A_i$ but not both at the same time. But even, if we have both M_1 and M_2, we can only obtain either A_1 or A_2. If we consider a predicate logic, knowing that an atomic formula $M(a)$ can be interpreted as a message carrying a value a and a predicate M as a communication channel, what does $\forall x.(M(x) \multimap A(x))$ mean ? In fact, it represents a process which receives a value of x via a message M and becomes $A(x)$. Moreover, $\exists x.A$ represents the possibility of hiding x that can be used as a private name. Moreover, there are some variants to express communication. For example, $\forall x.(M(a, x) \multimap A(x))$ receives only a message M the first argument of which matching with a and thus we can realize generative communication as in Linda.
Now, let us consider the link between proofs and process reductions. It is not always possible to interpret a provable sequent as the result of a process reduction. For example, $M_1 \multimap (M_2 \multimap P) \vdash M_2 \multimap (M_1 \multimap P)$ is provable but cannot be viewed as a reduction process. In some cases, it appears possible to characterize the sequents corresponding to a process reduction as the provable ones only with left rules (in the conjunctive case). To obtain the correspondence one can propose some choices concerning limitations of the rules application [21, 28].

3 FILL: Full Intuitionistic Linear Logic

Full Intuitionistic Linear Logic ($FILL$), unlike intuitionistic linear logic (ILL), includes the multiplication disjunction *par*. Some problems arise from the interaction between *par* and *linear implication* but a term assignment system gives an interpretation of proofs as some kind of non-deterministic function. The system does enjoy cut elimination property and is a direct result of an analysis of the categorical semantics [16].

3.1 The proof system

As soon as we try to incorporate the multiplicative disjunction *par* into the logic, we are forced to consider traditional sequents with many hypotheses and many conclusions. In $FILL$ *par* is denoted $\Box$ because it is different of $\wp$ in LL. It is formalized in a sequent calculus with multiple conclusions; the comma on the right-hand side of the turnstile as a vertical bar, that is a familiar notation for some kind of parallel process.
It is not easy to give a satisfactory account of the computational meaning of the proofs or term calculus for $FILL$ and to give an interpretation in terms of processes. The innovation of the system is more to introduce, considering sequents with many hypotheses and many conclusions, a term assignment system and a term calculus with good properties in such circumstances [16].
The rules of the proof system are the following:

$$\frac{}{A \vdash A}\ (Id) \qquad \frac{\Gamma \vdash A \mid \Delta \quad A, \Gamma' \vdash B}{\Gamma, \Gamma' \vdash \Delta \mid B}\ (Cut)$$

$$\frac{\Gamma, A, B, \Gamma' \vdash \Delta}{\Gamma, B, A, \Gamma' \vdash \Delta}\ (Ex_L) \qquad \frac{\Gamma \vdash \Delta \mid A \mid B \mid \Delta'}{\Gamma \vdash \Delta \mid B \mid A \mid \Delta'}\ (Ex_R)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, I \vdash \Delta}\ (I_L) \qquad \frac{}{\vdash I}\ (I_R) \qquad \frac{}{\bot \vdash}\ (\bot_L) \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \bot \mid \Delta}\ (\bot_R)$$

$$\frac{\Gamma, A, B \vdash C}{\Gamma, A \otimes B \vdash C}\ (\otimes_L) \qquad \frac{\Gamma \vdash A \mid \Delta \quad \Gamma' \vdash B \mid \Delta'}{\Gamma, \Gamma' \vdash A \otimes B \mid \Delta \mid \Delta'}\ (\otimes_R)$$

$$\frac{\Gamma, A \vdash C \quad \Gamma', B \vdash D}{\Gamma, \Gamma', A \Box B \vdash C \mid D}\ (\Box_L) \qquad \frac{\Gamma \vdash \Delta \mid A \mid B \mid \Delta'}{\Gamma \vdash \Delta \mid A \Box B \mid \Delta'}\ (\Box_R)$$

$$\frac{\Gamma \vdash A \mid \Delta \quad \Delta', B \vdash C}{\Gamma, A \multimap B, \Delta' \vdash C \mid \Delta}\ (\multimap_L) \qquad \frac{\Gamma, A \vdash B \mid \Delta}{\Gamma \vdash A \multimap B \mid \Delta}\ (\multimap_R)$$

$$\frac{\Gamma, A \vdash \Delta \quad \Gamma, B \vdash \Delta}{\Gamma, A \oplus B \vdash \Delta}\ (\oplus_L) \qquad \frac{\Gamma \vdash A \mid \Delta}{\Gamma \vdash A \oplus B \mid \Delta}\ (\oplus_{R1}) \qquad \frac{\Gamma \vdash B \mid \Delta}{\Gamma \vdash A \oplus B \mid \Delta}\ (\oplus_{R2})$$

$$\frac{\Gamma, A \vdash \Delta}{\Gamma, A \& B \vdash \Delta}\ (\&_{L1}) \qquad \frac{\Gamma, B \vdash \Delta}{\Gamma, A \& B \vdash \Delta}\ (\&_{L2}) \qquad \frac{\Gamma \vdash A \mid \Delta \quad \Gamma \vdash B \mid \Delta'}{\Gamma \vdash A \& B \mid \Delta \mid \Delta'}\ (\&_R)$$

$$\frac{\Gamma \vdash \Delta}{\Gamma, !B \vdash \Delta}\ (!w_L) \quad \frac{\Gamma, A \vdash \Delta}{\Gamma, !A \vdash \Delta}\ (!D_L) \quad \frac{\Gamma, !A \vdash \Delta}{\Gamma, !A, !A \vdash \Delta}\ (!c_L) \quad \frac{!\Gamma \vdash C \mid \Delta}{!\Gamma \vdash !C \mid \Delta}\ (!S_R)$$

3.2 Proofs and programming in FILL

Therefore, intuitively, it seems natural to investigate this logic as a model for concurrent programming based on logic. Considering the term calculus (we have not presented here) for $FILL$, it remains to give a satisfactory account of its computational meaning and also to give an interpretation in terms of processes. Our goal, in this paper, is to consider it as logical foundation for a framework to involve object-oriented, concurrent and logic programming and to illustrate its ability to represent non-determinisms at different levels (messages, process, proof search) in the context of concurrent systems specification.
From the concurrent programming point of view, we consider two complementary directions for this study based on proof search results in $FILL$, namely following the two main paradigms *proof-search as computation* and *proofs-as-computations*. In both cases, proof search management in linear logic is central as in other approaches on concurrency [18, 27] or logic (concurrent) programming [3, 13, 18]. Here, we can use a specialization of our results on proof search in fragments of LL [10, 9, 11] dedicated to $FILL$.

