Non–Commutative **Log**ic Programming Language **NoCLog**

Rémi Baudot – rb@lipn.univ-paris13.fr

In order to increase expressiveness of logic programming languages, we propose to define a Logic Programming language *NoCLog* based on non–commutative logic, developed by Ruet [12], using focalization for resolution. The *Focalization* property, which deals with the inner properties of the logical connectives during proof search, was introduced by Andreoli in term of a proof–search algorithm for Linear Logic and extended to NL by [10] for the order varieties version and by the author in [6] for the series–parallel version, thus giving theorem provers for such a logic. As we consider the series–parallel presentation more appropriate for implementation issues, we extend the theorem prover given in [6] to a programming language by proposing three resolution algorithms as variants of focusing for NoCLog, such that each of them expresses different logical properties or algorithmic behaviours in concurrent applications.

Focalization deals with the inner properties of the logical connectives, during proof search, which can thus be classified into two groups, named polarities, generating two kinds of non–determinism:

the *negative* connectives	the *positive* connectives
$\lhd$, $\parr$, $\bot$, ?, $\top$, &	1, $\odot$, $\otimes$, !, $\oplus$

where the $\odot$ and the $\lhd$ are respectively the non–commutative versions of the (multiplicative conjunction) $\otimes$ and of the (multiplicative disjunction) $\parr$. The additives $\oplus$ (disjunction) and & (conjunction) do not have non–commutative versions.

The point is that the positive connectives introduce synchronous (meaningful) non-determinism while the negative ones introduce asynchronous (meaningless) non–determinism. In the following we call a positive (resp. negative) formula a formula whose topmost connective is positive (resp. negative). Similarly polarity can be extended to atomic formulae, such that the duality reverses the polarity. A focusing proof search can then be considered as an alternation of negative and positive steps, such that the negative must be decomposed as soon as possible. During a positive step a choice must be made on a positive formula to be decomposed, which will then be called the *focus*. The main key of the latter step, is that a selected focus must be *fully decomposed* and that any positive atom met during the decomposition leads to an axiom link.

An execution of a NoCLog program can be seen as a particular proof search for a given formula. A NoCLog query noted $< P, g >$, is composed of a set of methods P (the Program), a goal which directs the proof search (possibly infinite) and of a resolution method based on focusing. Note that non–commutative logic defined by Ruet [1] has an important advantage w.r.t. other logics dealing with non–commutativity: it extends linear logic, keeping resource management as an essential feature. The major improvement is obtained by associating a series–parallel order to a sequent, instead of describing sequents as multisets of formulae. Similarly, NoCLog goals have a series–parallel order associated during the execution of a NoCLog program.

The NoCLog syntax is defined in term of goals, as below, where X is a negative atom.

negative (asynchronous) goals	*positive* (synchronous) goals
$g ::= X \mid ?X \mid g \,\&\, g \mid g \parr g \mid g \lhd g \mid \top \mid \bot$	$G ::= X^{\bot} \mid g \mid !g \mid 1 \mid G \otimes G \mid G \oplus G \mid G \odot G$

The goals direct the evolution of states (proof) by successive calls to the methods. In NoCLog, methods are of the form "$HEAD$ o— $BODY$" where the $HEAD$ is used to make a substitution of the

matching negative atoms in the sequent by the $BODY$. There exist two kinds of methods in NoCLog as described below.

• The *positive methods* are single–headed. They are defined by: $[X \circ\!\!- G]$

• The *negative methods* are multi–headed, as they are composed of a set of formulae connected by $\parr$ connectors: $[X_1 \parr \ldots \parr X_n \circ\!\!- g]$

•Strong focusing algorithm

Following the original focusing algorithm, the use of a method must be directly followed by a positive step with the body of the method selected as the focus, so that we can use a particular method only if it is fully decomposable. By the way, it makes no distinction between the negative and positive methods, as bipoles corresponding to negative methods are included in the positive ones. The main problem of such an algorithm is that it makes no distinction between recursively identical steps of a program execution and that it implies a pre–evaluation of the decomposition feasability (related to ressource sharing).

•Weak focusing algorithm

Nevertheless we can consider a *weaker* version, where we can use either kind of methods whose head matches atomic negative formulae in the sequent, whithout any conditions on the decomposition feasability of the body (i.e. without selection of a focus). This process corresponds to the decomposition of the formulae composing the head of the method. Remark that the use of negative methods corresponds to a total decomposition of the method as the negatives introduced by the method must be immediately decomposed.

We can now describe the operational behaviour of these two kinds of methods, for both algorithms, by the following:

- if the sequent is *flat* (i.e. it contains only atomic negative formulae) we can apply (concurrently)
  - (**weak**) any matching instance of the two kinds of methods,
  - (**strong**) any positive method fully decomposable;
- we add a particular algorithmic[1] behaviour for the **weak** case, in order to distinguish between the two kinds of methods: if the sequent is *synchronous–flat* (i.e. it only contains positive non–selectable as focuses or atomic negative formulae), we can concurrently apply any matching instance of negative methods. This behaviour is introduced to represent waiting (or sharing) problems in concurrent and distributed applications.

We can also consider another version of focusing which helps not to distinguish between equivalent proofs (or executions) of concurrent programs by applying, in the strong focusing point of view, all methods which do not share the same ressources (and thus are fully decomposable) in parallel. We call this algorithm the **strong–strong focusing algorithm**.

Implementations in NoCLog (with strong and weak focusing), given by the author and Cancila in [4], applied to the dining philosopher problem shown that such a language implements concurrent behaviour logically and efficiently, giving rise to original spatial solutions due to non–commutative representation of the problem. Furthermore a major improvment of the expressing power of such a language is given by the weak focusing algorithm, with the combination of the positive and negative methods, that help to implements ressource or time sharing problems.

Contrarily to the strong and weak focusing algorithms, the strong–strong focusing algorithm has not already a dedicated sequent system, due to the structural complexity of contexts in NL. Nevertheless implementations on planification problems, for which it seems well suited, are in progress.

[1] i.e. non–logical.

References

[1] V.M. Abrusci and P. Ruet. Non-commutative logic 1: the multiplicative fragment. To appear in Annals Pure Appl. Logic, 1998.

[2] Jean-Marc Andreoli. *Proposal for a Synthesis of Logic and Object-Oriented Programming Paradigms.* PhD thesis, University of Paris VI, 1990.

[3] Jean-Marc Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[4] Rémi Baudot and Daniela Cancila. The dining philosopher problem as a benchmark for the comparison of concurrent logic languages. To appear, 2000.

[5] Rémi Baudot and Christophe Fouqueré. Sigma13 - A Theorem Prover for Non-commutative Logic. To appear, 1999.

[6] Rémi Baudot and Christophe Fouqueré. Theorem prover with focusing strategy in non-commutative logic. Technical Report 99-06, Laboratoire d'Informatique de l'Université Paris Nord, December 1999.

[7] E. W. Dijkstra. *Co-operating Sequential Processes in Programming Languages.* Academic Press, f. genyus edition, 1965.

[8] Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[9] Roberto Maieli and Jean-Marc Andreoli. Focusing and proof-nets in linear and non-commutative logic. In *LPAR'99*, 1999.

[10] Roberto Maieli and Paul Ruet. Focusing non-commutative sequent calculus: the multiplicative case. To appear, 2000.

[11] Dale Miller. A multiple-conclusion specification logic. *Theoretical Computer Science*, 165(1):201–232, 1996.

[12] P. Ruet. *Logique non-commutative et programmation concurrente par contraintes.* PhD thesis, Université Denis Diderot, Paris 7, 1997.