# Logic Programming Revisited

Christophe Fouqueré, Virgile Mogbil

**To cite this version:**

Christophe Fouqueré, Virgile Mogbil. Logic Programming Revisited. 2011. hal-00198805v2

**HAL Id: hal-00198805**

**https://hal.science/hal-00198805v2**

Preprint submitted on 19 Apr 2011

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Logic Programming revisited

Christophe Fouqueré and Virgile Mogbil

LIPN – UMR7030, CNRS – Université Paris 13
99 av. J-B Clément, F–93430 Villetaneuse, France
{christophe.fouquere,virgile.mogbil}@lipn.univ-paris13.fr

**Abstract.** The so famous Prolog paradigm is based on the refutation of a goal, i.e. inferring the empty clause by means of the *resolution* principle. Switching to sequent calculus, this reduces to a cut-free proof search of the empty formula using the program clauses and the negation of the goal as hypotheses. It has already been applied to full Linear Logic. In this survey, we switch from sequent calculus to graph computation by means of bipolar modules. We explain in which extent the use of such structures in the framework of Linear Logic helps to enlarge logic programming towards a distributed backward and forward computation paradigm.

## 1 Introduction

Relations between Linear Logic and computation were studied from 1990 [1]. In particular, Kanovich [11] studies fragments of Linear Logic, their interpretations in terms of computation, their expressive power and complexity issues. In these first works, computation is already interpreted as the provability of sequents with respect to specialized sequent calculi. In the most simple fragment, multiplicative linear Horn sequents are of the form $W, \Gamma \vdash Z$ where $W$ is a product of atomic formulae (the linear facts), $\Gamma$ is a multiset of Horn linear implications $W_1 \multimap W_2$ (the linear clauses), and $Z$ is a product (the goal to be proven). In the multiplicative !-Horn fragment, sequents are of the form $W, \Gamma, !\Delta \vdash Z$ where $\Delta$ is a multiset of Horn linear implications (the classical clauses). These interpretations, and their extensions to the additive case, give fruitful results for complexity purposes and first insights on computational aspects. Works done during the 90's, mainly by Andreoli, Miller and others ([15], [2]), show that logic programming may be revisited by constraining proofs in a logical way. The so famous Prolog paradigm is based on the refutation of a goal, i.e. inferring the empty clause by means of the *resolution* principle. Switching to sequent calculus, this reduces to a cut-free proof search of the empty formula using the program clauses and the negation of the goal as hypothesis. This has already been detailed in a few papers ([5], [14]). This switch allows for considering extensions of the traditional Horn fragment of classical logic, as it gives a theoretical framework for a large bunch of sequent calculi.

In fact, the fundamental concept underlying logic programming in the framework of sequent calculus is the search for *uniform proofs*. Recalling Bruscoli and

Guglielmi's definition [5], "a uniform proof is a cut-free proof in which each occurrence of a sequent whose right-hand side contains a non-atomic formula is the lower sequent of the inference rule that introduces its top-level connective. In other words, proof-search is a repetition of (i) decomposition of the goal i.e. left-hand side (ii) use of a left inference rule i.e. progression step". The *progression step* corresponds to the resolution principle in Prolog paradigm. In order to define an *abstract logic programming language* (ALPL), the challenge is then to specify a suitable sequent calculus for which each provable sequent has at least one uniform proof. The systematic use of this principle is then the core of a correct and complete operational semantics.

This approach has been successfully used and implemented for various fragments of classical logic and also in linear logic. In particular, Andreoli [2] shows how full linear logic may be considered as an ALPL. Furthermore, his investigations lead to display two fundamental properties of linear logic: *focalization* and *polarity*. Briefly speaking, he has shown that connectives may be split into two dual sets with respect to the synchronicity of the decomposition they induce on formulae: a *negative* connective may be decomposed immediately whereas a *positive* connective decomposition depends on the context. Hence, formulae may be naturally viewed as alternate positive and negative stages. Focalization refers to the fact that there should exist formulae in a provable sequent whose decomposition is done on the two levels: positive and negative stages of such a *focus* have to be fully decomposed ('must be' when finding uniform proofs).

In this paper we work on the representation of foundations of logic programming by viewing computations in terms of graph computations. In this way we go one step further towards a distributed computation paradigm. Indeed by nature, the sequent calculus paradigm limits focalization to only one formula at a time. However, linear logic admits a parallel representation of proofs by means of *proof structures* [9]. Proof structures are directed graphs whose vertices are labelled with formulae and freely built from links (or hyper-edges) reflecting logic connectives. *Proof nets* are a subset of proof structures corresponding to formulae provable in linear logic. Although a proof search is sequential in the framework of the sequent calculus, the characterization of proof nets among proof structures is given as a *global* necessary and sufficient condition on the graph, called *correctness criterion.* One of the simplest correctness criteria consists in defining some rewriting system that reduces connected components of the graph to be checked: a proof net should reduce to a specific normal form. This methodology has first been applied by Danos [6] to the Multiplicative Linear Logic (MLL) where the normal form is a single vertex. In [8], we define a polarized version of proof nets, called *bipolar modules*, and give adapted correctness criteria not only for MLL but also for the Multiplicative Exponential Linear Logic (MELL).

