HAL
open science

Parallelism and Synchronization in an Infinitary Context

Ugo Dal Lago, Claudia Faggian, Benoît Valiron, Akira Yoshimizu

To cite this version:

Ugo Dal Lago, Claudia Faggian, Benoît Valiron, Akira Yoshimizu. Parallelism and Synchronization in an Infinitary Context. LICS 2015, Jul 2015, Kyoto, Japan. 10.1109/LICS.2015.58 . hal-01231813

HAL Id: hal-01231813

https://inria.hal.science/hal-01231813v1

Submitted on 20 Nov 2015

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.

Parallelism and Synchronization in an Infinitary Context

Ugo Dal Lago
Univ. di Bologna & INRIA

Claudia Faggian
CNRS & Univ. Paris Diderot

Benoît Valiron
CentraleSupélec & LRI, Univ. Paris Sud

Akira Yoshimizu
Univ. of Tokyo

***Abstract*—We study multitoken interaction machines in the context of a very expressive linear logical system with exponentials, fixpoints and synchronization. The advantage of such machines is to provide models in the style of the Geometry of Interaction, *i.e.*, an interactive semantics which is close to low-level implementation. On the one hand, we prove that despite the inherent complexity of the framework, interaction is guaranteed to be deadlock-free. On the other hand, the resulting logical system is powerful enough to embed PCF and to adequately model its behaviour, both when call-by-name and when call-by-value evaluation are considered. This is not the case for single-token stateless interactive machines.**

I. Introduction

What is the inherent parallelism of higher-order functional programs? Is it possible to turn λ-terms into low-level programs, at the same time exploiting this parallelism? Despite great advances in very close domains, these questions have not received a definite answer, yet. The main difficulties one faces when dealing with parallelism and functional programs are due to the higher-order nature of those programs, which turns them into objects having a non-trivial interactive behaviour.

The most promising approaches to the problems above are based on Game Semantics [1], [14] and the Geometry of Interaction [12] (GoI), themselves tools which were introduced with purely semantic motivations, but which have later been shown to have links to low-level formalisms such as asynchronous circuits [10]. This is especially obvious when Geometry of Interaction is presented in its most operational form, namely as a token machine [7].

Most operational accounts on the Geometry of Interaction are in *particle-style*, i.e., a *single* token travels around the net; this is largely due to the fact that parallel computation without any form of synchronization nor any data sharing is not particularly useful, so having multiple tokens would not add anything to the system. While some form of synchronization was implicit in earlier presentations of GoI, the latter has been given a proper status only recently, with the introduction of SMLL [4], where *multiple* tokens circulate simultaneously, and also *synchronize* at a new kind of node, called a *sync node*. All this has been realized in a minimalistic logic, namely multiplicative linear logic, a logical system which lacks any copying (or erasing) capability and, thus, is not an adequate model of realistic programming languages (except purely linear ones, whose role is relevant in quantum computation [24]).

Multitoken GoI machines are relatively straightforward to define in a linear setting: all *potential* sources of parallelism give rise to *actual* parallelism, since erasing and copying are simply forbidden. As a consequence, managing parallelism, and in particular the spawning of new tokens, is easy: the mere syntactical occurrence of a source of parallelism triggers the creation of a new token. Concretely, these sources of parallelism are *unit nodes* (when thought logically), or *constants* (when read through the lenses of functional programming). The reader will find an example in Section II, Fig. 1.

But can all this scale to more expressive proof theories and programming formalisms? If programs or proofs are allowed to copy or erase portions of themselves, the correspondence between potential and actual parallelism vanishes: any occurrence of a unit node can possibly be erased, thus giving rise to *no* token, or copied, thus creating *more than one* token. The underlying interactive machinery, then, necessarily becomes more complex. But *how*? The solution we propose here relies on linear logic itself: it is the way copying and erasing are handled by the exponential connectives of linear logic which gives us a way out. We find the resulting theory simple and elegant.

