Abstract

The cut elimination/ normalization in a logical calculus may simulate computations. This amounts to designing a programming language on a logical calculus. Typed λ–calculus is a familiar example. Linear logic has been attempted used for simulating both a "resource sensitive" computation, and parallel computations. We investigate how to let linear logic simulate parallel computations. The focus is on the multiplicative fragment in the proof net formalization of linear logic. A translation from such proofs into a corresponding CSP process is offered. It is shown that the translation yields a step–by–step correspondence between the cut elimination process and the CSP execution. Generalizations and related work are discussed. We are concerned with the handling of the failure of the Church–Rosser property. It fails for proof nets with "boxes". We point out the two possible strategies to adopt : either restrict attention the cases where the property does hold, or try to give an interpretation were the ordering matters.

Keywords : Simulating computations in logic; Linear logic; Cut elimination in proof nets; Parallel processing; Church–Rosser property;

1 Introduction

Linear logic (LL) was introduced by Girard in [5]. Two different formalizations of LL were offered : one as a sequent calculus, and one as proof nets. In the sequent calculus the prominent feature of LL is made clear : the restrictions on the structural rules weakening (W) and contraction (C) which usually read :

$$W\frac{\Gamma}{\Gamma, A} \quad \frac{\Gamma, A, A}{\Gamma, A}C$$

They no longer apply to an arbitrary formula, A. Only modalized formulas, $?A$, may be weakened and contracted and — this is the point — there are specific rules for introducing the modalities, thus making it explicit. In [10] LL is used to simulate the "resource sensitive" Petri nets.

The proof net formalization of LL is an attempt to obtain two advantages compared to the sequent calculus for LL : (i) identification of all the sequent proof which are only permutations of one another, (ii) allowing multiple conclusions. The first is what is usually accomplished with natural deduction, the second is the essential feature here : it allows a parallelization of the deduction and the cut elimination. The ambition expressed in (i) and (ii) above works only in a fragment of LL, first order multiplicative and additive LL without constants [6, 7, 3].

From a mathematical/ logical point of view, LL enjoys a number of appealing properties. The most important being : cut elimination, strong normalization and an embedding of both classical and intuitionistic logic.

Our concern is to investigate the possibilities of interpreting the process of cut elimination in LL computationally. More specifically, we attempt to express the parallel process of cut elimination in the proof net formalization of MLL^- in a more familiar formalism for concurrency, viz. Communicating Sequential Processes (CSP) [9]. The hope that normalization in proof nets should have such a connection with parallel computing is expressed in e.g. [4, 5].

Our approach is to :

1. give a translation $t(P)$ taking proof nets into a corresponding CSP process;
2. give a map $e(\cdot)$ which executes the process $t(P)$ one step according to the laws of CSP;
3. define an inverse map $i(\cdot)$ which takes as argument the result after executing the process $t(P)$ one step, and returns the proof net P' such that :

$$(*) \quad i(e(t(P))) = P', \quad r(P) = P'$$

$r(\cdot)$ is the reduction given by the cut elimination process in the proof nets of MLL^-. In other words, the following diagram commutes :

$$\begin{array}{ccc} P & \stackrel{r}{\longrightarrow} & P' \\ \downarrow t & & \uparrow i \\ t(P) & \stackrel{e}{\longrightarrow} & e(t(P)) \end{array}$$

In section 2 a short presentation of the relevant parts of LL and CSP is given. The translation $t(\cdot)$ and map $e(\cdot)$ are given in section 3, and section 4 contains the map $i(\cdot)$ together with $(*)$.

Section 5 is a discussion of the results and the possibilities of generalizations. We comment on related work, especially [1, 2]. [2] gives a translation which properly contains ours. We make three points : it does make use of a process algebra strictly stronger than CSP (the π–calculus [11]), it does not aim at giving a step–by–step correspondence, and it focus on the cases where the Church–Rosser property holds. We discuss two different strategies handling the failure of the Church–Rosser property : either avoid it or try to incorporate it.

2 LL and CSP preliminaries

We only give a brief presentation of the relevant parts of LL and CSP and refer to [5], [9], for a proper exposition. Only the proof net formalization for $MLL^{-}$and a subset of CSP are covered.

2.1 LL

In LL negation $(\cdot^{\perp})$ is a defined operator. It is given by the fact that the de Morgan laws hold for each of the multiplicative, additive, exponential and quantifier fragments of LL separately. In particular we have for MLL :

$$A = A^{\perp\perp}, (A \otimes B)^{\perp} = A^{\perp} + B^{\perp} (A+B)^{\perp} = A^{\perp} \otimes B^{\perp}$$

Proof nets are graphs built from axiom links (A atomic)

$$\overset{\sqcap}{A\ A^{\perp}}$$

and continued with one of the three nodes :

$$\otimes\frac{A \quad B}{A \otimes B} \qquad +\frac{A \quad B}{A+B} \qquad \frac{A \quad A^{\perp}}{\mathrm{CUT}}$$

The proof nets are inductively defined by:

- An axiom link is a proof net with the two conclusion A and $A^{\perp}$.

- $\dfrac{\begin{matrix} P & Q \\ \vdots & \vdots \\ A & B \end{matrix}}{A \otimes B}$ is a proof net provided P is a proof net with conclusion A, and Q is another proof net with conclusion B. The resulting proof net has as conclusions those of P and Q, but with A and B removed and with $A \otimes B$ as a new conclusion.

- $\dfrac{\begin{matrix} P & P \\ \vdots & \vdots \\ A & B \end{matrix}}{A+B}$ is a proof net if A and B are two conclusions of the same proof net P. The resulting proof net removed the A and B as conclusions, but has added $A+B$.