4 On dynamics and proof search in FILL

It is well known that Linear Logic provides us with natural encodings of based-on state transitions systems, for instance extended Petri nets [17, 27]. One main point consists in proving an equivalence between the reachability problem for such nets and the derivability problem in the corresponding LL fragment.
The classical net definition does not describe some dynamic aspects of the behavior that will be understandable through the encoding and the proof search in $FILL$ [16]. For example, we can consider that there is only one token in a net but we do not know its place, namely A or B. Thus we will introduce a "don't know" non-determinism according to the place of the token that we naturally represent in our logic. The classical token representation and behavior will be still available. To start, we illustrate the dynamics in $FILL$ through the informal definition of extended Petri nets, called $NDTN$, focusing on the proof search process.

Definition 1. In a $NDTN$ each place p_i from S will be associated with a literal p_i with the following grammar $p_i ::= a \mid p_i \otimes p_i \mid p_i \Box p_i$, a being an atomic formula.
The static structure of a $NDTN$, i.e. the *transitions*, is represented by a set of proper axioms of the form $t ::= !(p \multimap p)$ (or $p \vdash p$ if we use the formal system with the cut rule). The tokens (*resources*) of a $NDTN$ are represented by expressions of the grammar $Tk ::= tk \mid Tk \Box Tk \mid Tk \otimes Tk$, tk being an atomic formula.

The dynamic aspects of such a net directly appears through the proof-search process into the $FILL$ system. A first example consists in considering a transition t: $!(A \Box B \multimap C)$, one token $A \Box B$ and to produce C. The token means that we have a token in A or a token in B but we do not know it. In fact, with the following proof of the sequent $!(A \Box B \multimap C), A \Box B \vdash C$,

$$
\begin{prooftree}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$A \vdash A$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$B \vdash B$}
\RightLabel{$\Box_L$}
\BinaryInfC{$A \Box B \vdash A | B$}
\RightLabel{$\Box_R$}
\UnaryInfC{$A \Box B \vdash A \Box B$}
\AxiomC{}
\RightLabel{ax}
\UnaryInfC{$C \vdash C$}
\RightLabel{$\multimap_L$}
\BinaryInfC{$A \Box B \multimap C, A \Box B \vdash C$}
\RightLabel{$!D_L$}
\UnaryInfC{$!(A \Box B \multimap C), A \Box B \vdash C$}
\end{prooftree}
$$

we observe that the transition t fires to produce a token at place C if and only if there exists a token at place A or a token at place B but not both at the same time. It means that only one token exists for two places A,B. The proof represents the behavior of the system including this non-determinism.

Let us consider a more specific (and significant) example to illustrate the interest of this logic to specify or modelize the behavior of concurrent systems. We study the following instance of philosophers problem in the case of *two* philosophers and *one* fork. A philosopher can eat $(Ea_i, i = 1, 2)$ only if he gets a fork to his left hand $(Hl_i, i = 1, 2)$ otherwise he gives the fork to the other philosopher in one of its hands $(Hr_i, Hl_i, i = 1, 2)$ but we cannot know the which one. When a philosopher eats, he puts the fork down to the left or right hand-side randomly.
In this problem, we have a specific non-determinism because we do not know where the fork is and how one philosopher gives the fork to the other. We can use a $NDTN$ net to represent this problem, its translation into clauses in $FILL$ and the proof search to model such situations. The set T of transitions (as proper axioms of the linear theory L_T) is composed by: $Hl_1 \vdash Ea_1$, $Ea_1 \vdash Hr_1 \Box Hl_1$, $Hr_1 \vdash Hl_2 \Box Hr_2$, $Hl_2 \vdash Ea_2$, $Ea_2 \vdash Hl_2 \Box Hr_2$, $Hr_2 \vdash Hr_1 \Box Hl_1$.

Remark. Here, we use transitions expressed in a system including the cut rule, but the general form $P \vdash Q$ can be replaced by the expression (or clause) $!(P \multimap Q)$ in a system without cut rule.

In fact, the proof encodes the behavior and the evolution of this system we can consider, for example, the $Hr_1 \Box Hl_2$ formula as initialization, that means that the fork is in the right hand of the philosopher 1 or in the left hand of the philosopher 2. We have here a non-determinism and thus the system can change in different ways that will correspond to different proofs of the sequent $Hr_1 \Box Hl_2 \vdash \Sigma$ with Σ non-instanciated. They could help us to understand what could happen for this system and its behavior ? In fact we will observe that α) a philosopher can eat for ever, β) the philosophers can take and put down the fork and never eat and also γ) they can eat one after the other.

Remark. The proof is never ending. Because here we have modelize the system without constraints and this (concrete) problem is without end in reality. The proof construction is an actual representation of what happen in the system.

Let us build a proof Π of $Hr_1 \Box Hl_2 \vdash \Sigma$, with Σ being instanciated at each step of the proof construction process. Thus if we consider the two subproofs Π_1 and Π_2 defined by

$\Pi_1 \equiv$

$$\dfrac{\dfrac{}{Hl_2 \vdash \Gamma_1}\,ax \qquad \dfrac{\dfrac{}{Hr_2 \vdash Hr_1 \Box Hl_1}\,ax \qquad \dfrac{\dfrac{}{Hr_1 \vdash \Delta_1} \qquad \dfrac{\dfrac{}{Hl_1 \vdash Ea_1}\,ax \qquad \dfrac{}{Ea_1 \vdash \Delta_2}\,ax}{Hl_1 \vdash \Delta_2}\,cut}{Hr_1 \Box Hl_1 \vdash \Gamma_2}\,\Box_L}{Hr_2 \vdash \Gamma_2}\,cut}{Hl_2 \Box Hr_2 \vdash \Sigma_1}\,\Box_L$$

and $\Pi_2 \equiv$

$$\dfrac{\dfrac{}{Hl_2 \vdash Ea_2}\,ax \qquad \dfrac{}{Ea_2 \vdash \Sigma_2}\,ax}{Hl_2 \vdash \Sigma_2}\,cut$$

we obtain $\Pi \equiv$

$$\dfrac{\dfrac{\dfrac{}{Hr_1 \vdash Hl_2 \Box Hr_2}\,ax \qquad \Pi_1}{Hr_1 \vdash \Sigma_1}\,cut \qquad \Pi_2}{Hr_1 \Box Hl_2 \vdash \Sigma_1 | \Sigma_2}\,\Box_L$$