In this paper we generalize the ideas behind SMLL in giving a proper status to synchronization and parallelism in GoI. We show that multiple tokens and synchronization can work well together in a *very expressive* logical system, namely multiplicative linear logic with *exponentials*, *fixpoints*, and *units*. The resulting system, called SMEYLL, is then general enough to simulate universal models of functional programming: we prove that PCF can be embedded into SMEYLL, both when call-by-name and call-by-value evaluation are considered. The latter is not the case for single-token machines, as we illustrate in Section II.

An extended version of this paper with proofs and more details is available [5]. The first author is partially supported by the ANR project 12IS02001 PACE.

Contributions

This paper's main contributions can be summarized as follows:

- *An Expressive Logical System.* We introduce SMEYLL nets, whose expressiveness is increased over MELL nets by several constructs: we have *fixpoints* (captured by the Y-box), an operator for *synchronization* (the sync node), and a *primitive conditional* (captured by the $\perp$-box). The presence of fixpoints forces us to consider a restricted notion of reduction, namely closed *surface reduction* (*i.e.*,

reduction never takes place inside a box). Cuts can *not* be eliminated (in general) from SMEYLL proofs, as one expects in a system with fixpoints. Reduction, however, is proved to be *deadlock-free*, *i.e.*, normal forms cannot contain surface cuts.

- *A Multitoken Interactive Machine.* SMEYLL nets are seen as interactive objects through their synchronous interactive abstract machine (SIAM in the following). Multiple tokens circulate around the net simultaneously, and synchronize at sync nodes. We prove that the SIAM is an *adequate computational model*, in the sense that it precisely reflects normalization through machine execution. The other central result about the SIAM is *deadlock-freeness*, *i.e.*, if the machine terminates it does so in a final state. In other words, the execution does not get stuck, which in principle could happen as we have several tokens running in parallel and to which we apply guarded operators (*e.g.*, synchronization). Our proof comes from the interplay of nets and machines: we transfer *termination* from machines to nets, and then transfer *back deadlock-freeness* from nets to machines.
- *A Fresh Look at CBV and CBN.* A slight variation on SMEYLL nets, and the corresponding notion of interactive machine, is shown to be an adequate model of reduction for Plotkin's PCF [22]. This works both for call-by-name and call-by-value evaluation and, noticeably, the *same* interactive machine is shown to work in *both* cases: what drives the adoption of each of the two mechanisms is, simply, the translation of terms into proofs. What is surprising here is that CBV can be handled by a stateless interactive machine, even without the need to go through a CPS translation. This is essentially due to the presence of multiple tokens.
- *New Proof Techniques.* Deadlock-freeness is a key issue when working with multitoken machines. A direct scheme to prove it (the one used in [4]) would be: (i) prove cut elimination for the nets, (ii) prove soundness for the machine, and (iii) deduce deadlock-freeness from (i) and (ii). However, in a setting with fixpoints, cut elimination is not available because termination simply does not hold[1]. Instead, we develop a new technique, which heavily exploit the interplay between net rewriting and the multitoken machine. Namely, we *transfer* termination of the machine (including termination as a deadlock) into termination of the nets. This combinatorial technique is novel and uses multiple tokens in an essential way. It appears to be of technical interest in its own.

Related Work

Almost thirty years after its introduction, the literature on GoI is vast. Without any aim of being exhaustive, we only mention the works which are closest in spirit to what we are doing here.

[1]Even without fixpoints, there is to the authors' knowledge no direct combinatorial proof of termination for surface reduction.

The fact that GoI can be turned into an implementation scheme for purely functional (but expressive) λ-calculi, has been observed since the beginning of the nineties [7], [17]. Among the different ways GoI can be formulated, both (directed) virtual reduction and bideterministic automata have been shown to be amenable to such a treatment. In the first case, parallel implementations [20], [21] have also been introduced. We claim that the kind of parallel execution we obtain in this work is different, being based on the underlying automaton and not on virtual reduction.