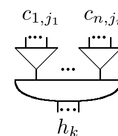
This paper is a survey of various logic programming models based on what is known in Linear Logic, from the simplest linear model to complex models that contain constraints. A novelty part ensures that all ingredients could be put together to achieve our goal. We show how clauses of a logic program may be represented as *elementary bipolar modules* (EBMs). We first consider com-

putation on elementary modules able to represent the progression principle as well as its dual part as a resource-conscious Prolog style ALPL. We prove that this fragment does not need any correctness criterion except unification. Mixing the previous situation and its dual one allows us to model disjunctive linear programs. Once again, no correctness criterion is necessary. We then extend it to MLL, hence including necessarily correctness criteria to deal with parallelism. In section 3, we present an extension where exponentials are allowed.

We finally explain why the use of specific graph structures (our bipolar modules) in the framework of Linear Logic helps to enlarge logic programming towards a distributed backward and forward computation paradigm. Three key concepts are required: reversibility of computation as contraposition of programs is safely available, desynchronization of the application of clauses and the check of correctness of these applications, and partial application of clauses as these two last points authorize distribution of the computation.

## 2 Multiplicative Linear Logic Programming

### 2.1 Basic Definitions




*Bipolar modules* (BMs) are specific graphs taking into account the polarized nature of linear logic. A BM is a directed graph with pending edges built on basic blocks called *elementary bipolar modules* (EBMs). EBMs are directed graphs with pending edges with one *positive pole* under a non empty finite set of *negative poles* (see opposite). We call *hypotheses* the propositional variables $h_k$ and *conclusions* the propositional variables $c_{i,j}$.[1] An EBM is *initial* (resp. *final*) if its set of hypotheses is empty (resp. its set of conclusions is empty). The construction of a BM is obtained by composing EBMs, linking hypotheses to conclusions of same labels. More precisely, let $M$ be an EBM, the associated formula is:

$$t(M) = \big(\bigotimes_{k\in K} h_k\big) \multimap \big(\parr_{i\in I}(\bigotimes_{j\in J} c_{i,j})\big)$$

An EBM may be interpreted as a clause in logic programming. As the formula associated to the composition of two EBMs corresponds to the tensor of the associated formulae of the EBMs, composition is interpreted as a pre-application of two clauses (this will be a rewriting step used for checking correctness). Then a BM viewed as the successive composition of EBMs, corresponds to a computation. When a BM is *closed*, i.e. without pending edges, computation ends. Otherwise, the bipolar module is called *open* and computation may eventually proceed. Note that the structure of a BM represents the computation history.

Checking labels during compositions is insufficient as a BM may not correspond to a (partial or not) valid computation: e.g. we do not want to accept

[1] We stick to the propositional case. This presentation could have been extended to the first-order case while replacing identity of variables by unification on terms.

that some piece of information (either hypothesis or conclusion) is used twice if it should be linear, or that two incompatible resources are consumed simultaneously to fire a clause! Correctness has obviously to be given with respect to the underlying fragment of Linear Logic to control the validity of the composition. A closed BM $M$ is correct if $t(M) \vdash$ is provable in the fragment considered so far, $t(M)$ being its associated formula. In the general case, a BM $M$ is correct if there exists a correct closed BM where $M$ is a subgraph. In other words, a correct BM corresponds to a partial computation for which continuations may be given satisfying the logic (but may be locked because available EBMs may be inadequate). In the next subsection, this framework is used to show that a linear[2] Prolog-like programming language may be easily modelled: linear Horn clauses are represented as "intuitionnistic" conjunctive EBMs. A first extension allows for expressing disjunction. For these two cases, checking labels during composition is sufficient to prove correctness. We end by taking into account MLL for which a particular correctness criterion has to be given.

### 2.2 Horn-like Linear Logic Programming: Resolution and Progression

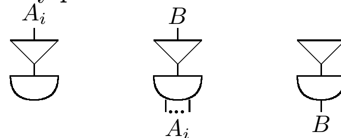
Let $\mathcal{U}$ be the following system given in a Prolog style on the left part, and in classical logic on the right part:

$$\begin{cases} A_i :- \\ B :- A_1, \ldots, A_n \\ \quad :- B \end{cases} \qquad \text{i.e.} \quad \begin{array}{r} \rightarrow A_i \\ \bigwedge_{i \in I} A_i \rightarrow B \\ B \rightarrow \end{array} \begin{array}{l} \left.\begin{array}{l} \\ \\ \end{array}\right\} \text{Program} \\ \} \quad \text{Goal} \end{array}$$

where $I \neq \emptyset$. $B$ is the goal to be proven, $A_i$ are facts[3]. We model in linear logic the system $\mathcal{U}$ supposing variables linear. This is in fact not an oversimplification if one remembers that SLD-resolution acts this way with goals and that reusable clauses are easily taken into account as explained in section 3. We then get the following set of Linear Logic formulae:

$$\begin{cases} 1 \multimap A_i \text{ for each } i \\ \bigotimes_{i \in I} A_i \multimap B \\ B \multimap \bot \end{cases}$$

Solving the goal $B$ is equivalent to proving the sequent $!\Gamma, B \multimap \bot \vdash$ where $\Gamma$ is the program (i.e. the set of previous formulae except the goal formula). The notation $!\Gamma$ stands for the set of $!C$ where $C \in \Gamma$. We could search equivalently for a uniform proof of $!\Gamma \vdash B$ in the !-Horn fragment of MLL. This set of formulae may be graphically presented as EBMs in the following way:




[2] Variables can only be used once.

[3] The first clause on the right as well as on the left is repeated for each subscript $i$.