Here ax corresponds to a proper axiom in the theory issued of the translation into $FILL$.
In fact, we have $\Sigma \equiv \Sigma_1|\Sigma_2$, $\Sigma_1 \equiv \Gamma_1|\Gamma_2$ and $\Gamma_2 \equiv \Delta_1|\Delta_2$.
In this proof, we have made some choices. For example, we have instanciated Γ_1 with Hl_2. It is a possible choice but we could choose Ea_2 that corresponds to the fourth transition (or proper axioms). In the same way, we have fixed some choices that instantiate Δ_1 with Hr_1, Δ_2 with Ea_1 and Σ_2 with Ea_2. Therefore, from these choices, we have instanciated Σ as $\Sigma \equiv Hl_2|Hr_1|Ea_1|Ea_2$.

Another proof is possible with a cut on Ea_2. In fact, we can claim that a proof encodes a process in several states (depending on the branches of the proof). Let us remark that the instantiation is a ramdom choice and thus we could go on infinitely in a branch of a proof. But with adapted instantiation we can stop the evolution of the system encoded into $FILL$. If we look on the branches of the proof, the system has terminated in one of the four following configurations: a) the fork is in the left hand of philosopher 2, b) the fork is in the right hand of philosopher 1, c) the philosopher 1 eats and d) the philosopher 2 eats. Moreover, we are able to recover the way leading to one of these states by following the adequate branch in the proof. In fact, we could build the proof of the sequent $Hr_1 \Box Hl_2 \vdash Ea_2|\Sigma_2$ if we have in mind to force the construction of a proof branch encoding the process that leads the philosopher 2 to eat.

5 A Concurrent Logic Programming language based on FILL

We consider here the connection between Linear Logic and concurrent programming based on the *proof search as computation* paradigm. In such approach, computation is regarded as a controlled deduction in the underlying logic and several linear logic languages were proposed as refinements of logic programming [13, 14] or as new concurrent programming languages [18]. The concurrent linear logic programming we propose naturally models message-passing style asynchronous communication, with messages and processes represented in terms of linear logic formulas.

5.1 Important choices for the design

If we want to propose a concurrent logic programming language, we can use directly our results on proof search to design an interpretor of such a language as an automated theorem prover as in [4, 14, 33]. Another point consists in modelizing concurrency in a logical framework with a process calculus where formulas are considered as processes and proof search (or construction) as the process reduction [19, 28].

In fact, an important point is the choice of the *logical fragment* where we need, at the same time, efficient proof search strategies and its ability to modelize interesting systems. For example, some logical connectors cannot be used sometimes because they have no interpretation in a given process calculus. Let us recall that, for a given logical fragment of LL, we have a general method to normalize proofs [11] and to design proof search strategies [10]. For our purpose, we need to find a compromise between the modelization ability and the efficiency of the execution into the framework. The former depends on the syntax of the formulas chosen to express clauses and goals and the latter on the degree of normalization we want, knowing that we want to keep non-determinisms for execution.
Some works have proposed adequate frameworks depending on the linear logical fragments (into CLL or ILL) and on specific proof forms (uniform proofs or canonical proofs [13, 14]). One main point in our proposal consists in understanding how to interpret proof construction as a process reduction and thus to propose adequate calculi for this purpose. Such proposal is based on two complementary aspects: to propose the syntactic form (of formulas or sequents) for the interpretation of the state changes of the process, to normalize proofs in this fragment to propose a new formulation of the deductive system. In this paper, one of the goals consists in choosing $FILL$ as underlying logic to propose an approach for concurrent object programming in a logical point of view.

The interesting idea consists in considering the *formulas as processes* and the logical operations on these as algebraic operations on processes [19, 28, 35, 36]. D. Miller [28] has proposed two possibilities to compare the interpretation of linear sequents with process reduction. "P reduces to Q" can be translated as "the sequent $P \vdash Q$ is provable" (conjunctive interpretation) or as "the sequent $Q \vdash P$ is provable" (disjunctive interpretation). Thus, the representation of the operations on processes depends on the previous choice. For example, in various works, the parallel composition is represented by $\otimes$ with the first interpretation and by $\wp$ with the second one. From this choice of interpretation, it is important to focus on the representation of communication. We can decide to represent communication into or out the logic. In the latter case, we represent senders and receivers by using higher-order constants *send* or *get* that are extra-logical [28]. Moreover, in this case, the communication is asynchronous represented by a proper axiom. In our approach, as others authors, we prefer to work as much as possible in a *pure logical framework* where messages are specific formulas and linear implication ($\multimap$) represents the communication, having in mind the connections between linear logic and concurrent programming [20].

5.2 The process language

In this section, we define and analyze a new framework for concurrent computation based on linear logic, having an operational semantics described in terms of proof construction. It well captures concurrent computation based on asynchronous communication. Moreover, as other proposals like ACL or $HACL$ [20], it could provide with a new insight into other models of asynchronous concurrent computation form a logical point of view.
We define the grammar of the process and the messages (into $FILL$) of the language that is an extension of the system CPL [32] that was included in intuitionistic linear logic. Here, due to the $\Box$ connective, we are able to extend the grammar of messages and processes for representing some non-determinisms at the logical level.
The grammar to define the processes (P_r) (messages (M) and strict processes (P)) is the following

$$M ::= At|M\Box M|M \otimes M$$
$$P_r ::= M|P$$
$$P ::= I|0|M \otimes P|M \multimap P|P \otimes P|P\Box P|P\&P|\forall x P|\exists x P|!P$$

and we manipulate sequents with the form : $\Gamma \vdash P|\Sigma$ with Γ and Σ multisets of processes and P a process. Messages are considered as particular process to obtain a framework where communication is asynchronous. They are represented not only by atomic formulas as in [31] but also by formulas composed with $\Box$ and $\otimes$. The other processes, that are not messages are represented by compound formulas and the sequents are $FILL$ sequents.