The fact that GoI can simulate call-by-name evaluation is well-known, and indeed most of earlier results relied on this notion of reduction. As in games [2], call-by-value requires a more sophisticated machinery to be handled by GoI. This machinery, almost invariably, relies on effects [13], [23], even when the underlying language is purely functional. This paper suggests an alternative route, which consists in making the underlying machine parallel, nodes staying stateless.

Another line of work is definitely worth mentioning here, namely Ghica and coauthors' Geometry of Synthesis [8], [9], in which GoI suggests a way to compile programs into circuits. The obtained circuit, however, is bound to be sequential, since the interaction machinery on which everything is based is particle-style.

On the side of nets, Y-boxes allow us to handle *recursion*. A similar box was originally introduced by Montelatici [19], even though in a polarized setting. Our Y-box differs from it both in the typing and in the dynamics; these differences are what make it possible to build a GoI model.

II. On Multiple Tokens and the Exponentials

In this section, we will explain through a series of examples *how* one can build a multitoken machine for a non-linear typed λ-calculus, and *why* this is not trivial.

Let us first consider a term computing a simple arithmetical expression, namely $M = (\lambda x.\lambda y.x + y)(4 - 2)(1 + 2)$. This term evaluates to 5 and is purely linear, i.e. the variables x and y appear exactly once in the body of the abstraction. How could one evaluate this term trying to exploit the inherent parallelism in it? Since we *a priori* know that the term is linear, we know that the subexpressions $S = (4-2)$ and $T = (1+2)$ are indeed needed to compute the result, and thus can be evaluated in parallel. The subexpression $x+y$ could be treated itself this way, but its arguments are missing, and should be waited for. What we have just described is precisely the way the multitoken machine for SMLL works [4], as in Fig. 1 (left): each constant in the underlying proof gives rise to a separate token, which flows towards the result. Arithmetical operations act as synchronization points. Now, consider a slight variation on the term M above, namely $N = (\lambda x.\lambda y.x{+}x)(4{-}2)(1{+}2)$. The term has a different normal form, namely 4, and is *not* linear, for two different reasons: on the one hand, the variable x is used twice, and on the other, the variable y is not used at all. How should one proceed, then, if one wants to evaluate the term in parallel? One possibility consists in evaluating subexpressions *only if* they are really needed. Since

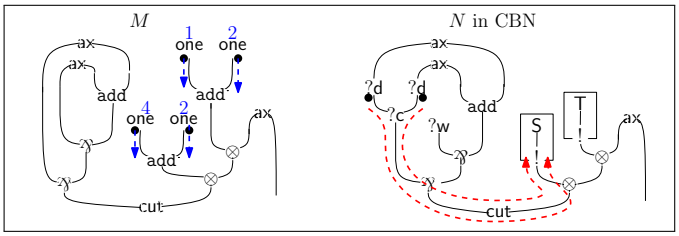


Fig. 1. Actual vs. Potential Parallelism.

the subexpression $x+x$ is of course needed (it is, after all, the result!), one can start evaluating it. The value of the variable x, as a consequence, is needed, and the subexpression it will be substituted for, namely $4-2$, must itself be evaluated. On the other hand, $1+2$ should not be evaluated, simply because its value does not contribute to the final result. This is precisely what call-by-name evaluation actually do. The interactive machine which we define in this paper captures this process. It has to be noticed, in particular, that discovering that one of the subexpressions is needed, while the other is not, requires some work. The way we handle all this is strongly related to the structure of the exponentials in linear logic. We give the CBN translation of N in Fig. 1 (right). The two rightmost subterms are translated into exponential boxes (where S is the net for $4-2$ and T for $1+2$), which serve as *boundaries* for parallelism: whatever potential parallelism a box includes, must be triggered before giving rise to an actual parallelism. Each of the occurrences of the variable x triggers a new kind of token, which starts from the dereliction nodes ($?d$) at the surface and whose purpose is precisely to look for the box the variable will be substituted for. We call these *dereliction tokens*.