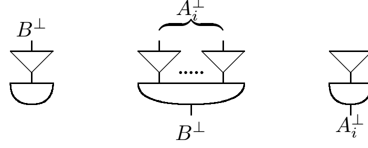
One resolution step corresponds to a top-down composition of modules, beginning with the one without any conclusion (the final one on the right). It is straightforward to notice that there is a correct closed module obtained this way from the final EBM in $\mathcal{U}$ iff there is a proof of $!\Gamma \vdash B$. This is the case for the whole Horn-like fragment. Note also that composition of modules is always correct in this fragment: there is no need for specific checks except identification of variables or first-order unification as it is already the case with resolution in Prolog. The programming language we get is an ALPL as far as the following strategy is used: we begin with an EBM without conclusions and only apply top-down composition of modules. This strategy is correct and complete with respect to the underlying sequent calculus, reminiscent of SLD-resolution:

$$\frac{}{!\Gamma \vdash 1} \qquad \frac{!\Gamma, !(A_1 \otimes \cdots \otimes A_n \multimap B) \vdash A_1, \ldots, A_n, C_1, \ldots, C_p}{!\Gamma, !(A_1 \otimes \cdots \otimes A_n \multimap B) \vdash B, C_1, \ldots, C_p}$$

We could obviously and safely base our proof search on a right introduction by contraposing it. Let $\mathcal{U}^{\mathcal{C}}$ be the contraposed system of $\mathcal{U}$:

$$\begin{cases} \neg A_i \rightarrow \\ \neg B \rightarrow \bigvee_{i\in I}(\neg A_i) \\ \quad \rightarrow \neg B \end{cases} \quad \text{i.e.} \quad \begin{cases} A_i^{\perp} \multimap \perp \\ B^{\perp} \multimap ⅋_{i\in I} A_i^{\perp} \\ 1 \multimap B^{\perp} \end{cases}$$

where $I \neq \emptyset$. The set of corresponding EBMs is the following one where each disjunctive $A_i^{\perp}$ is introduced by a negative pole:



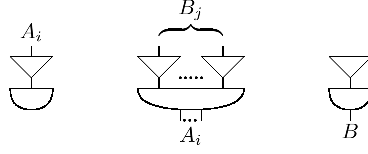

In this case the programming strategy reduces to a (bottom-up) composition of modules, starting with an EBM without any hypothesis. Note that only a unification check is required during the composition process. The strategy is correct and complete, and the system is equivalent to the previous one.

### 2.3 Disjunctive Resolution and Conjunctive Progression

The two (equivalent) previous presentations may be mixed by considering EBMs having multisets of (conjunctive) hypotheses as well as (disjunctive) conclusions. The programming system is no more Horn-like. Let $\mathcal{V}$ be the following system, expressed in the right as a multiset of linear logic formulae:

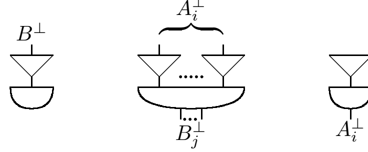
$$\begin{cases} \rightarrow A_i \\ \bigwedge_{i\in I} A_i \rightarrow \bigvee_{j\in J} B_j \\ B \rightarrow \end{cases} \quad \text{i.e.} \quad \begin{cases} 1 \multimap A_i \\ \bigotimes_{i\in I} A_i \multimap ⅋_{j\in J} B_j \\ B \multimap \perp \end{cases}$$

where $I, J \neq \emptyset$. The corresponding EBMs are the following ones:




Computation is still expressed as a composition of modules and uniform proofs are obtained with a top-down strategy. However this situation requires (a little) more attention with respect to correctness: Composition of modules is correct iff it does not induce cycles in the graph. This situation was first characterized by Andreoli et al. [3]. This is a degenerate case of the situation described in the following subsection. In the spirit of what was done in the previous subsection, one can contrapose the system $\mathcal{V}$, and EBMs are keeped in the same fragment:

$$\begin{cases} \neg A_i \rightarrow \\ \bigwedge_{j\in J} \neg B_j \rightarrow \bigvee_{i\in I} \neg A_i \\ \rightarrow \neg B \end{cases} \quad \text{i.e.} \quad \begin{cases} A_i^\perp \multimap \perp \\ \bigotimes_{j\in J} B_j^\perp \multimap \invamp_{i\in I} A_i^\perp \\ 1 \multimap B^\perp \end{cases}$$




Uniform proofs are now obtained with a bottom-up strategy and the acyclicity is as before required to get correct modules. The two presentations may be freely mixed: the proof construction is no more uniform as it corresponds to a forward computation in Prolog-like systems.

### 2.4 Generalized Resolution and Generalized Progression

The previous subsection shows that our framework may support dualization (it is an obvious consequence of the duality of Linear Logic). In order to simplify the presentation, we stick in the remainder to the classical viewpoint for clauses, i.e. backward computation, keeping in mind that forward computation is always available. The extension of the previous systems to the non-intuitionnistic (linear) case leads to a full treatment of MLL. In a first attempt, we consider the extension $\mathcal{W}$ of the system $\mathcal{U}^{\mathcal{C}}$ and its corresponding class of EBMs, the head of a clause is now given by a conjunctive set of disjunctive hypotheses:

$$\begin{cases} 1 \multimap A \\ A \multimap \invamp_{i\in I} \bigotimes_{j\in J} B_{i,j} \\ \bigotimes_{l\in L} B_l \multimap \perp \end{cases}$$

The acyclicity criterion given for previous cases is now insufficient. Defining a correctness criterion is made easier if one takes care of the focalization property [7]: A sequent is provable iff there exists a proof such that decomposition of the

positive stratum of formulae is done in one step. Considering our modules, it means that one can define a reduction relation such that each step reduces one positive-negative pair of nodes. The reduction relation $\twoheadrightarrow$ in Fig. 1 uses this fact. This system is confluent and terminates.