5.3 Basic concepts

Let us first consider the problem of *communication*. A sender has the form $M \otimes P$, where M is the message to send and P the process activated after sending it. We will use another classical notation, from process algebras, i.e., $\underline{M}.P$. A receiver has the form $M \multimap P$ (notation: $M.P$) where M is the waited message and P the process after receipt. Furthermore, the sequent $M, M \multimap P \vdash P$ is provable in our framework and interpreted as the receipt of M by the process $M \multimap P$ that becomes P after receipt.
The operations on processes are represented by the logic connectives. $P \otimes P$ and $P\&P$ respectively represent the *parallel composition* and the *alternative choice*. Moreover, $\forall x.P$ allows the *generalization* over variables that allows communication with value passing and $\exists x.P$ allows restriction over variables, used for communication channel hiding. $!P$ represents the recursion operator which allows to manage recursive definitions of processes and the mobility in the processes structure. Finally, the constant I (unit element of $\otimes$) represents the termination process and 0 represent the abort process.

Remark. Concerning communication and operations on processes, the possible choices are similar to the ones of ACL [18] that are represented with the disjunctive form.

A main difference lies (also with Hcc [24]) on the use of the ! connector, not only for using a message several times but for expressing recursive definitions without extra logical constants.

As in other frameworks, we can represent different forms of communication. Except point-to-point communication as in the actor model, we can express the generative communication where several receivers can compete to receive a message. For instance the sequent $M, M \multimap P_1, M \multimap P_2 \vdash P_1 \otimes P_2$ is not provable because a message can only be read by one process. Broadcast communication, where a message can be read by any number of receivers, is possible as illustrated by the following provable sequent $!M,\ M \multimap P_1,\ M \multimap P_2 \vdash P_1 \otimes P_2$. Moreover, message passing and variable sharing communication can be expressed. More details about such representation can be found in [31]. Another point that arises is that it is not always possible to transfer processes in the communication. To solve it HACL [20] extends ACL in the sense that processes are parametrized by other processes but it could improve the modularity and the code reuse of concurrent programs.

5.4 The logical system

The syntax of the language being fixed, it determinates the logical fragment of FILL we have to consider. Having in mind to interpret proofs in terms of process reduction, we will apply our general method of proof normalization based on the study of permutability properties and on inference movements in a proof [11].
Starting from the permutability results for full LL we can specialized them to the rules involved in the previous sequents, i.e., $id, 0_L, I_L, \multimap_L, \otimes_L, \Box_L, \&_L, !_L, w!_L, c!_L, \forall_L, \exists_L, I_R, \multimap_R, \otimes_R, \Box_R, \&_R, !_R, \forall_R, \exists_R$ and thus to determinate the inference movements in a proof.
The inferences that can be moved up in a proof are $T_\uparrow = \{!D_L, !w_L, \multimap_L, \otimes_R, \exists_R, \Box_L\}$ and the ones that can be moved down in a proof are $T_\downarrow = \{I_L, \otimes_L, \exists_L, !c_L, \multimap_R, \&_R, \forall_R, \Box_R\}$. With these results we are able to normalize the proofs, knowing that a computation will be a bottom-up proof-search in the system.
Let us start to consider some inferences movements to define a new formal system. The $\multimap_L$ inferences (producing receivers) can be moved up to the inference introducing the active formula and thus we propose new rules obtained by merging these inferences. We can do the same with the $\otimes_R$ and then move up the weakening and contraction rules. Thus we obtain a new formal system equivalent to the original one.

1) Communication rules

$$\frac{M, P, \Gamma \vdash Q|\Sigma}{M \otimes P, \Gamma \vdash Q|\Sigma}\ S_L \qquad \frac{\Gamma \vdash Q|\Sigma}{M, \Gamma \vdash M \otimes Q|\Sigma}\ S_R$$

$$\frac{P, \Gamma \vdash Q|\Sigma}{M, M \multimap P, \Gamma \vdash Q|\Sigma}\ R_L \qquad \frac{M, \Gamma \vdash Q|\Sigma}{\Gamma \vdash M \multimap Q|\Sigma}\ R_R$$

2) Composition rules

$$\frac{P_1, P_2, \Gamma \vdash Q|\Sigma}{P_1 \otimes P_2, \Gamma \vdash Q|\Sigma}\ Par_L \qquad \frac{\Gamma_1, !\Gamma \vdash Q_1|\Sigma_1 \quad \Gamma_2, !\Gamma \vdash Q_2|\Sigma_2}{\Gamma_1, \Gamma_2, !\Gamma \vdash Q_1 \otimes Q_2|\Sigma_1|\Sigma_2}\ Par_R$$

$$\frac{\Gamma_1, !\Gamma \vdash P|\Sigma_1 \quad P, \Gamma_2, !\Gamma \vdash Q|\Sigma_2}{\Gamma_1, \Gamma_2, !\Gamma \vdash Q|\Sigma_1|\Sigma_2}\ Comp$$

$$\frac{P_1, \Gamma \vdash Q|\Sigma}{P_1 \& P_2, \Gamma \vdash Q|\Sigma}\ Alt_L \qquad \frac{P_2, \Gamma \vdash Q|\Sigma}{P_1 \& P_2, \Gamma \vdash Q|\Sigma}\ Alt_L \qquad \frac{\Gamma \vdash Q_1|\Sigma_1 \quad \Gamma, \vdash Q_2|\Sigma_2}{\Gamma \vdash Q_1 \& Q_2|\Sigma_1|\Sigma_2}\ Alt_R$$

3) Message selection rule

$$\frac{M_1, \Delta_1 \vdash \Gamma_1 \quad M_2, \Delta_2 \vdash \Gamma_2}{M_1 \Box M_2, \Delta_1, \Delta_2 \vdash \Gamma_1|\Gamma_2}\ Msel$$

4) Process selection rule

$$\frac{P_1, \Delta_1 \vdash \Gamma_1 \quad P_2, \Delta_2 \vdash \Gamma_2}{P_1 \Box P_2, \Delta_1, \Delta_2 \vdash \Gamma_1|\Gamma_2}\ Psel_L \qquad \frac{\Gamma \vdash Q_1|Q_2|\Sigma}{\Gamma \vdash Q_1 \Box Q_2|\Sigma}\ Psel_R$$