What happens if we rather want to be consistent with call-by-*value* evaluation? In this case, both subterms $(4-2)$ and $(1+2)$ in the term N above should be evaluated. Let us however consider a more extreme example, in which call-by-name and call-by-value have different *observable* behaviors, for example the term $L=(\lambda x.1)\Omega$, where $\Omega=(\lambda x.xx)(\lambda x.xx)$. The call-by-value evaluation of L gives rise to divergence, while in call-by-name L evaluates to 1. Something extremely interesting happens here. We give the call-by-value translation of L in Fig. 2. First of all, we observe that a standard *single-*

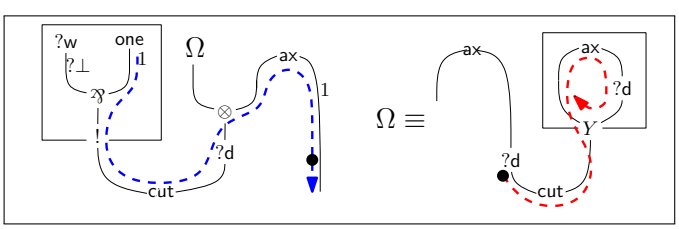


Fig. 2. The CBV-translation of $(\lambda x.1)\Omega$.

token machine would start from the conclusion, find the node one, and exit again: such a machine would simply converge on the term L. When running on the term Ω alone, the machine would diverge, but as subterm of L, Ω is never reached, so the machine's behaviour on L is not the one which we would expect in call-by-value. Our multitoken machine, instead, simultaneously launches tokens from all dereliction nodes at surface: the dereliction token coming out of Ω (represented on the right in Fig. 2) reaches the Y-box, and makes the machine diverge.

We end this section by stressing that the interactive machine we use is the same, and that this machine correctly models CBN and CBV, solely depending on the chosen translation of terms into nets. The call-by-name translation of L puts the subterm Ω in a box which is simply unreachable from the rest of the net (as in the case of T in Fig. 1), and our machine converges as expected. The call-by-value translation of L, on the other hand, does *not* put Ω inside a box. As a consequence, there is no barrier to the computation to which Ω gives rise—the same as if Ω would be alone—and our machine correctly diverges. This is the key difficulty in any interactive treatment of CBV, and we claim that the way we have solved it is novel.

III. Nets and a Multitoken Interactive Machine

We start with an overview of this section, which is divided into four parts.

Nets and Their Dynamics: SMEYLL nets come with *rewriting* rules, which provide an operational semantics for them, and with a *correctness* criterion, which ultimately guarantees that nets rewriting is deadlock-free.

Multitoken Machines: On any net we define a *multitoken* machine, called SIAM, which provides an effective computational model in the style of GoI. A fundamental property we need to check for the machine is *deadlock-freeness*, *i.e.*, if the machine terminates it does so in a final state. From the beginnings of linear logic, the correctness criterion of nets has been interpreted as deadlock-freeness in distributed systems [3]; this is also the case for MELLS. Here, however, we work with surface reduction, and we have fixpoints. For these reasons, a rather refined approach is needed.

The Interplay Between Nets and Machines: Nets rewriting and the SIAM are tightly related. We establish the following results. Let R denote a net, $\mathcal{M}_R$ its machine, and $\rightsquigarrow$ the net rewriting relation. First of all, we know that (i) if R is cut-and-sync-free, the machine $\mathcal{M}_R$ terminates in a final state. On the net hand, we establish that (ii) *net rewriting is deadlock-free*: if no reduction is possible from R, then R is cut-and-sync-free. On the machine side, we establish that (iii) if $R \rightsquigarrow S$ then $\mathcal{M}_R$ converges/deadlocks iff the same holds for $\mathcal{M}_S$. We then use the multitoken paradigm to provide a decreasing parameter for net rewriting, and establish that (iv) if $\mathcal{M}_R$ terminates, then R has no infinite sequence of reductions. Putting all this together, it follows that *multitoken machines are deadlock-free*.

Computational Semantics: Finally, by using the machine representation, we associate a denotational semantics to nets, which we prove to be sound with respect to net reduction.