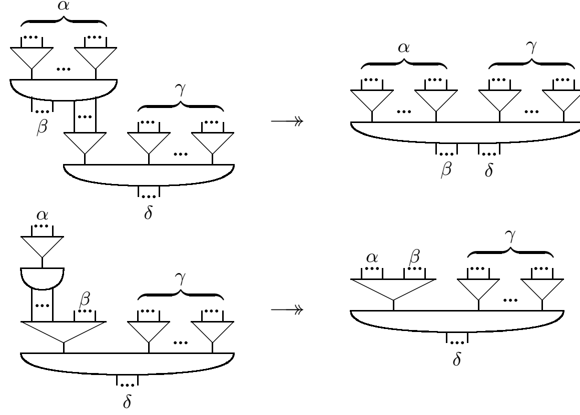



**Fig. 1.** Reduction relation.

**Theorem 1 (BM correctness [8]).** *A closed module $M$ is correct iff $M \twoheadrightarrow^*$*

Usual technique may be applied here to get a linear algorithm ([10]). Moreover the proof of the theorem needs only one reduction rule. Another proof can be given with the other reduction rule: each rule is sufficient and necessary! In other words, correctness is also checked with the same criterion in the contraposed system. Hence uniform proofs may be obtained either by a bottom-up or a top-down strategy generalizing respectively the progression and the resolution principles. We finally get the most general linear version by considering a generalized conjunctive system:

$$\begin{cases} 1 \multimap A \\ \bigotimes_{k \in K} A_k \multimap \bigparr_{i \in I} \bigotimes_{j \in J} B_{i,j} \\ \bigotimes_{l \in L} B_l \multimap \bot \end{cases}$$

## 3 Multiplicative Exponential Logic Programming

### 3.1 Multiplicative Exponential Linear Logic

Adding exponentials to the language obviously increases its expressivity: it allows for representing reusable resources as it is the case in classical logic. In linear logic, the modality 'of course' ! enjoys the main property: $!A \multimap A \otimes \cdots \otimes A$, in other words $!A$ being given, $A$ is usable as many times as necessary. Technically, three operations are required: *contraction*, *dereliction* and *weakening*. The first

operation states that $!A$ is duplicable: in our paradigm, an elementary module is used twice. Dereliction allows to consider the classical formula $!A$ as the linear one $A$. The last operation states that $!A$ may be forgotten: let us just permit the equivalent elementary module to be omitted. The dual modality 'why not' $?$ may be interpreted in the following way: as $A^\perp$ waits for a linear resource $A$, $?A^\perp$ waits for the 'classical' resource $!A$. This *promotion* operation is more complex. The introduction of a ? requires to have a 'classical' context: proof nets correctness holds if a 'box' delimits the context (which has itself to be correct). Entries of such a box are given by one ! and a set of ?.

**From Sequent Calculus to ?-EBMs.** The translation between formulas of Multiplicative Exponential Linear Logic (MELL) and modules is not so easy as it is without exponentials. We consider an extension of MELL with the neutral element 1 for $\otimes$. Formulae are built from the following grammar: $F := 1 \mid G$ where

$$G := A \mid A^\perp \mid G \otimes 1 \mid 1 \otimes G \mid G \otimes G \mid G \parr G \mid ?G \mid !G$$

Converting formulae to modules requires the use of polarization and focalization. Focalization allows to consider $n$-ary connectives as in the previous sections. Formulae are polarized negatively or positively according to their main connectives, considering conveniently that variables $A, B, \ldots$ are positive whereas their negations $A^\perp, B^\perp, \ldots$ are negative. If $A$ is a positive formula, $?A$ is negative whereas $!A^\perp$ is positive. Hence exponential connectives may be split into two parts: $!A^\perp = \downarrow\sharp A^\perp$ and $?A = \uparrow\flat A$. The shift connectives $\downarrow$ and $\uparrow$ change the polarities. The introduction of shift connectives may be generalized also to the linear case whenever there is a change of polarity. The two modalities $\flat$ and $\sharp$ express exponentiality.

We consider a slightly different version of a polarized sequent calculus as it was designed by Boudes [4] or Laurent [13]: the system $\text{LL}_{\texttt{pol}}$ given by Laurent takes care of multiplicative as well as additive connectives where atomic formulas are always exponentialized. Although his aim is to develop a polarized sequent calculus, the sequent calculus we give is a first step toward a polarized proof structure calculus. For the understanding we present a binary version in Fig. 2 but following our motivations our language ${}_n\text{MELL}_{\texttt{pol}}$ is $n$-ary, the decomposition of exponentials is explicit and atomic formulae may be linear or exponential. The grammar for ${}_n\text{MELL}_{\texttt{pol}}$ is given in the following way where the set of formulae is explicitly split into positive ($P$) and negative ($N$) formulae ($A$ is a positive atomic formula):

$$\begin{cases} P := \bigotimes_{i \in I} \rho_i \mid \flat(\bigotimes_{i \in I} \rho_i) \\ \rho := A \qquad\quad \mid \downarrow N \end{cases} \qquad \begin{cases} N := \parr_{k \in K} \nu_k \mid \sharp(\parr_{k \in K} \nu_k) \\ \nu := A^\perp \qquad\quad \mid \uparrow P \end{cases}$$