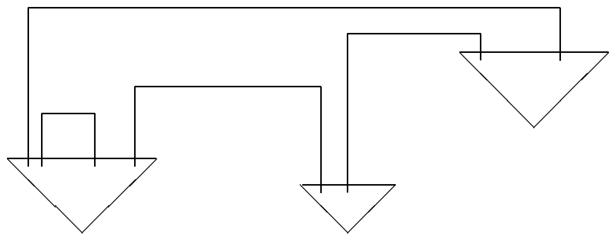


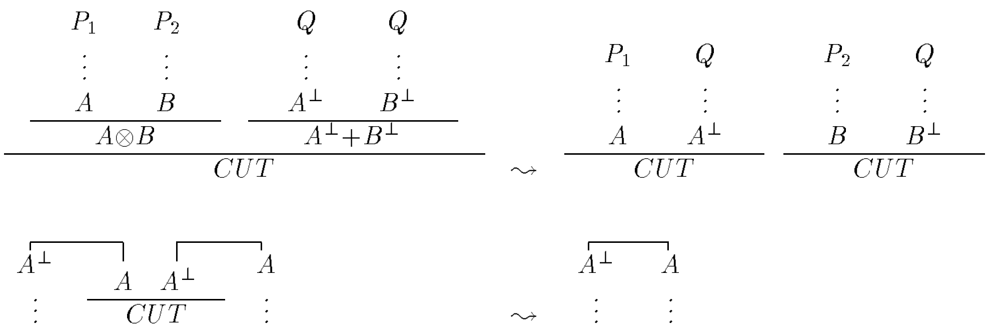
Figure 1: Proof net trees

- $$\frac{\begin{array}{cc} P & Q \\ \vdots & \vdots \\ A & A^{\perp} \end{array}}{CUT}$$ is a proof net if P is a proof net with conclusion A and Q is another proof net with conclusion $A^{\perp}$. The resulting proof net has the same conclusions as two original ones taken together, only with the A and $A^{\perp}$ removed. CUT is not a formula (and accordingly not a conclusion).

Remark 1 *Proof nets are seen to consist of a number of trees — we call them* **proof net trees** *— which have as a bottom node either (i) a conclusion formula, or (ii) a CUT. They are linked together at the top, possibly across two trees. The proof net trees are the* cut free *construction of (i) the conclusion formula, or (ii) the two cut formulas above CUT, respectively. This is illustrated in figure 1.*

In the proof net formalization of MLL^{-} there are only symmetric cases in the cut–elimination (when axioms are taken as atomic formulas only). The reduction steps are of two kinds : either a compound formula is split and the cut propagated, or the process terminates as the formula is atomic. These cases define the reduction function $r(\cdot)$ on proof nets in MLL^{-} :

Definition 2 (Reduction of proof nets) *The reduction $r(\cdot)$ on proof nets of MLL^{-} is given by :*



The reductions are local and may thus be performed in parallel. This is the behaviour we want to capture using a formalism for expressing concurrency, CSP.

2.2 CSP

CSP come equipped with three things : (i) a programming language, (ii) specification of behaviour (expressed algebraically) for every language construct from (i), and (iii) a formal semantics. We leave the latter out.

The basic entities for explaining (i) are processes (that which may execute by engaging in events and are built from the basic language constructs), events (which are what processes engage in), traces (which are a history of the events a process already *have* engaged in), and alphabets (which are the set of events a process *may* engage in).

There is a wide range of CSP programming language constructs. We focus on the ones relevant for our purposes. They do, however, fall in two categories :

- I These are the ones absolutely necessary for defining the translation from the proof nets. It is a "simplified" version of CSP with only two constructs $\rightarrow, ||$ (explained further below) and only a subset of the laws governing the behaviour of $||$.
- II These include two more language constructs ($|, STOP$) and a more elaborate definition of the alphabets, $\alpha(P)$, of the process P. This is needed because all CSP laws related to the subset of constructs used are in effect.

In short, the basic ideas for expressing the cut elimination process of proof nets underlying the translation are found already in I. But to avoid technical CSP problems, II is needed. We will use version II.

We present CSP version I first. The language constructs are : (i) given the process P, the process $a \rightarrow P$ (pronounced "prefixing") is to first engage in event a and then proceed as process P, and (ii) given two processes P and Q, form $P||Q$ (pronounced "interaction" or "parallel composition") which is the parallel, synchronized execution of P and Q.

Notation 3 *We take $\rightarrow$ to bind tighter than $||$.*

For a trace s, P/s denotes the resulting process after P has engaged in the events of s. In the case of a singleton trace $\langle a \rangle$ we have :

$$(a \rightarrow P)/\langle a \rangle \quad = \quad P \tag{1}$$

Interaction $||$ is the essential CSP construct — it is the only way to make (sub)-processes execute. $||$ is commutative and associative. The following law expresses the basic property that the two processes engage in (i.e. synchronize) their common event c, and then continue to interact :

$$c \rightarrow P || c \rightarrow Q \quad = \quad c \rightarrow (P || Q) \tag{2}$$

This is the basic law which we will make use of in the translation when describing the cut elimination process.

With the more elaborate version II of CSP, we look more carefully at the alphabets. For CSP processes to be well–defined, they have to obey certain requirements on their alphabets. The requirements are given relative to each of the language constructs. The alphabet requirement on $\rightarrow$ is that $\alpha(a \rightarrow P) = \alpha(P)$ and on $||$ that $\alpha(P||Q) = \alpha(P) \cup \alpha(Q)$. $||$ thus allows the composition of processes with different alphabets, a fact to be exploited later.