5) Recurrence, restriction, generalization rules

$$\frac{!P, P, \Gamma \vdash Q|\Sigma}{!P, \Gamma \vdash Q|\Sigma}\ rec_L \qquad \frac{!\Gamma \vdash Q|\Sigma}{!\Gamma \vdash !Q|\Sigma}\ rec_R$$

$$\frac{P[y/x], \Gamma \vdash Q|\Sigma}{\exists P, \Gamma \vdash Q|\Sigma}\ res_L \qquad \frac{\Gamma \vdash Q[t/x]|\Sigma}{\Gamma \vdash \exists x Q|\Sigma}\ res_R$$

$$\frac{P[t/x], \Gamma \vdash Q|\Sigma}{\forall x P, \Gamma \vdash Q|\Sigma}\ gen_L \qquad \frac{\Gamma \vdash Q[y/x]|\Sigma}{\Gamma \vdash \forall x Q|\Sigma}\ gen_R$$

6) Termination rules

$$\frac{\Gamma \vdash Q|\Sigma}{I, \Gamma \vdash Q|\Sigma}\ term_L \qquad \frac{}{!\Gamma \vdash I|\Sigma}\ term_R \qquad \frac{}{0, \Gamma \vdash P|\Sigma}\ brk$$

Let us recall the notation for a receiver and a sender, that is $\underline{M}.P$ for the sender $M \otimes P$ and $M.P$ for the receiver $M \multimap P$. Let us mention the fact that this system is correct and complete w.r.t. linear logic.

Theorem 2. *A sequent $\Gamma \vdash P|\Sigma$ is provable in this new system if and only if it is provable in the FILL system.*

5.5 Non-determinisms

The $\Box$ operator allows to introduce non determinisms during the specification of a process and thus to see the state of the same process in two different choices.
If we consider a sender of the form $\underline{M_1 \Box M_2 \Box M_3}.P$ then it corresponds to send the message M_1 or M_2 or M_3. If we consider the receiver $M_1 \Box M_2.P$ then P becomes active if it receives the message M_1 or M_2. In the same way, if we consider $M_3.Q$ as receiver, Q is active if it receives the message M_3. These possibilities characterize the extension of the messages grammar compared to CPL [31, 32] where the only possibility consists in having only a *and* between messages. In fact, processes can be activated by several different messages. Moreover, the interest of $FILL$ sequents of the form $\Sigma \vdash P|\Delta$ lies in the use of non consumable messages, Then we can force some activations with a particular message and Δ is used to collect the messages

that are not consumed; if we instantiate it with a given message at the beginning of the proof search then it means that we do not use effectively this message. In fact, an important point to mention is that a proof that contains n applications of the rule $\Box_L$ encodes $n+1$ executions of process.

Remark. A possible modification of this logical framework could be to consider $\Box$ as the operator of parallel composition. In this case, due to the duality rule between $\otimes$ and $\Box$, we could propose a parallel composition rule

$$\frac{P_1, \Gamma_1 \vdash \Delta_1 \quad P_2, \Gamma_2 \vdash \Delta_2}{P_1 \Box P_2, \Gamma_1, \Gamma_2 \vdash \Delta_1 | \Delta_2}\ Par_L$$

with sequents of the form $\Sigma \vdash P_1|P_2|....|P_n$, the succedent of which represents directly the parallel composition. Moreover, in this case, a rule Par_R would be not necessary.

6 On Objects and Concurrency

In this section, we illustrate the interest of this framework to modelize some important concurrent programming concepts and mainly focus on the ones involved in concurrent object-oriented style programming. In fact this framework has similar characteristics as the ACL [18] or CPL [31] languages but completed with non-determinisms management. Thus, we illustrate the interest of $FILL$ and the specific use of $\Box$ for this approach of concurrent object logic programming.

6.1 Concurrent Object-oriented style programming

We emphasize, in this part, the possibilities to represent in our logical framework objects and routines and also to modelize some features as for instance inheritance.

Objects and classes representation. Let us consider, at first, an example significant for object management [31].
An object of class *polygon* is represented by a message (atomic formula) denoted $(vertices\ :\ (id, [p_1, ..p_n]))$, the list of points p_i describing the position of polygon *id*. If we have other attributes, each one will be represented by an atomic formula connected by $\otimes$ to each other. The set of routines is represented by a process P_{rout} that is a parallel composition of recursive processes P_{r_i} that define the different procedures r_i. For instance, the process corresponding to the *translation* operation can be represented by the formula $P_{trans} \equiv \exists\ ad\ !\ \forall v\ id\ (trans : (v, id)).\forall x(vertices : (id, x)).\underline{(ad\ in : (x, v))}.\forall y(ad\ out : (x, v)\ y).\underline{(vertices : (id, y))}.I \otimes P_{ad}$, with P_{ad} being a process that adds a vector v to each vertex of x.
If we want to define the class *rectangle* from the previous class then an object of this new class inherits of the attributes of *polygone* with new proper attributes as *width* and *length*. Therefore, this new object is represented by the formula $(vertices\ :\ (id, [p_1, ..p_4])) \otimes (width\ :\ (id, x)) \otimes (length\ :\ (id, y))$. It inherits from the routines and one also can add other ones to P_{rout}, using the parallel composition.

Non-determinisms. Let us precise illustrate now the interest to use $FILL$ (and in fact the $\Box$ connector) as the underlying logic to modelize some non-determinisms in concurrent systems. For that, we consider the previous example (see section 4) about the dining philosophers; it can be described by defining the behavior of each philosopher and then by connecting them, if necessary, with another process.
The process named $philo\ n(fork)$ is represented by the formula
$P_{phi} \equiv \forall n\ f\ (fork\ in\ : (f,n)) \otimes (\ (hand\ in : (r,n))\ \Box\ (hand\ in : (l,n))) \multimap "eating"$.
With such a process we are able to specify that a philosopher is ambidextrous and consequently that he can eat if he has a fork in the right or left hand.
As we previously mentioned, the proof construction in the $FILL$ system will represent the two possible cases into a proof.
Because of this property (to be ambidextrous), we will denote this philosopher by $P_{ambphil}$. Now, a left-handed philosopher $P_{lh-phil}$ can be considered as a particular ambidextrous philosopher that would not be able to use its right hand to eat. Then such philosopher inherits some attributes of $P_{ambphil}$ but a message of the form (r,n) on the channel $(hand\ in)$ will not activate the process named $eating$.
In this case, we have to add a new attribute, using the parallel composition $\otimes$ connective, and then we can create a new object defined by the formula $P_{lh-phil} \equiv (left\_hander : n) \otimes P_{ambphil}$ from the initial one.
Here we are in the case of a generative point-to-point communication but several processes are competing for a message.
Let us notice that we could propose an abstract syntax with guide lines of the translation in linear logic adopting a more standard and programming syntax as in [18, 20] but we prefer to emphasize the logical foundations of such languages.