It is easy to define a sequent calculus for ${}_n\text{MELL}_{\texttt{pol}}$ from the binary sequent calculus (Fig. 2) and a translation $(-)^-$ from MELL to ${}_n\text{MELL}_{\texttt{pol}}$ such that if

$F$ is a MELL formula, $\vdash_{\texttt{MELL}} F$ is provable iff $\vdash_{n\texttt{MELL}_{\texttt{pol}}} F^-$ is provable: $\mathbf{1}^+ = \mathbf{1}$,

$$
\begin{array}{rlrl}
A^+ &= A & A^{\perp -} &= A^\perp \\
(F_1 \otimes F_2)^+ &= F_1^+ \otimes F_2^+ & (F_1 \parr F_2)^- &= F_1^- \parr F_2^- \\
(!F)^+ &= \downarrow\sharp F^- & (?F)^- &= \uparrow\flat F^+ \\
F^+ &= \downarrow F^- \text{otherwise} & F^- &= \uparrow F^+ \text{otherwise}
\end{array}
$$

$$
\frac{}{\vdash A, A^\perp}\ (axiom) \quad \frac{\vdash \Gamma, A \quad \vdash A^\perp, \Delta}{\vdash \Gamma, \Delta}\ (cut) \quad \frac{\vdash A, B, \Gamma}{\vdash A \parr B, \Gamma}\ (\parr) \quad \frac{\vdash \Gamma, A \quad \vdash \Delta, B}{\vdash \Gamma, A \otimes B, \Delta}\ (\otimes)
$$

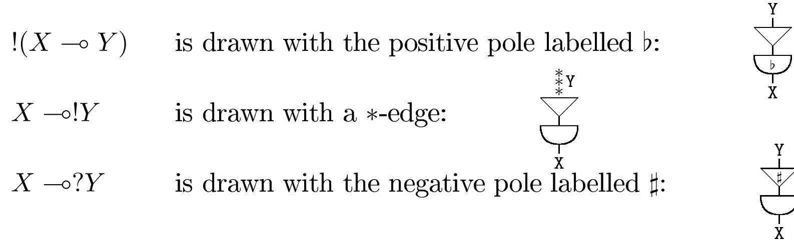
$$
\frac{\vdash \Gamma, P}{\vdash \Gamma, \uparrow P}\ (\uparrow) \quad \frac{\vdash \Gamma, N}{\vdash \Gamma, \downarrow N}\ (\downarrow)
$$

$$
\frac{\vdash \flat\Gamma, N}{\vdash \flat\Gamma, \sharp N}\ (\sharp p) \quad \frac{\vdash \Gamma, P}{\vdash \Gamma, \flat P}\ (\flat d) \quad \frac{\vdash \Gamma}{\vdash \Gamma, \flat P}\ (\flat w) \quad \frac{\vdash \Gamma, \flat P, \flat P}{\vdash \Gamma, \flat P}\ (\flat c)
$$

**Fig. 2.** Binary sequent calculus

Exponential rules are given in a standard way: dereliction ($\flat d$), weakening ($\flat w$) and contraction ($\flat c$) express the reusability of the underlying formula, promotion ($\sharp p$) is its dual constraining formulas in the context to be under the scope of a $\flat$ modality. To obtain bipolar module, we need to manage atomic formulae in exponential context: we add for each atomic formula an *exponential atomic formula* in the language (see [8] for details). This exponential atomic formula is noted with a $\sharp$ superscript: $A^\sharp$ is defined as $\downarrow\sharp\uparrow A$. We still preserve provability.

The general shape of a *?-EBMs* is given Fig. 3 in the middle. Positive and negative poles may now be labelled, hypotheses may be linear (as in previous sections) or exponential (links are noted with $*$).[4] A ?-EBM is reusable when $\flat$ labels its positive part, $\sharp$ labels a promoted negative pole that requires composition on top to be done only with reusable ?-EBMs as characterized in the rewriting system given in subsection 3.2. $*$ labels an exponential atomic negative conclusion of a ?-EBM and we refer to $*$-edge in that case. Roughly, the correspondence between places of exponentials in formulae and labelled elements is the following one:



$!(X \multimap Y)$ is drawn with the positive pole labelled $\flat$:

$X \multimap !Y$ is drawn with a $*$-edge:

$X \multimap ?Y$ is drawn with the negative pole labelled $\sharp$:

[4] A bracket means that the label is optional.

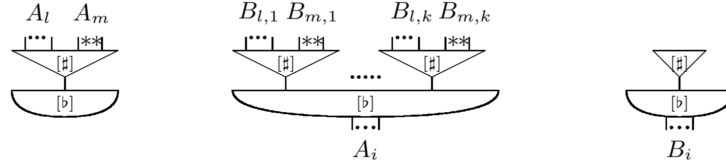



**Fig. 3.** ?-EBMs (initial, of general shape, final)

Using the convention that a 0-ary tensor is 1 and brackets mean optional, one gives the following system $\mathcal{T}$ corresponding to the set of ?-EBMs of Fig. 3 where the clause $\mathbb{C}$ of a ?-EBM generalizes the clause given for an EBM:

$$\left\{\begin{array}{c}[!](1 \multimap [?]\ (\bigotimes_{l\in L}\ A_l \bigotimes_{m\in M}\ A^{\sharp}_m)) \\ [!](\bigotimes_{i\in I}\ A_i \multimap \parr_{k\in K}[?]\ (\bigotimes_{l\in L}\ B_{l,k} \bigotimes_{m\in M}\ B^{\sharp}_{m,k})) = \mathbb{C} \\ [!](\bigotimes_{i\in I}\ B_i \multimap \bot)\end{array}\right.$$