A new construct, $|$ (pronounced "choice"), is introduced. $P|Q$ is either P or Q depending on the composition with $||$. Choice is idempotent, symmetrical and distributes through $\rightarrow$ (when the events are equal). The requirement on the alphabet is that $\alpha(P|Q) = \alpha(P) = \alpha(Q)$. The actual choice between P, Q is made when $P|Q$ interacts through $||$ with another process. A process $STOP_\alpha$ is defined. It does not engage in any of the events in its alphabet α.

The additional information about the alphabets of the processes play a role when they start to execute (i.e. interact). Let $a \in \alpha(P) - \alpha(Q), b \in \alpha(Q) - \alpha(P), c, d \in \alpha(P) \cap \alpha(Q), c \neq d$. The operational behaviour about $||$ is then given by (2) in addition to :

$$\begin{aligned}
a \rightarrow P || c \rightarrow Q \quad &= \quad a \rightarrow (P || c \rightarrow Q) && (3)\\
a \rightarrow P || b \rightarrow Q \quad &= \quad (a \rightarrow (P || b \rightarrow Q)) | (b \rightarrow (a \rightarrow P || Q)) && (4)\\
(c \rightarrow P | d \rightarrow Q) || (c \rightarrow R) \quad &= \quad c \rightarrow (P || R) && (5)\\
(c \rightarrow P || d \rightarrow Q) \quad &= \quad STOP_{\alpha(P) \cup \alpha(Q)} && (6)
\end{aligned}$$

Remark 4 *The point we will use is the difference between* (4) *and* (6) *: if the two events are unequal but elements of* different *alphabets, as a, b in* (4)*, then the composition behaves like an alternation — either engage in a or in b, then proceed. If the two events are unequal and from the* same *alphabets, as for c, d in* (6)*, then the process halts.*

As we will see later, a proof net is to be translated into a process of the form: $e \rightarrow \ldots || f \rightarrow \ldots$ where e and f are (the translation of) the conclusions of two proof net trees of the proof net. If now e and f belong to the same alphabet, (6) *applies and the process stops immediately. We avoid this by making sure that e and f belong to disjoint alphabets. The mechanism for ensuring this is given in section 3.2.*

3 The translation

3.1 The principles

We want to encode proof nets as CSP processes, i.e. in terms of processes and events. The components of a proof net which must be so interpreted are thus :

(i) the formulas,

(ii) rules $(\otimes, +)$,

(iii) cuts

(iv) axiom links

(v) the proof net trees of the proof net.

We translate proof nets bottom up by the function $t(\cdot)$.

There will be a need to reference the literals of formulas in an unambiguously way to distinguish between different instances of the same formula. To this end, an indexing (see definition 6 below) operation which gives every literal in the formulas of the conclusions a unique name is defined. These indices are subsequently passed on upwards. Axiom links also receive unique indices.

Notation 5 *We use i, j, k, l for the unique indices.*

(i) Formulas are interpreted as events. So $A_i, A_i \otimes B_j, (A_i \otimes B_j) + C_k$ are three different events.

(ii) The proof net trees are sequences of nodes which (read from the bottom) : may start with one CUT node, proceeds by an arbitrary number of $\otimes, +$ nodes and end in axiom links. If the possible, bottom CUT node is left out, there are only sequences of rules ending in axiom links. Such a sequence of nodes is captured in the translated process by the prefixing, $\rightarrow$, of events :

$$\frac{\ldots}{A_i}$$

is translated into $A_i \rightarrow t(\ldots)$. Furthermore, the branching in connection with the $\otimes, +$ nodes is translated by composing the two branches. Taken together we obtain for a configuration :

$$\begin{array}{cc} P & Q \\ \vdots & \vdots \end{array}$$
$$\otimes\,\frac{A_i \qquad B_j}{A_i \otimes B_j}$$

the process $A_i \otimes B_j \rightarrow (A_i \rightarrow t(P) \| B_j \rightarrow t(Q))$. This process first engages in the event $A_i \otimes B_j$, then it proceeds. See comments in the next paragraph concerning the alphabet.

(iii) Similarly, a cut node is translated into the composition of the two branches. We introduce an equality, $\approx$, on formulas/ events which identifies *dual* formulas (definition 7 below). There will, however, be a notable difference concerning the alphabets.

LL	CSP
proof net	process
rule	prefixing
formula	event
cut–elimination	interaction

Figure 2: The principles of the translation

In the case with branching in connection with the $\otimes, +$ rules, the two branches will have *disjoint* alphabets whereas the two branches of a CUT will have the *same* alphabet. As was pointed out in the last remark in section 2.2, the behaviour will accordingly be very different despite the syntactic similarity that $||$ is used in both cases. The interaction in connection with the cuts will turn out to be the ones where (2) is applicable, i.e. to successful interaction. This fits well with the fact that the only dynamic feature of the proof nets is the cut elimination, and the only truly operational behaviour in CSP is successful interaction.

(iv) The axiom links are coded by a pair of special processes $S_k, S_k^\perp$ for every axiom link with index k. These processes are left unspecified as we will never be interested in their behaviour (see section 4).

(v) The translation of a proof net is by translating the proof net trees in turn and then forming the parallel composition of all. The alphabets of any the translation of any pair of proof net trees are disjoint (see observation ($X1$) below). Disjoint alphabets signify unrelated alternation as we have seen. This expresses the fact that one may operate on *any* of the conclusions of a proof net without disturbing the rest.

The basic idea of the translation, $t(\cdot)$, is summed up in figure 2.