A. Nets and Their Dynamics.

In this section we introduce SMEYLL nets, which are a generalization of MELL proof nets. For a detailed account

on proof nets, we refer the reader to Laurent's notes [15]: our approach to correctness, as well as the way to deal with weakening, is very close to the one described there.

1) Formulas: The language of SMEYLL *formulas* is identical to the one for MELL. For the sake of simplifying the presentation, we exclude propositional variables and assume that the only *atomic formulas are the units* (including propositional variables is straightforward but more verbose; we deal with it in [5]). The language of formulas is therefore:

$$A ::= 1 \;\Big|\; \bot \;\Big|\; A \otimes A \;\Big|\; A \parr A \;\Big|\; !A \;\Big|\; ?A.$$

Linear negation $(\cdot)^\perp$ is extended into an involution on all formulas as usual: $A^{\perp\perp} \equiv A$, $1^\perp \equiv \bot$, $(A \otimes B)^\perp \equiv A^\perp \parr B^\perp$, $(!A)^\perp \equiv ?A^\perp$. Linear implication is a defined connective: $A \multimap B \equiv A^\perp \parr B$.

Atoms and connectives of linear logic are usually divided in two classes: positive and negative. Here however, we define *positive* (denoted by P) and *negative* (denoted by N) those formulas which are built from units in the following way: $P ::= 1 \;\Big|\; P \otimes P$, and $N ::= \bot \;\Big|\; N \parr N$. So in particular, there are formulas which are neither positive nor negative, e.g. $\bot \parr 1$.

2) Structures: A SMEYLL *structure* is a finite labeled *directed* graph built over the alphabet of nodes which is represented in Fig. 3 (where the *orientation* is the top-bottom one). All edges have a source, but some edges may have no target; such dangling edges are called the *conclusions of the structure*. The edges are labeled with SMEYLL formulas; the label of an edge is called its *type*. We call those edges which are represented below (resp. above) a node *conclusions* (resp. *premisses*) of the node. We will often say that a node "has a conclusion (premiss) A" as shortcut for "has a conclusion (premiss) of type A". When we need more precision, we distinguish between an edge and its type, and we use variables such as e, f for the edges.

The nodes !, Y and $\bot$ are called *boxes*. One among their conclusions (the leftmost ones in Fig. 3, which have type $!A$, $!A$ and $\bot$, respectively) is said to be *principal*, the other ones being *auxiliary*. !-boxes and Y-boxes are *exponential*. An exponential box is *closed* if it has no auxiliary conclusions. To each box is associated, in an inductive way, a structure which is called the *content* of the box. To the !-box we associate a structure with conclusions $A, ?\Gamma$. To the Y-box corresponds a structure of conclusions $A, ?A^\perp, ?\Gamma$. To the $\bot$-box is associated a structure of non-empty conclusions Γ, together with a new node bot of conclusion $\bot$. We represent a box b and its content S as in Fig. 4. With a slight abuse of terminology, the nodes and edges of S are said to be *inside* b. Similarly, a crossing of any box's border is said to be a *door*, and we often speak of premiss and conclusion *of a* (principal or auxiliary) *door*. Note that the principal door of the Y-box (marked by Y) has premisses $A, ?A^\perp$ and conclusion $!A$.

A node *occurs at depth 0* or *at surface* in the structure R if it is a node of R, while it *occurs at depth* $n+1$ in R if it occurs at depth n in a structure associated to a box of R. Please observe that nets being defined inductively, the depth of nodes is always finite. The sort of each node induces constraints on

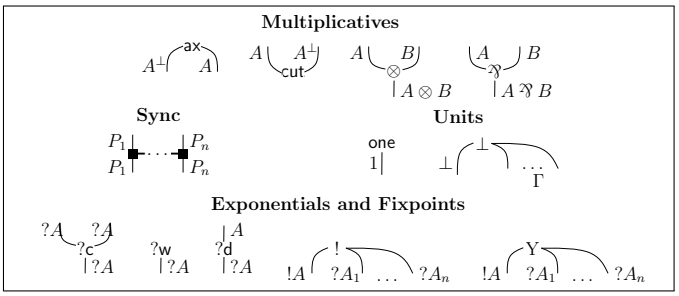


Fig. 3. SMEYLL Nodes.