Such a clause could be interpreted as: $\mathbb{C}$ is a reusable clause iff ! is explicit. The application of a clause is allowed if the $A_i$ are available, then one of the conclusions is fired, a conclusion being a multiset of atomic formulae $B_{l,k}$ or exponential, i.e. reusable, atomic formulae $B^{\sharp}_{m,k}$. If the ? modality is present, the multiset of conclusions is required to be reusable as a whole: not only these conclusions cannot be used with a linear clause but such a clause cannot use linear hypotheses. This is in fact the meaning of the promotion rule: the context must be exponential. For example, consider the set of clauses $\{1 \multimap A\otimes B, B \multimap ?C, !(A \otimes C) \multimap \bot\}$. The corresponding module we get is drawn in Fig.4 on the left hand side. The figure on the right hand side is the corresponding proof-structure (see [9, 12] for definitions of proof structures with boxes). The traversal of the box without the use of a ♭-node shows that the sequent is not provable (a dereliction should have been applied), i.e. the ?-EBM on the left is not correct.

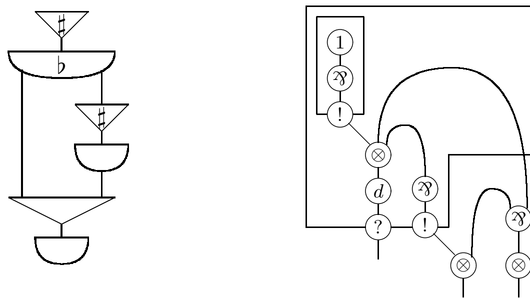


**Fig. 4.** ?-EBM and proof nets

**From ?-EBMs to Modules.** Definitions given in section 2.1 and correctness of modules, cannot be straightfully extended to the exponential case, i.e. composition of ?-EBMs (still called modules). Obviously, composition should satisfy identification of variables occurring on links, noticing that $*$-edges can only be linked to $*$-edges. However, though dereliction and weakening properties of exponentials have no consequence on composition, contraction requires a special attention. With *implicit* contraction, ?-EBMs are not concretely duplicated but one may consider multiple incident edges (see Fig. 5). In fact, as a ?-EBM may be as a whole duplicated with respect to contraction, the degree of such edges should be the same for each link place. However, it is obvious that non-determinism occurs in such a case: nothing in composition of ?-EBMs allows to know if a link refers to one or the other of the "clones". As a non-trivial example, let us consider program,goal and facts given Fig. 5. Note that $\sharp$ as well as exponential atomic formulae are not used in this example. It is easy to prove that the system is correct and uses two contraction rules. Let us follow a trip beginning from the facts. The pairs $a_1, a_2$ and $b_1, b_2$ fire respectively the pairs $u_1, v_1$ and $u_2, v_2$. However, it is not straightforward to know how to continue the trips. In fact, correctness requires to fire the two top EBMs on left and right (since they are linear, they must be used once before finishing the trip), going down firing again the two ?-EBMs and finally firing the top central (linear) EBM.

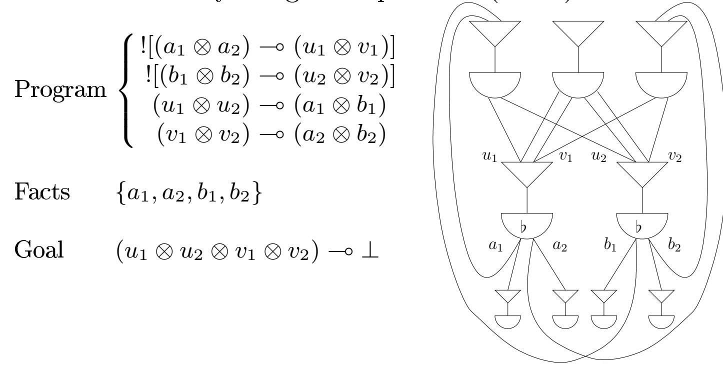
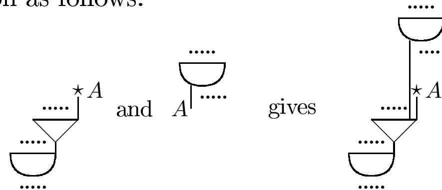



**Fig. 5.** Example of an ambiguous module

In the following, we consider *explicit* contraction: ?-EBMs with positive nodes labelled $\flat$, and $*$-edges are duplicated if necessary, mimicking the property $!A \multimap !A \otimes A$, hence the degree of edges is always 1, keeping the information the duplication carried out. The definition of composition given in section 2.1 is then adapted for ?-EBMs labelled $\flat$ and $*$-edges. For example, $*$-edges are duplicated during a composition as follows:



### 3.2 ?-EBMs and corresponding Correctness Criteria

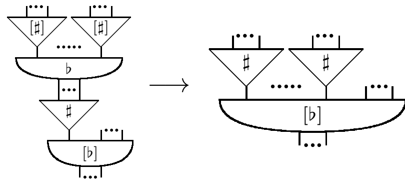
We consider the introduction of exponentials with respect to its consequences regarding the correctness of modules and to its interest for programming. Extending the language with exponentials yields a major difficulty due to the promotion rule, i.e. when ?-EBMs have $\sharp$ labels, as it is inherently contextual. Two approaches may be considered: exclude $\flat$ in the scope of other $\flat$, a kind of digging, or authorize the full language. We first consider another system where, drastically, the promotion rule is not allowed: $\sharp$ is excluded from the language.