6.2 Messages reception

We can provide, as in ACL, a process represented by $M_1 \otimes M_2 .... \otimes M_n \otimes P$, which waits for messages $M_1, M_2, .., M_n$ and becomes P. It can receive the messages M_i in an arbitrary order ($\otimes$ is associative and commutative). Such mechanism, with multiple messages reception, is useful for the synchronization between processes. If we come back to our example about philosophers then we can build the process $(fork\ in :) \otimes (hand\ in\ :) \otimes P_{ph}$ from the initial one P_{ph}.
But we can also consider partial reception of message. A receiver of a message can start some computation immediately after receiving a part of a message. Such a mechanism enables us to exploit concurrency between senders and receivers. Going back to our example, the philosopher $P_{lh-phil}$ can start to eat when he has a fork and the message $(left\_hander : n)$ can arrive after.

6.3 Dynamic restructuring of processes

Computing environments may change while programs are running. Concurrent process should dynamically adjust themselves to such changes because optimal configuration of process in a certain environment may not be optimal in another environment. The possibility to express dynamic restructuring of processes is a main motivation of proposals such as Π-calculus [30]. As in ACL or CPL, it is possible to define a process that dynamically composes or decomposes other processes to adjust their granularity. It is a main interest of this proposal with programming concepts represented at the logical level.

6.4 Synchronization and sequencing between processes

In *HACL* [20], the authors have illustrated the use of higher-order processes, in particular in a similar example about philosophers. The problem is described by defining the behavior of each philosopher and then connecting them by a process. Moreover, a specific process, named *seq*, is needed for sequencing the execution in this system.
In our example, if we consider that, when a philosopher has finished to eat, he gives the fork to the other philosopher (that is also ambidextrous) then we have to introduce sequencing in its behavior description. In our viewpoint to represent operations on processes into the underlying logic, we could consider the sequentiality through a logical connective and consequently to adapt or extend our proposal and the results we have obtained. One possible way could be to study non-commutative linear logic or linear logic extended with a sequentiality connective as in [34]. We could easily consider this connective that allows to insert sequentiality at the logical level but it would lead to other new problems about the proof search techniques to develop. Further work will consider this possible extension.
Let us assume the existence of such a sequentiality connective denoted $\rhd$, we can define, in our example, a philosopher with the following process $philo\ n(fork)$ that is represented by
$P_{newphil} \equiv \forall n\ f\ (fork\ in\ : (f,n)) \otimes ((hand\ in : (r,n)) \Box\ (hand\ in : (l,n))) \multimap "eating"$
$\rhd\ "finish"$
$\rhd\ (fork\ out\ : (f, ano(n))) \otimes (hand\ out : (r, ano(n)) \Box\ (hand\ out : (l, ano(n)))$
with $ano(n)$ represents the philosopher that is not the philosopher n.
It means that a philosopher is ambidextrous, he eats if he has a fork in the right or left hand and then, when he has finished to eat, he gives the fork to the other philosopher that is also ambidextrous.
In future work, we will consider a process calculus extending the one we present here, including sequentiality and will compare with the works on higher-order processes [20]. A main interest will be to use the representation of the process operations in a pure logical framework that is the main idea inside this approach of concurrent object logic programming.

Remark. Another point to consider in the future will be the equivalence of processes in such framework based on asynchronous message passing. We could start with a proposal based on the notion of *interface* [32] or another one based on *bisimulation* (from π-calculus) as in [15].

In this section, we have defined the representation of objects, classes and naturally concurrent objects in our framework. Now, we can come back to the *proof-as-computation* paradigm by looking if we can represent also other forms of objects and classes definitions as in [7].

7 Proofs-as-computations perspective

To complete our approach of object-concurrent programming with FILL proof, let us come back to the *proof-as-computation* paradigm. A first point consists to analyze the problem of state modification looking if we are able to consider in our system the representation of a state into a FILL sequent as in [7, 29].

7.1 More on the dynamics

One of our goal is to model, using the above language, systems that can dynamically modify their state. In some other logical models of object-oriented features [4, 26] we have methods presented by logic clauses that are used to rewrite one state into a new one. In this way, based-on Forum frameworks [29] combine both higher-order logic and linear logic and emphasize the representation of objects [7] and the proofs-as-computations perspective. It is an attempt to fit a general notion of state into a LL sequent.

In fact we can also represent, in our framework, the basic concepts used in these proposals. In this way we can refine the structure of the initial sequent form $\Gamma \vdash P|\Sigma$ into $!\mathcal{P}, \mathcal{S}, \mathcal{M} \vdash \mathcal{P}'|\Sigma$ where $\mathcal{S}$ can be considered as the current state of the system, $!\mathcal{P}$ is a set of definitions, $\mathcal{M}$ is the multi-set of messages (pending activities of the system), we can pick up atoms in $\mathcal{S}$ and in $\mathcal{M}$ and thus to obtain, with communication rules, a new state and a new message.

In this context, objects, methods and classes can be represented as in [7], i.e., objects are atomic formulas, classes are universally quantified formulas and auxiliary operations are expressed by atomic formulas. For instance, if objects are atoms of the form $(id\ Attrs\ (\&M_i))$ then the methods M_i are defined as $\forall x.(id\ Attrs\ Ms) \otimes (id\ Head) \multimap (id'\ Attrs'\ Ms') \otimes Msg_1 ... \otimes Msg_n$ where Msg_i is a message.

As example of the representation of such approach in our framework, in a proofs-as-computations perspective, we consider how to simulate the behavior of imperative paradigms as in [7].

The program executing the assignment will be expressed by the formula
$P_{ass} \equiv \forall x\ y\ v\ w\ g.(assign\ x\ y\ g) \otimes (x\ v) \otimes (y\ w) \multimap g \otimes (x\ v) \otimes (y\ w)$.