3.2 Indexing, translation and execution

Definition 6 (Indexing) *The indexing ($\cdot^\bullet$)is given by*

All bottom formulas are indexed first by indexing all literals in turn :

$$(B\otimes C)^\bullet = B^\bullet \otimes C^\bullet,\ (B+C)^\bullet = B^\bullet + C^\bullet,\ A^\bullet = A_i \tag{7}$$

where A atomic, i a new index. Then the following rules are applied :

$$\left(\begin{array}{cc} P & Q \\ \vdots & \vdots \\ A & A^\perp \\ \hline \multicolumn{2}{c}{CUT} \end{array}\right)^\bullet = \begin{array}{cc} (P)^\bullet & (Q)^\bullet \\ \vdots & \vdots \\ A_i & A_i^\perp \\ \hline \multicolumn{2}{c}{CUT} \end{array},\ i \textit{ a new index} \tag{8}$$

$$\left(\begin{array}{cc} P & Q \\ \vdots & \vdots \\ A & B \\ \hline \multicolumn{2}{c}{A_i \otimes B_j} \end{array} \otimes \right)^\bullet = \begin{array}{cc} (P)^\bullet & (Q)^\bullet \\ \vdots & \vdots \\ A_i & B_j \\ \hline \multicolumn{2}{c}{A_i \otimes B_j} \end{array} \otimes \tag{9}$$

$$\left(\begin{array}{cc} P & P \\ \vdots & \vdots \\ A & B \\ \hline \multicolumn{2}{c}{A_i + B_j} \end{array} + \right)^\bullet = \begin{array}{cc} (P)^\bullet & (P)^\bullet \\ \vdots & \vdots \\ A_i & B_j \\ \hline \multicolumn{2}{c}{A_i + B_j} \end{array} + \tag{10}$$

$$(\overline{A_i \; A_j^\perp})^\bullet = \overset{l}{\overline{A_i \; A_j^\perp}} \quad , \; l \textit{ a new index} \tag{11}$$

From now on we take all proof nets to be indexed.

In section 3.1 we explained that successful interaction was to be modelled by cut elimination. But in CSP interaction is between two (syntactically) equal events whereas two cut formulas are dual. To cope with this, we make the following modification of CSP : equality of events are defined by $\approx$.

Definition 7 (Equality) *Equality ($\approx$) of events is given by :*

- *If A is atomic then $A_i \approx A_i^\perp$*
- *If $A_i \approx A_i^\perp, B_j \approx B_j^\perp$ then $A_i \otimes B_j \approx A_i^\perp + B_j^\perp$, $A_i + B_j \approx A_i^\perp \otimes B_j^\perp$*
- *If $i \neq j$ then $A_i \not\approx B_j$ for any formulas A, B*

If $F_i \approx F_i^\perp$, we write $\langle F_i, F_i^\perp \rangle$ for the singleton trace consisting of this event.

Remark 8 *(i) The modification of CSP induced by the introduction of an explicit equality, $\approx$, is minor. In CCS [12] the interaction is between dual events.*

(ii) We could have kept closer to CSP and avoided $\approx$ by modifying the translation $t(\cdot)$ in equation (13) (see below) to yield A_i in both branches. The only problem with this is that it makes an inverse map, $i(\cdot)$, (see section 4) more difficult as there will be no immediate way to decide whether $i(A_i \to P \| A_i \to Q)$ is to be given the value :

$$\begin{array}{cc} i(P) & i(Q) \\ \vdots & \vdots \\ A & A^\perp \\ \hline \multicolumn{2}{c}{CUT} \end{array} \quad or \quad \begin{array}{cc} i(P) & i(Q) \\ \vdots & \vdots \\ A^\perp & A \\ \hline \multicolumn{2}{c}{CUT} \end{array}$$

Observations :

($X1$) As the conclusions of the proof net are indexed in turn, the indices of any two proof net trees are different.

($X2$) (i) The set of indexed formulas in any two proof net trees are disjoint, (ii) except for the two branches of a cut, which have exactly the same set of formulas.

(i) follows from ($X1$), (ii) from the definition of $\approx$ together with the fact that the two branches are the cut free construction of the two, dual cut formulas.

The translation of the proof net to a CSP process will take the form : $\ldots||\ldots||\ldots$ where the $\ldots$'s represent the translation of the proof net trees. We translate these proof net trees separately. Then we define the alphabets of the translated proof net trees.

Let $t(P_1), \ldots, t(P_n)$ be the translation of the proof net trees $P_1, \ldots P_n$ given by $(13) - (16)$ below, then :

$$t(P) \quad = \quad t(P_1)||\ldots||t(P_n) \tag{12}$$

is the translation of the complete proof net P consisting of $P_1, \ldots, P_n$.

Definition 9 (Translation of proof net trees) *The translation of a proof net tree is given by :*

$$t\left(\begin{array}{c} \begin{array}{cc} P & Q \\ \vdots & \vdots \\ A_i & A_i^{\perp} \end{array} \\ \hline CUT \end{array}\right) \quad = \quad A_i \rightarrow t(P)||A_i^{\perp} \rightarrow t(Q) \tag{13}$$

$$t\left(\begin{array}{c} \begin{array}{cc} P & Q \\ \vdots & \vdots \\ A_i & B_j \end{array} \\ \hline A_i \otimes B_j \end{array}\otimes\right) \quad = \quad ((A_i \otimes B_j) \rightarrow t(A_i \rightarrow P||B_j \rightarrow Q)) \tag{14}$$

$$t\left(\begin{array}{c} \begin{array}{cc} P & P \\ \vdots & \vdots \\ A_i & B_j \end{array} \\ \hline A_i + B_j \end{array}+\right) \quad = \quad ((A_i + B_j) \rightarrow t(A_i \rightarrow P||B_j \rightarrow Q)) \tag{15}$$

$$t(\overbrace{A_i \;\; A_j^{\perp}}^{l}) = A_i \rightarrow S_l||A_j^{\perp} \rightarrow S_l^{\perp} \tag{16}$$

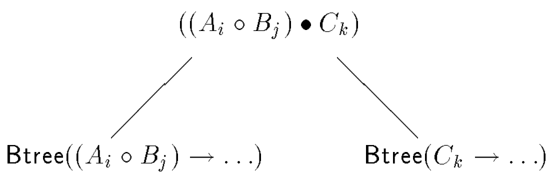


Figure 3: Representation of a translated process as a binary tree, $\mathsf{Btree}(\cdot)$.