**Programming with Linear and Reusable Resources: Exclude $\sharp$.** Allowing $\flat$ in the language is sufficient to embed the framework of the previous sections in a programming language: one can consider a program as a set of (exponential, reusable) ?-EBMs together with a multiset of (linear, usable once) ?-EBMs. Clauses are of the following form:

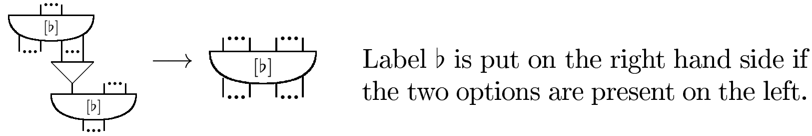
$$\begin{cases} \quad !(1 \multimap (\bigotimes_{l \in L} A_l \bigotimes_{m \in M} A^{\sharp}_{m})) \\ !(\bigotimes_{i \in I} A_i \multimap \bindnasrepma_{k \in K} (\bigotimes_{l \in L} B_{l,k} \bigotimes_{m \in M} B^{\sharp}_{m,k})) \\ !(\bigotimes_{i \in I} B_i \multimap \bot) \end{cases} = \mathbb{C}$$

Note that variables named $B^{\sharp}_{m,k}$ correspond to reusable resources, and that this system already extends classical logic programming in a straightforward way. Correctness of modules is tested with the same reduction relation $\twoheadrightarrow$ given in section 2.4, after deleting $*$-edges (i.e. application of the weakening rule) and by considering that normal forms may contain ?-EBMs.

**Programming with Constraints on Reusability of Resources.** The $\sharp$ modality expresses a constraint on a given resource whereas $\flat$ expresses the fact that some resource is reusable. Let us detail this point. The clause $X \multimap !Y$, i.e. a $*$-edge for $Y$, means that the resource $Y$ is a classical fired resource as long as resource $X$ is given: $Y$ may be used zero, one or more times in the sequel of the program execution. This is the standard way logic programming is achieved. On the contrary, $X \multimap ?Y$, i.e. the negative pole labelled by $\sharp$, means that, $X$ being given, resource $Y$ may be used only as a hypothesis in a ?-EBM (such as $!(Y \multimap Z)$). Furthermore, other hypotheses used by such a ?-EBM must be exponential. Hence excluding $\flat$ in the scope of other $\flat$ in the construction of formulae amounts to simplifying the correctness criterion as the reduction rule does not impose constraints on other hypotheses of the ?-EBM to be reduced:

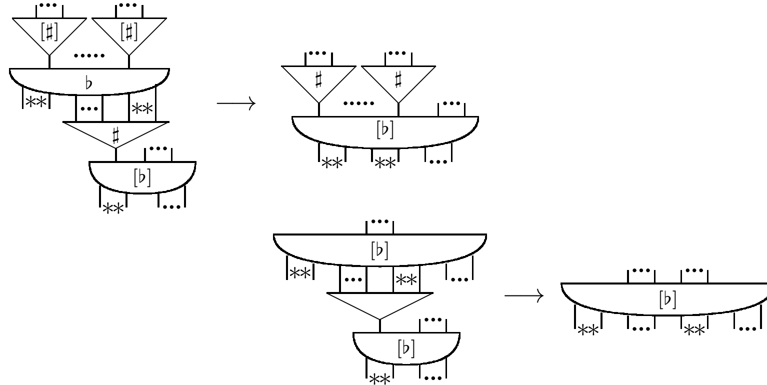



Label $\flat$ is put on the right hand side if an option is present on the left.

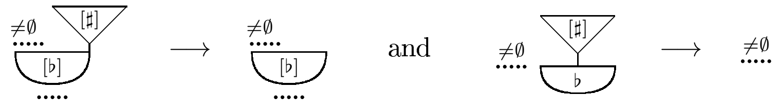



Label ♭ is put on the right hand side if the two options are present on the left.

If the full language is allowed, exponential variables may be used as hypotheses and the reduction system is given by the following two rules, with the same conditions on label ♭ on the right part as before:




Propositions equivalent to the ones given for the multiplicative case may be proved for each reduction system (and preserved by forgetting exponentials). We must also characterize normal forms. To get a correctness theorem, we add to the reduction system two rules corresponding to neutrality of **1** and weakening of ♭:




**Theorem 2 (?-EBM correctness [8]).** *A closed module $M$ is correct iff $M \longrightarrow^*$ or $M \longrightarrow^*$.*

Note that the correctness criterion is still checked in linear time with respect to the number of the nodes of the BM. However checking correctness of a computation on-the-fly remains a difficulty as reduction with the rewriting system does not commute with composition of BMs as we noticed in [8].

## 4 Towards Distributed Computation

What does distributed logic programming require? What could be the benefits? Let us consider an agent-based architecture. Each agent has at disposal a set of *modules*, that may be called services or functionalities in other programming paradigms. These modules may be used for achieving some task, either searching how to do it, or deducing consequences of some situation. With that

in mind, the same architecture may be used in different contexts: planning, test, execution, assembling. Current distributed and/or collaborative programming languages have these objectives, however they are unable to take into account non-intuitionnism even if this is clearly required if one takes seriously into account open and dynamic situations: there may be several ways to tackle a problem and these possibilities may not be always available. Furthermore, current programming languages either are based on Prolog or use classical logic as a type system. In the two cases, usability of resources is badly managed. E.g. neither closed-world assumption nor first-order usage are adequate solutions as the integration to the logical structure is unclean. The shift to Linear Logic prevents these drawbacks by providing a complete non-intuitionnistic as well as intuitionnistics logical framework. Moreover, the graph formalism we presented in previous sections satisfy three fundamental properties: reversibility, disconnection between application of clauses and correctness checking, partial application of clauses. These three key concepts help to enlarge logic programming towards a distributed backward and forward computation paradigm.