The computation to swap the values of the variables x and y is given by the following proof, knowing that $\mathcal{S}$ is initially $(x\ 1) \otimes (y\ 2) \otimes (z\ 0)$ and a program of swapping x and y is $(assign\ z\ x\ (assign\ x\ y\ (assign\ y\ z\ I)))$,

$$\cfrac{\cfrac{\cfrac{\cfrac{\overline{!P_{ass}, I, (x\ 2), (y\ 1), (z\ 1) \vdash \Sigma}}{!P_{ass}, I \otimes (x\ 2) \otimes (y\ 1) \otimes (z\ 1) \vdash \Sigma}\ R_L}{!P_{ass}, assign(y\ z\ I) \otimes (x\ 2) \otimes (y\ 2) \otimes (z\ 1) \vdash \Sigma}\ R_L}{!P_{ass}, (assign\ x\ y\ (assign\ y\ z\ I)) \otimes (x\ 1) \otimes (y\ 2) \otimes (z\ 1) \vdash \Sigma}\ R_L}{!P_{ass}, (assign\ z\ x\ (assign\ x\ y\ (assign\ y\ z\ I))) \otimes (x\ 1) \otimes (y\ 2) \otimes (z\ 0) \vdash \Sigma}\ R_L$$

The form of P_{ass} forces the computation to be deterministic because we imposed sequentiality between assignments, using continuations as the last parameter g to *assign*. Without this goal passing, we can define a non-deterministic computation (different proofs with different final states). We have forced the sequentiality in a context of the interpretation of *proofs* as *computations* where one prove that there exists a sequence of transitions from an initial to a final state ($S_i \vdash S_f$).

7.2 FILL and extended nets

We have seen in a previous section the dynamics in the proof search in $FILL$ through the definition of extended Petri net, called $NDTN$. They are extension to this fragment of the Kanovich's proposal [17]. Thus, we can consider this logical fragment as a specification logic for concurrent processes from this point of view.
In this section, we briefly consider the equivalence between the reachability problem for a $NDTN$ net and the derivability problem in the $FILL$ logical fragment.
The reachability relation between two states S_1, S_n of a $NDTN$ net P is represented by the notion of provability of the sequent $[S_1] \vdash [S_n]$ in the linear theory $L_{[P]}$, $[P$ being the proper axioms (or clauses) that encode the transitions of P.
[.] is the function that associates to a state S_i of the net P a formula of the linear logic $[S_i]$ that encodes this state. We denote f this function $f(S_i) = [S_i]$.

Theorem 3. *Let π be a proof of the sequent $F_1 \vdash F_n$ in the linear theory L_T, there exists a $NDTN$ net P having $S_1 = f^{-1}(F_1)$ as initial state and $S_n = f^{-1}(F_n)$ as final state that is reachable from S_1 with $[P] = T$*

Proof. Let π be a proof of a sequent $F_1 \vdash F_n$ in the linear theory L_T, we proceed by structural induction considering the last inference I_l.

□

Theorem 4. (Completeness) *Let P be a $NDTN$ net, S_1 its initial state and S_n a state (not necessary final) such that S_n reachable from S_1, then the sequent $[S_1] \vdash [S_n]$ is provable in the linear theory $L_{[P]}$.*

Proof. By induction on the number of transitions.

□

A further work consists in analyzing deeply the relationship between this approach of proof-as-computation perspective and our based on $FILL$ framework.

8 Concluding remarks

Intuitively, it can appear natural to investigate Full Intuitionistic Linear Logic ($FILL$) as a model for concurrent logic programming, principally because it involves the representation of some non-determinisms. To validate this intuition, we have investigated the design of a logical framework for logic, concurrent object programming based on $FILL$, in order to grasp some non-deterministic cases that can occur within concurrent processes. Moreover, we have proposed a process calculus based on $FILL$ and the *proof search as computation* programming paradigm. This new framework, where computation is described in terms of proof construction, extended other proposals as ACL [18] or CPL system [31, 32]. It allows some non-determinisms in the message management and also to represent interesting mechanisms for concurrent programming that are described can be described in a pure logical form. In fact it exhibits attractive features as a concurrent programming languages providing mechanisms for object-oriented concurrent computing.

References

1. S. Abramsky. Computational interpretations of linear logic. *Theoretical Computer Science*, 111(1-2):3–58, 1993.
2. V. Alexiev. Applications of Linear Logic to Computation: An Overview. *Bulletin of the IGPL*, 2(1):77–107, 1994.
3. J.M. Andreoli and R. Pareschi. Logic programming with sequent systems: A linear logic approach. In *Int. Workshop on Extensions of Logic Programming, LNCS 475*, pages 1–30, Tübingen, Germany, December 1989.
4. J.M. Andreoli and R. Pareschi. Linear objects: logical processes with built-in inheritance. In *7th Conference on Logic Programming, MIT Press*, pages 495–510, Jerusalem, June 1990.
5. J.M. Andreoli, R. Pareschi, and M. Bourgois. Dynamic programming as multiagent programming. In *ECOOP'91 - Workshop on Object-Based Concurrent Computing, LNCS 612*, pages 163–176, Genova, July 1991. Springer-Verlag.
6. N. Benton, G. Bierman, V. de Paiva, and M. Hyland. A term calculus for intuitionistic linear logic. In *Int. Conference on Typed Lambda Calculi and Applications, LNCS 664*, pages 75–90, Utrecht, The Netherlands, March 1993.
7. G. Delzanno and M. Martelli. Objects in Forum. In *International Logic Programming Symposium, ILPS'95*, Portland, Oregon, December 1995.
8. D. Galmiche. Canonical proofs for linear logic programming frameworks. In *Workshop on Proof-theoretical extensions of logic programming*, Santa Margherita Ligure, Italy, June 1994.
9. D. Galmiche and E. Boudinet. Proof search for programming in intuitionistic linear logic. In *CADE-12 Workshop on Proof search in type-theoretic languages*, Nancy, France, June 1994.
10. D. Galmiche and G. Perrier. Foundations of proof search strategies design in linear logic. In *Logic at St Petersburg '94, Symposium on Logical Foundations of Computer Science, LNCS 813*, pages 101–113, St Petersburg, Russia, July 1994.
11. D. Galmiche and G. Perrier. On proof normalization in linear logic. *Theoretical Computer Science*, 135(1):67–110, 1994.
12. J.Y. Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
13. J. Harland and D. Pym. On resolution in fragments of classical linear logic. In *LPAR'92, International Conference on Logic Programming and Automated Reasoning, LNAI 624*, pages 30–41, St. Petersburg, Russia, July 1992.
14. J. Hodas and D. Miller. Logic programming in a fragment of intuitionistic linear logic. *Journal of Information and Computation*, 110:327–365, 1994.
15. K. Honda and M. Tokoro. On asynchronous communication semantics. In *ECOOP'91 - Workshop on Object-Based Concurrent Computing, LNCS 612*, pages 21–51, Genova, July 1991. Springer-Verlag.
16. M. Hyland and V. de Paiva. Full intuitionistic linear logic (extended abstract). *Annals of Pure and Applied Logic*, 64:273–291, 1993.
17. M. Kanovich. Petri nets, Horn programs, Linear logic and Vector games. In *Int. Symposium on Theoretical Aspects of Computer Software, TACS'94, LNCS 789*, pages 642–666, Sendai, Japan, April 1994.
18. N. Kobayashi and A. Yonezawa. ACL - a concurrent linear logic programming paradigm. In *Int. Symposium on Logic Programming*, pages 279–294, Vancouver, October 1993.
19. N. Kobayashi and A. Yonezawa. Asynchronous communication model based on linear logic. *Formal Aspects of Computing*, 3:279–294, 1994.