In section 5 we comment briefly on the related translation in [2].

It remains to define the alphabets of the processes constructed by the applications of (13) – (16)[1]. Let $\circ, \bullet$ denote $\otimes$ or $+$. Represent the translation of a proof net tree, say :

$$P = (((A_i \circ B_j) \bullet C_k) \rightarrow ((A_i \circ B_j) \rightarrow \ldots || C_k \rightarrow \ldots))$$

as a binary tree, $\mathsf{Btree}(P)$, where the prefixed events/ formulas of P appear as interior nodes as in figure 3.

$\mathsf{Btree}(S_k) = S_k, \mathsf{Btree}(S_k^\perp) = S_k^\perp$, for any k.

Any subprocess of P may now be referenced uniquely by a formula node in $\mathsf{Btree}(P)$, and we do not make a distinction in what follows. For a node $G_m \in \mathsf{Btree}(P)$, let $\mathsf{left}(G_m)$, $\mathsf{right}(G_m)$, $\mathsf{mother}(G_m)$ denote the left subtree, right subtree, predecessor of G_m, respectively, and $\mathsf{interior}(Q)$ the set of non–leaf nodes in the subtree with root G_m. Define $\mathsf{extra}(G_m)$ for any $G_m \in \mathsf{interior}(P)$ to be the set of formulas along the longest path going up to the right from G_m, i.e. it is the smallest set satisfying :

$$(i)\ G_m \in \mathsf{extra}(G_m), \quad (ii)\ \{K_p |\ \exists H_n \in \mathsf{extra}(G_m).\mathsf{left}(K_p) = H_n\}$$

Definition 10 (Alphabets) *The alphabet $\alpha(G_m)$ of any non–S_k G_m of P is defined by :*

$$\alpha(G_m) \quad = \quad \mathsf{interior}(G_m) \cup \mathsf{extra}(G_m) \tag{17}$$

and for the S_k's :

$$\alpha(S_k) \quad = \quad \alpha(\mathsf{mother}(S_k)) \tag{18}$$

Observation :

$(T1)$ For any $G_m \in \mathsf{Btree}(P)$ we have that $\alpha(\mathsf{left}(G_m)) \cap \alpha(\mathsf{right}(G_m)) = \emptyset$.

In the example given in section A the alphabets of the subprocesses have been given according to the above definition.

The execution step, $e(\cdot)$, has two components : (i) a successful, parallel engagement in an event c (equation (2) above), and (ii) forgetting the common event c and only focusing on the continued behaviour. We could have defined the component (ii) by a function $\cdot^\circ : (\langle A_i, A_i^\perp \rangle \rightarrow P)^\circ = P$, and the identity otherwise.

[1] The definition could have been simplified if we kept only to version I of CSP (see section 2.2)

But this would not bring out the fact that $\cdot^\circ$ is definable in CSP through (1)[2]. We instead combine the laws (2) and (1) into :

Definition 11 (Single execution step) *The single execution step $e(\cdot)$ is defined by :*

$$e(A_i \to P \| A_i^\perp \to Q) = P \| Q \tag{19}$$

and is otherwise the identity function.

Observations :

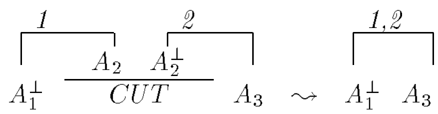
$(E1)$ Execute, $e(\cdot)$, on translated, cut-free proofs is the identity function.

$(E2)$ We execute *one* step. After the completion of this step, e is the identity :

$$e(A_i \to P \| A_i^\perp \to Q) = P \| Q = e(P \| Q)$$

We give a simple example. A more elaborate one is given in section A.

Example 1 *Consider the single reduction step :*



Let P denote the proof net to the left of $\rightsquigarrow$. Then $r(P)$ is the one to the right. We have that :

$$\begin{array}{rcl} t(P) & \overset{(16,13)}{=} & A_1^\perp \to S_1^\perp \| A_2 \to S_1 \| A_2^\perp \to S_2^\perp \| A_3 \to S_2 \\ e(t(P)) & \overset{(19)}{=} & A_1^\perp \to S_1^\perp \| (S_1 \| S_2^\perp) \| A_3 \to S_2 \end{array}$$

Note the similarities with the resulting process above and a direct translation of $r(P)$ using (16). In the next section we shall see how to interpret $(S_1 \| S_2^\perp)$ as the collapsing of the axiom links 1 and 2. This enables an inverse map from the resulting processes to proof nets to be defined.

4 Correspondence

Our aim in this section is to establish :

$$i(e(t(P))) = r(P)$$

We make use of a function $f(\cdot)$ for recording the initial axiom links in the proof net, i.e. before the application of r. The link :

[2] This is a point taken from [2] where the hiding operator is used. Hiding conceals all communication of an event, whereas (1), only conceals one event. But as no events will show up twice as they come from the cut–free construction of a conclusion and obey the subformula property, this amounts to the same.

$$\overset{l}{\sqcap}\quad A_i \; A_j^\perp$$

is recorded by f by defining :

$$f(l) = A_i, \; f(l^\perp) = A_j^\perp$$

As the translation $t(\cdot)$ of a proof net, the function $i(\cdot)$ is defined by induction on each proof net tree separately. If the subprocesses $P_1, \ldots P_n$ of P, $P = P_1 \| \ldots \| P_n$, are already translated by application of the rules $(21) - (25)$ below by specifying each of $i(P_1), \ldots, i(P_n)$, then :

$$i(P) \quad = \quad i(P_1) \quad \ldots \; i(P_n) \tag{20}$$

is the inverse of the complete process, the collection of the proof net trees.