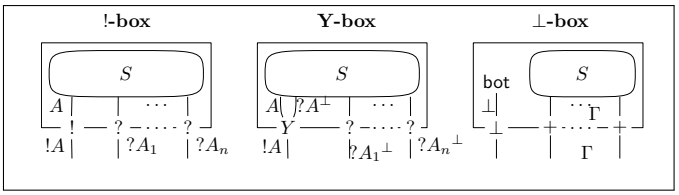


Fig. 4. SMEYLL Boxes.

the number and the labels of its premisses and conclusions, which are shown in Fig. 3. We observe that the $\bot$-box is *the same* as in [11] and corresponds to the sequent calculus rule $\dfrac{\vdash \Gamma}{\vdash \bot, \Gamma}$. All nodes are standard except sync nodes and Y-boxes, which need some further explanation:

- Y-boxes model *recursion* (more on this when we introduce the reduction rules). Proof-theoretically, the Y-box corresponds to adding the following fix-point sequent calculus rule to MELL:
$$\frac{\vdash A, ?A^\perp, ?\Gamma}{\vdash !A, ?\Gamma}\; Y$$
- Sync nodes model *synchronization points*. A sync node has n premisses and n conclusions; for each i ($1 \leq i \leq n$) the i-th premiss and the *corresponding* i-th conclusion are typed by the *same* formula, which needs to be *positive*.

A relevant class of structures is that of positive ones. A structure R is *positive* if all its conclusions are positive formulas. This *does not* mean that all formulas occurring in R are positive: R can be very complex; the constraint only deals with R's conclusions. Positivity captures the class of nets to which we are going to give computational meaning in Section IV.

3) Correctness: A *net* is a structure which fulfills a *correctness criterion* defined by means of switching paths (see [15]). A *switching path* on the structure R is an undirected path[2] such that (i) for each $\parr$-or-$?c$-node, the path uses at most one of the two *premisses*, and (ii) for each sync node, the path uses at most one of the *conclusions*. The former condition is standard, the latter condition rules out paths which bounce on sync nodes "from below": a path crossing a sync node

[2]By path, in this paper we always mean a *simple path* (no repetition of either nodes or edges).

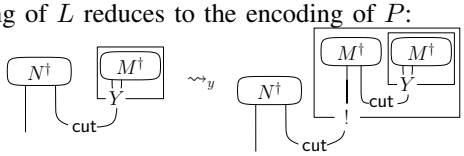
may traverse one premiss and one conclusion, or traverse two distinct premisses. A structure is *correct* if:

1. none of its switching paths are cyclic, and
2. the content of each of its boxes is itself correct.

The reader familiar with linear logic correctness criteria did probably notice that the only condition we require is acyclicity, and that connectedness is simply not enforced (as, e.g., in Danos and Regnier's criterion [6]). Actually, the only role of connectedness consists in ruling out the so-called Mix rule from the sequent calculus. This is not relevant in our development, so we will ignore it. An advantage of accepting the Mix rule is that we do not need extra conditions for dealing with weakening. A similar approach is adopted by Laurent [15]. In the following, when we talk of MELL (resp. MLL), we actually always mean MELL + Mix (resp. MLL + Mix).