20. N. Kobayashi and A. Yonezawa. Higher-order concurrent linear logic programming. In *Int. Workshop on Theory and Practice of Parallel Programming, LNCS 907*, pages 137–166, Sendai, Japan, November 1994.
21. N. Kobayashi and A. Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA'94*, pages 31–45, 1994.
22. J. Lilius. High-level nets and linear logic. In *13th Int. Conference on Applications and Theory of Petri Nets, LNCS 616*, pages 310–327, Sheffield, UK, 1992.
23. P. Lincoln and J. Mitchell. Operational aspects of linear lambda calculus. In *7th IEEE Symposium on Logic in Computer Science*, pages 235–246, Santa-Cruz, California, 1992.
24. P. Lincoln and V. Saraswat. Higher order, linear concurrent constraint programming. Manuscript, July 1992.
25. J. Meseguer. A logical theory of concurrent objects. In *OOPSLA-ECOOP '90*, pages 101–115, Ottawa, October 1990. Sigplan Notices, (25), 10.
26. J. Meseguer. Rewriting as a unified model of concurrency. In *CONCUR'90, LNCS 458*, pages 384–400, Amsterdam, August 1990.
27. J. Meseguer and N. Marti-Oliet. From Petri nets to Linear Logic. *Math. Struct. in Comp. Science*, 1:69–101, 1991.
28. D. Miller. The π-calculus as a theory of linear logic: Preliminary results. In *Workshop on Extensions of Logic Programming, LNCS 660*, pages 242–265, 1992.
29. D. Miller. A multiple-conclusion meta-logic. In *9th IEEE Symposium on Logic in Computer Science*, pages 272–281, Paris, France, July 1994.
30. R. Milner. Functions as processes. *Math. Struct. in Comp. Science*, 2(2):119–146, 1992.
31. G. Perrier. Concurrent Programming in Linear Logic. Technical Report 95-R-052, CRIN-CNRS, Nancy, March 1995.
32. G. Perrier. A model of concurrency based on linear logic. In *Conference on Computer Science Logic, CSL'95*, Paderborn, Germany, 1995.
33. D. Pym and J. Harland. A uniform proof-theoretic investigation of linear logic programming. *Journal of Logic and Computation*, 4(2):175–207, 1994.
34. C. Retoré. Pomset logic: a non-commutative extension of classical linear logic. In *Conference on Computer Science Logic, CSL'95*, Paderborn, Germany, September 1995.
35. V.A. Saraswat. Concurrent constraint programming. In *7th ACM Symposium on Principles of Programming Languages*, pages 232–245, San Francisco, California, 1990.
36. P. Volpe. Concurrent logic programming as uniform linear proofs. In *Algebraic and Logic Programming, ALP'94, LNCS 850*, pages 133–149, Madrid, Spain, 1994.

A The Linear sequent calculus

The inference system we use is the classical linear sequent calculus where, a sequent is an expression $\Gamma \vdash \Delta$ where Γ and Δ are multisets of linear formulae, i.e, formulae built on the linear primitives $\mathbf{0}, \mathbf{1}, \bot, \top, ()^{\bot}, \otimes, \wp, \&, \oplus, \forall, \exists, ?, !$. For a complete presentation of this system we refer to [12].

1) *Identity Group*

$$\frac{}{\vdash A, A^{\bot}}\ Id \qquad \frac{\Gamma \vdash A, \Delta \quad A, \Lambda \vdash \Theta}{\Gamma, \Lambda \vdash \Delta, \Theta}\ cut$$

2) *Structural Group*

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

3) *Logical Group*

$$\frac{A, B, \Gamma \vdash \Delta}{A \otimes B, \Gamma \vdash \Delta}\ \otimes_L \qquad \frac{\Gamma \vdash \Delta, A \quad \Lambda \vdash \Theta, B}{\Gamma, \Lambda \vdash \Delta, \Theta, A \otimes B}\ \otimes_R \qquad \frac{A, \Gamma \vdash \Delta \quad B, \Lambda \vdash \Theta}{A \wp B, \Gamma, \Lambda \vdash \Delta, \Theta}\ \wp_L \qquad \frac{\Gamma \vdash \Delta, A, B}{\Gamma \vdash \Delta, A \wp B}\ \wp_R$$

$$\frac{}{\vdash \mathbf{1}}\ 1_R \qquad \frac{\Gamma \vdash \Delta}{\mathbf{1}, \Gamma \vdash \Delta}\ 1_L \qquad \frac{}{\bot \vdash}\ \bot_L \qquad \frac{\Gamma \vdash \Delta}{\Gamma \vdash \Delta, \bot}\ \bot_R$$

$$\frac{\Gamma \vdash \Delta, A \quad B, \Lambda \vdash \Theta}{A \multimap B, \Gamma, \Lambda \vdash \Delta, \Theta}\ \multimap_L \qquad \frac{A, \Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \multimap B}\ \multimap_R \qquad \frac{\Gamma \vdash \Delta, A}{A^{\bot}, \Gamma \vdash \Delta}\ \bot_L \qquad \frac{A, \Gamma \vdash \Delta}{\Gamma \vdash \Delta, A^{\bot}}\ \bot_R$$

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

$$\frac{\Gamma \vdash \Delta, A}{\Gamma \vdash \Delta, A \oplus B}\ \oplus_R \qquad \frac{\Gamma \vdash \Delta, B}{\Gamma \vdash \Delta, A \oplus B}\ \oplus_R \qquad \frac{}{\Gamma \vdash \Delta, \top}\ \top_R \qquad \frac{}{\mathbf{0}, \Gamma \vdash \Delta}\ 0_L$$

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

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