Definition 12 (Inverse map) *The map $i(\cdot)$ strips off the indices at the same time as the proof net is reconstructed.*

$$i(A_i \rightarrow P \| A_i^\perp \rightarrow Q) \quad = \quad \frac{\begin{array}{cc} i(P) & i(Q) \\ \vdots & \vdots \\ A & A^\perp \end{array}}{CUT} \tag{21}$$

$$i(A_i \otimes B_j \rightarrow (A_i \rightarrow P \| B_j \rightarrow Q)) \quad = \quad \frac{\begin{array}{cc} i(P) & i(Q) \\ \vdots & \vdots \\ A & B \end{array}}{A \otimes B} \tag{22}$$

$$i(A_i + B_j \rightarrow (A_i \rightarrow P \| B_j \rightarrow P)) \quad = \quad \frac{\begin{array}{cc} i(P) & i(P) \\ \vdots & \vdots \\ A & B \end{array}}{A + B} \tag{23}$$

$$i(A_i \rightarrow S_k \| A_j^\perp \rightarrow S_k^\perp) \quad = \quad \overset{\sqcap}{A \; A^\perp} \tag{24}$$

$$i(\; (S_i \| S_{i+1}^\perp) \| \ldots \; \| (S_{i+n-1} \| S_{i+n}^\perp) \;) \quad = \quad \overset{\sqcap}{f(i) \; f((i+n)^\perp)} \tag{25}$$

Leave out the indices of the literals in $f(i), f((i+n)^\perp)$ in (25)*.*

Proposition 1 (Correspondence) *The reduction steps in the cut elimination process for multiplicative proof nets correspond step–by–step to the execution, $e(\cdot)$, of the translated CSP process :*

$$(*) \quad r(P) = i(e(t(P)))$$

Proof *The proof is by induction on the number of cut nodes in the original proof net.*

Base case $(*)$ **:** *Let the proof net P be cut free. We then have that r is the identity function. By $(E1)$, $e(\cdot)$ is also the identity. Only cases* (14), (15) *and* (16)*,* *apply to* $t(\cdot)$ *(in addition to* (12)*). The only cases from* $i(\cdot)$ *are* (22), (23) *and* (24) *(in addition to* (20)*.*

It remains to show :

$$(**) \quad P = i(t(P))$$

This is done by induction on the construction of the (cut free) proof net P.

Base case $(**)$ **:** *P is :*

$$\overbrace{A_i \quad A_j^{\perp}}^{k}$$

and :

$$\begin{aligned} t(P) &\overset{(16)}{=} A_i \rightarrow S_k \| A_j^{\perp} \rightarrow S_k^{\perp} \\ i(t(P)) &\overset{(24)}{=} P \end{aligned}$$

Inductive case $(**)$ *: There are two cases :* $\otimes$ *and* $+$.

(i)

$$P = \cfrac{\begin{matrix} P_1 & & P_2 \\ \vdots & & \vdots \\ A & & B \end{matrix}}{A \otimes B}$$

and we have that :

$$\begin{aligned} t(P) &\overset{(14)}{=} A_i \otimes B_j \rightarrow (A_i \rightarrow t(P_1) \| B_j \rightarrow t(P_2)) \\ i(t(P)) &\overset{(22)}{=} \cfrac{\begin{matrix} i(t(P_1)) & & i(t(P_2)) \\ \vdots & & \vdots \\ A & & B \end{matrix}}{A \otimes B} \\ &\overset{(IH(**))}{=} P \end{aligned}$$

(ii) *The case for* $+$ *is similar to the one above.*

Inductive case $(*)$**:** *There are two cases according to the two rules for reducing proof nets.*

(i) *P is to the left of* $\rightsquigarrow$, $r(P)$ *to the right.*

$$\cfrac{\cfrac{\begin{matrix} P_1 & & P_2 \\ \vdots & & \vdots \\ A & & B \end{matrix}}{A \otimes B} \qquad \cfrac{\begin{matrix} Q & & Q \\ \vdots & & \vdots \\ A^{\perp} & & B^{\perp} \end{matrix}}{A^{\perp} + B^{\perp}}}{CUT} \quad \rightsquigarrow \quad \cfrac{\begin{matrix} P_1 & & Q \\ \vdots & & \vdots \\ A & & A^{\perp} \end{matrix}}{CUT} \qquad \cfrac{\begin{matrix} P_2 & & Q \\ \vdots & & \vdots \\ B & & B^{\perp} \end{matrix}}{CUT}$$

We have :

$$\begin{aligned} t(P) &\overset{(13,14,15)}{=} [(A_i \otimes B_j \rightarrow (A_i \rightarrow t(P_1) \| B_j \rightarrow t(P_2)))] \| \\ &\qquad [(A_i^{\perp} + B_j^{\perp} \rightarrow (A_i^{\perp} \rightarrow t(Q) \| B_j^{\perp} \rightarrow t(Q)))] \\ e(t(P)) &\overset{(19)}{=} A_i \rightarrow t(P_1) \| B_j \rightarrow t(P_2) \| A_i^{\perp} \rightarrow t(Q) \| B_j^{\perp} \rightarrow t(Q) \end{aligned}$$

$$
\begin{array}{lcl}
 & \overset{(E2)}{=} & A_i \to e(t(P_1))\|B_j \to e(t(P_2))\|A_i^\perp \to e(t(Q))\|B_j^\perp \to e(t(Q)) \\
i(e(t(P))) & \overset{(21,20)}{=} & \dfrac{\begin{array}{cc} i(e(t(P_1))) & i(e(t(Q))) \\ \vdots & \vdots \\ A & A^\perp \end{array}}{CUT} \qquad \dfrac{\begin{array}{cc} i(e(t(P_2))) & i(e(t(Q))) \\ \vdots & \vdots \\ B & B^\perp \end{array}}{CUT} \\
 & \overset{(IH(*))}{=} & r(P)
\end{array}
$$