Computation is reversible as far as contraposition may be safely used in designing logic programs. It follows from the fact that full Linear Logic may be viewed as an Abstract Logic Programming Language [2]: conjunction as well as disjunction are included in the logic structure of clauses. It is also obviously related to duality of Linear Logic. This property allows us to define contraposed versions for the intuitionnistic as well as the non-intuitionnistic cases (section 2). Note that the structure of modules gives for free this result as examplified in the generalization of the multiplicative case where resolution and progression are available for the same class of modules. Backward and forward computations are not only admissible complete strategies but they can also be mixed as needed. This is particularly important in an agent-based architecture where modules are the knowledge of agents that try to combine them to reach some predefined goal.

In standard logic programming, resolution and progression (i.e. application of clauses) are defined as a search for *uniform* proofs: each step should give rise to the full decomposition of some clause with respect to a given goal. When considering Horn programs, the fact that the propositional variable that is the head of a clause unifies with one literal of the goal is sufficient for a resolution step to be valid. Parallel resolutions may be done when applied to distinct literals. This is no longer the case when dealing with full linear logic as the structure of the head of a clause is far more complex. For example, the head of a clause in the exponential case has the following form: $⅋_{k\in K}[?]\ (\bigotimes_{l\in L}\ B_{l,k} \bigotimes_{m\in M}\ B^{\sharp}_{m,k})$. Checking the validity of a resolution step requires to capture from the goal a structure containing literals $B_{l,k}$ and $B^{\sharp}_{m,k}$, hence parallelization cannot be done without managing a synchronization mechanism with respect to a complex structure of literals. We showed in this paper that validity of a resolution or progression step may be done in two parts: first checking that literals appearing in the head are available in the goal (as in the standard case), second check of some correctness criterion that may be delayed until the end of the computation.

Remember also that these correctness criteria are given as rewriting systems with a complexity linear in terms of resolution steps.

Finally, the graph viewpoint seems to be the more adequate data model. This was first pointed out by Andreoli in [3] in a very specific case (a fragment of the multiplicative linear logic). We recall in this paper that result may be extended to full exponential multiplicative clauses. A sequent viewpoint enforces the reduction of clauses (i.e. of the head) to occur when applying a clause. A contrario, the composition between two graphs may concern only part of their respective bottom and top structures. This allows for considering distributed programs among a set of hosts. Partial reductions are undertaken between pairs of hosts, the correctness of the whole process being checked only at the end of the computation, thanks to the previous remark.

## References


1. S. Abramsky. Computational interpretations of linear logic. *Theor. Comput. Sci.*, 111(1&2):3–57, 1993.
2. J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. Logic and Computation*, 2(3):297–347, 1992.
3. J.-M. Andreoli and L. Mazaré. Concurrent construction of proof-nets. In *CSL*, volume 2803 of *LNCS*, pages 29–42. Springer, 2003.
4. P. Boudes. Projecting games on hypercoherences. In J. Díaz, J. Karhumäki, A. Lepistö, and D. Sannella, editors, *ICALP*, volume 3142 of *LNCS*, pages 257–268. Springer, 2004.
5. P. Bruscoli and A. Guglielmi. A tutorial on proof theoretic foundations of logic programming. In *ICLP*, volume 2916 of *LNCS*, pages 109–127. Springer, 2003.
6. V. Danos. *Une application de la logique linéaire à l'étude des processus de normalisation (principalement de $\lambda$-calcul)*. PhD thesis, Université Denis Diderot, Paris 7, 1990.
7. C. Fouqueré and V. Mogbil. Rewritings in polarized (partial) proof structures. Technical report, Technische Universität Dresden, July 2005. ISSN 1430-211X.
8. C. Fouqueré and V. Mogbil. Rewritings for polarized multiplicative and exponential proof structures. *Electr. Notes Theor. Comput. Sci.*, 203(1):109–121, 2008.
9. J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.
10. S. Guerrini. Correctness of multiplicative proof nets is linear. In *Logic in Computer Science*, pages 454–463, 1999.
11. M. I. Kanovich. Linear logic as a logic of computations. *Ann. Pure Appl. Logic*, 67(1-3):183–212, 1994.
12. Y. Lafont. From proof-nets to interaction nets. In J.-Y. Girard, Y. Lafont, and L. Regnier, editors, *Advances in Linear Logic*, volume 222 of *London Math. Society Lecture Note*, pages 225–247. Cambridge Univ. Press, 1995.
13. O. Laurent. Syntax vs. semantics: a polarized approach. *Theoretical Computer Science*, 343(1–2):177–206, Oct. 2005.
14. D. Miller. Overview of linear logic programming. In T. Ehrhard and al., editors, *Linear Logic in Computer Science*, volume 316 of *London Math. Society Lecture Note*. Cambridge University Press, 2004.
15. D. Miller and al. Uniform proofs as a foundation for logic programming. *Ann. Pure Appl. Logic*, 51(1-2):125–157, 1991.