4) Net Reduction: Reduction rules for nets are sketched in Fig. 5. Reduction rules can be applied only at surface (*i.e.*, when the redex occurs at depth 0), and not in an arbitrary context. Moreover, observe that reduction rules involving an *exponential* box can only be applied when the box is *closed*, *i.e.*, when it has no auxiliary doors. We write $\rightsquigarrow$ for the rewriting relation induced by these rules. Some reduction rules deserve some further explanations:

- The y-rule unfolds a Y-box, this way modeling recursion. The intuition should be clear when looking at the translation of the PCF term $L = \texttt{letrec}\, f\, x = M\, \texttt{in}\, N$, which reduces to the explicit substitution of f by $\lambda x.\texttt{letrec}\, f\, x = M\, \texttt{in}\, M$ in N, call it P. Indeed, the encoding of L reduces to the encoding of P:



(where $M^\dagger$ and $N^\dagger$ stand for the encodings of M and N, respectively). When (and only if!) N recursively calls f, the corresponding d node "opens" the !-box for the first iteration of f; if f further uses a recursive call of itself, the Y-box again turns into yet another !-box and is opened, and so on.
- The $s.el$-rule erases a sync link whose premisses are *all* conclusions of one nodes.
- The w-rule, corresponding to a cut with weakening, *deletes* the redex (because the box has no auxiliary conclusions).
- The $bot.el$-rule opens a $\perp$-box.

It is immediate to check that correctness is preserved by all reduction rules.

Lemma 1. *If R is a net and $R \rightsquigarrow S$, then S is itself a net.*

Since the constraints exclude most of the commutations which are present in MELL, rewriting enjoys a strong form of confluence:

Proposition 2 (Confluence and Uniqueness of Normal Forms)**.** The rewriting relation $\rightsquigarrow$ has the following properties:

1. it is confluent and normal forms are unique;

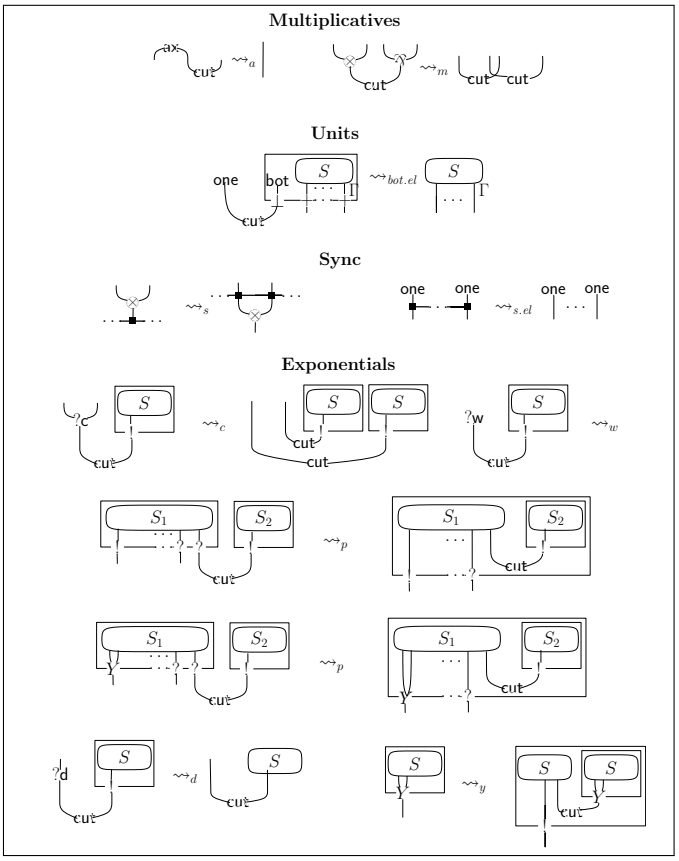


Fig. 5. SMEYLL Net Rewriting Rules.

2. any net weakly normalizes iff it strongly normalizes.

Proof. The only critical pairs are the trivial ones of MLL, leading to the same net. Therefore, reduction enjoys a diamond property (uniform confluence): if $R \rightsquigarrow S$ and $R \rightsquigarrow T$, then either $S = T$ or there exists U such that $S \rightsquigarrow U$ and $T \rightsquigarrow U$. (1) and (2) are direct consequences. $\square$

The strict constraints on rewriting, however, render cut elimination non-trivial: it is not obvious that a reduction step is available whenever a cut is present. We need to prove that in presence of a cut, there is always a valid redex (*i.e.*, it is surface, and any exponential box acted upon is closed). The main difficulty comes from $\perp$-boxes, as they can hide large parts of the net, and in particular dereliction nodes which may be necessary to fire a reduction. However, the following establishes that as long as there are cuts or syncs, it is always possible to perform a valid reduction.