(ii) Again, P is to the left of $\rightsquigarrow$, $r(P)$ to the right.

$$
\begin{array}{cccc}
 & k & l & \\
A_j^\perp & \dfrac{A_i \quad A_i^\perp}{CUT} & & A_p \\
\vdots & & & \vdots
\end{array}
\quad \rightsquigarrow \quad
\begin{array}{cc}
k & l \\
A_j^\perp & A_p \\
\vdots & \vdots
\end{array}
$$

$$
\begin{array}{rcl}
t(P) & \overset{(16,13)}{=} & (\ldots \to (\cdot\|A_j^\perp \to S_k^\perp))\|A_i \to S_k\|A_i^\perp \to S_l^\perp\|(\ldots \to (A_p \to S_l\|\cdot)) \\
e(t(P)) & \overset{(19)}{=} & (\ldots \to (\cdot\|A_j^\perp \to S_k^\perp))\|(S_k\|S_l^\perp)\|(\ldots \to (A_p \to S_l\|\cdot)) \\
i(e(t(P))) & \overset{(25,20)}{=} & \begin{array}{cc} \multicolumn{2}{c}{k, l} \\ A_j^\perp & A_p \\ \vdots & \vdots \end{array} \\
 & = & r(P)
\end{array}
$$

In the second last line we made use of the facts :

$$f(k^\perp) = A_j^\perp, \quad f(k) = A_i, \quad f(l^\perp) = A_i^\perp, \quad f(l) = A_p$$

5 Concluding remarks

We have given an interpretation of normalization of MLL^- proof nets as a parallel computation in CSP. Proposition 1 states that the correspondence is step–by–step correct.

A natural next step would be to try to make (parallel) computational sense of the normalization process of a larger fragment of the proof net formalization of LL. These fragments all rely on "proof boxes" [5] which contain side premises and subsequently need commuting cases in the cut elimination[3].

There are, from a logical point of view, certain problems with the proof boxes. The most important is that the Church–Rosser property of the cut elimination process fails for the commutative cases[4] with proof boxes. There have been later developments of the theory of proof nets which have succeeded in freeing a larger fragment — first order multiplicative and additive LL without constants — of the proof boxes [6, 7, 3].

[3] In MLL^- the nodes $\otimes, +$ do not contain side premises — only the subformulas of the introduced conclusion formula. In section 2.1 we pointed out the consequence that there are only symmetrical cases in the cut elimination of this fragment.

[4] In fact the property fails for symmetrical cases in the exponential fragment. But this failure is "minor" as it is only a difference in the order in which a context $?\Gamma$ is weakened or contracted. This may be viewed as a slight defect of the syntax of proof nets.

[1, 2] consider interpretations of the cut elimination process of *all* of LL. They accordingly incorporate strictly more than the fragment considered by us. The focus is on (what corresponds to) the symmetrical cases in the cut elimination. The source of the failure of the Church–Rosser property, the commutative cases, are left out. In [1] an alternative to proof nets — called proof expressions — is introduced. The reduction rules for such proof expressions correspond to symmetrical cases of cut elimination. In [2] an approach much closer to ours is followed. The symmetrical cases of cut elimination of all of LL is interpreted in the π – calculus [11]. The π – calculus is a generalization of the kind of process algebras such as CSP and CCS to allow links (or "channels") as first class objects which may be transmitted as values along a link. This extra strength is used in one place : to express, more directly than ours with $(S_k \| S_l^{\perp})$, the collapsing of the axiom links.

It seems to us to be two possible strategies to adopt in a interpreting LL computationally :

- One could let go of the intention of giving an interpretation of the proof objects of LL as formalized in the theory of proof nets, and instead focus on either a reformulation or a restriction to the symmetrical cuts. This underlies the elegant approaches of [1, 2]. In favour of this, one may hold that the failure of the Church–Rosser property with both the sequent and proof net formalization of LL shows the shortcomings of the two formalization and that there should be no reason to keep to them too literally.

- If one sticks to (say) the proof net formalization of LL, one have to make sense of the failure of the Church–Rosser property. From a computational point of view, this amounts to giving an interpretation where the order in which two steps are executed play a role. But this need not be as dramatic as from a logical point of view. It is not a rare situation in programming. Let A and B be :

  A : `send(x); receive(y)` B : `receive(y); send(x)`

  They may very well not yield the same result (take `x` to contain `y` : in A the old value for `x` is sent, whereas in B the new one depending on the value of `y` just received is sent).

References

[1] Samson Abramsky. Computational interpretations of linear logic. Research report DOC 90/20, Imperial Collage, England, 1990. also in Theoretical Computer Science, special issue in 1991 on 1990 Workshop of Meth. Found. Prog. Semantics, to apprear.

[2] Samson Abramsky. Proofs as processes. (manuscript), 1991.

[3] Gianluigi Bellin. *Mechanizing proof theory : resource–aware logics and proof transformations to extract implicit information.* PhD thesis, Stanford, 1990. STAN–CS–90–1319.

[4] Jean-Yves Girard. Linear logic and parallelism. In M.V. Zilli, editor, *Mathematical models for the semantics of parallelism*, pages 166–182. Springer–Verlag, 1986. Lecture Notes in Computer Science 280.

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

[6] Jean-Yves Girard. Quantifiers in linear logic. In *Atti del Congreso temi e prospettive della logica e della filosofia della sciences contempranee*, pages 95–130, Bologna, Italy, 1988. Clueb.

[7] Jean-Yves Girard. Quantifiers in linear logic II. (preprint), 1991.

[8] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1989.

[9] C.A.R. Hoare. *Communicating sequential processes.* Prentice–Hall, 1985.

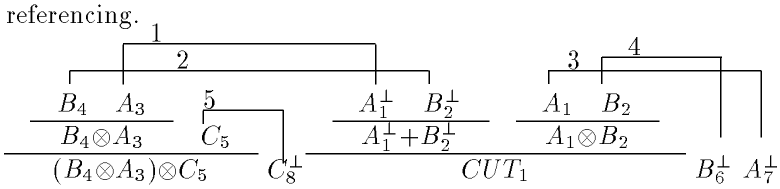
[10] Narciso Martì-Oliet and José Meseguer. From Petri nets to linear logic. SRI - CSL - 89 - 4, SRI International, Stanford, 1989.

[11] R. Milner, J.G. Parrow, and D.J. Walker. A calculus for mobile processes, parts I and II. ECS-LFCS-98-85 and -86, LFCS, Dept. of computer science, Edinburgh University, 1989.

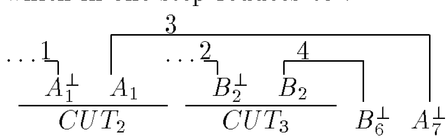
[12] Robin Milner. *A calculus for communicating systems.* Springer–Verlag, 1980. Lecture Notes in Computer Science 92.

A An example

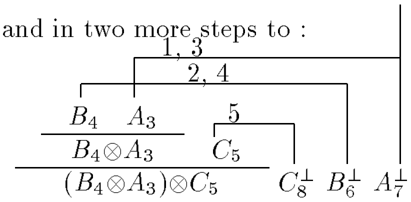
We consider the example from [8]. First the normalization process with proof nets is given, then the corresponding translated process. Cut–nodes have been numbered to ease referencing.



which in one step reduces to :



and in two more steps to :



The translation of the original proof net reads :

$$
\begin{array}{l}
\overbrace{(B_4 \otimes A_3) \otimes C_5 \to [B_4 \otimes A_3 \to (B_4 \to S_2 \| A_3 \to S_1)] \| C_5 \to S_5}^{P_1} \| \overbrace{C_8^\perp \to S_5^\perp}^{P_2} \| \\
\overbrace{A_1^\perp + B_2^\perp \to (A_1^\perp \to S_1^\perp \| B_2^\perp \to S_2^\perp) \|}^{P_3} \\
\overbrace{A_1 \otimes B_2 \to (A_1 \to S_3 \| B_2 \to S_4)}^{P_4} \| \overbrace{B_6^\perp \to S_4^\perp}^{P_5} \| \overbrace{A_7^\perp \to S_3^\perp}^{P_6}
\end{array}
$$

We write out the alphabets according to definition 10 before describing the cut elimination and the execution of the corresponding process.

The proof net trees of the proof net are $P_1, \ldots, P_6$. Their alphabets are :

$$
\begin{array}{rcl}
\alpha(P_1) & = & \{(B_4 \otimes A_3) \otimes C_5, B_4 \otimes A_3, C_5, B_4, A_3\} \\
\alpha(P_2) & = & \{C_8^\perp\} \\
\alpha(P_3) & = & \{A_1^\perp + B_2^\perp, A_1^\perp, B_2^\perp\} \\
\alpha(P_4) & = & \{A_1 \otimes B_2, A_1, B_2\} \\
\alpha(P_5) & = & \{B_6^\perp\} \\
\alpha(P_6) & = & \{A_7^\perp\} \\
\alpha(S_2) & = & \{(B_4 \otimes A_3) \otimes C_5, B_4 \otimes A_3, B_4\} \\
\alpha(S_1) & = & \{A_3\} \\
\alpha(S_5) & = & \{C_5\} \\
\alpha(S_5^\perp) & = & \{C_8^\perp\} \\
\alpha(S_1^\perp) & = & \{A_1^\perp + B_2^\perp, A_1^\perp\} \\
\alpha(S_2^\perp) & = & \{B_2^\perp\} \\
\alpha(S_3) & = & \{A_1 \otimes B_2, A_1\} \\
\alpha(S_4) & = & \{B_2\} \\
\alpha(S_4^\perp) & = & \{B_6^\perp\} \\
\alpha(S_3^\perp) & = & \{A_7^\perp\}
\end{array}
$$

An illustration of observation $(T1)$ could be to consider the alphabets of the left and right subtrees of $\mathsf{Btree}(P_1)$:

$$
\alpha(B_4 \otimes A_3 \to (B_4 \to S_2 \| A_3 \to S_1)) \cap \alpha(C_5 \to S_5) = \{B_4 \otimes A_3, B_4, A_3\} \cap \{C_5\} = \emptyset
$$

We return to the description of the normalization. After eliminating CUT_1 we get :

$$
\cdots \| (A_1^\perp \to S_1^\perp \| A_1 \to S_3) \| (B_2^\perp \to S_2^\perp \| B_2 \to S_4) \| \cdots
$$

...and eliminating CUT_2, CUT_3 we obtain :

$$
\cdots \| \overbrace{(S_1^\perp \| S_3)}^{(a)} \| \overbrace{(S_2^\perp \| S_4)}^{(b)} \| \cdots
$$

where (a) and (b) express the presence of the new axiom links going from :

$$f(1) = A_3 \text{ to } f(3^{\perp}) = A_7^{\perp}, \quad \text{and} \quad f(2) = B_4 \text{ to } f(4^{\perp}) = B_6^{\perp}$$

as given by (25).