Theorem 3 (Deadlock-Freeness for Nets)**.** Let R be a positive SMEYLL net. If R contains cuts or sync nodes, then there exists S such that $R \rightsquigarrow S$.

The rather long proof is given in [5]. The key element is the definition of an order on the boxes occurring in R at depth 0, the existence of which relies on the correctness criterion. The order captures the dependency among boxes, *i.e.*, exposes the order in which cuts are eliminated.

Corollary 4 (Cut Elimination)**.** Let R be a positive SMEYLL net. If $R \rightsquigarrow^* S$ and S cannot be further reduced, then S is a cut free net which only contains one and $\otimes$ nodes, *i.e.*, essentially a (very simple) MLL net.

Discussion on Positivity: Positivity of the nets is an assumption which in this section we use *to simplify auxiliary lemmas*; it will not appear in our main result, namely Theorem 11.

B. SIAM

All along this section, R indicates a SMEYLL structure (with no hypothesis on correctness, unless otherwise stated).

1) Preliminary Notions: Some auxiliary definitions are needed before we can introduce our interactive machines. *Exponential signatures* are defined by the following grammar

$$\sigma ::= * \;\Big|\; l(\sigma) \;\Big|\; r(\sigma) \;\Big|\; \lceil \sigma, \sigma \rceil \;\Big|\; y(\sigma, \sigma),$$

while *stacks* are defined as follows

$$s ::= \epsilon \;\Big|\; l.s \;\Big|\; r.s \;\Big|\; \sigma.s \;\Big|\; \delta,$$

where ϵ is the empty stack and . denotes concatenation (and, thus, $s.\epsilon = s$). Given a formula A, a stack s *indicates an occurrence* α *of an atom* (resp. *an occurrence* μ *of a modality*) in A if $s[A] = \alpha$ (resp. $s[A] = \mu$), where $s[A]$ is defined as follows:

- $\epsilon[\alpha] = \alpha$,
- $\sigma.\delta[\mu B] = \mu$,
- $\sigma.t[\mu B] = t[B]$ whenever $t \neq \delta$,
- $l.t[B \Box C] = t[B]$ and $r.t[B \Box C] = t[C]$, where $\Box$ is either $\otimes$ or $\parr$.

We observe that a stack can indicate a modality only if its head is δ.

Example 5. Given the formula $A = !(\bot \otimes !1)$, the stack $*.\delta$ indicates the first occurrence of !, $*.r.*.\delta[A]$ gives the second occurrence of !, and $*.\delta, *.l[A] = \bot$.

The set of R's *positions* $\mathtt{POS}_R$ contains all the triples in the form (e, s, t), where:

1. e is an edge of R,
2. the *formula stack* s is either δ or a stack which indicates an occurrence of atom or modality in the type A of e,
3. the *box stack* t is a stack of n exponential signatures, where n is the number of exponential boxes inside which e appears.

We use the metavariables $\mathbf{s}$ and $\mathbf{p}$ to indicate positions. For each position $\mathbf{p} = (e, s, t)$, we define its *direction* $\mathtt{dir}(\mathbf{p})$ as *upwards* ($\uparrow$) if s indicates an occurrence of ! or of negative atom, as *downwards* ($\downarrow$) if s indicates an occurrence of ? or of positive atom, as *stable* ($\leftrightarrow$) if $s = \delta$ or if the edge e is the conclusion of a bot node. A position $\mathbf{p} = (e, s, \epsilon)$ is *initial* (resp. *final*) if e is a conclusion of R, and $dir(\mathbf{p})$ is $\uparrow$ (resp. $\downarrow$). For simplicity, on initial (final) positions, we require all exponential signatures in s to be $*$. So for example, if $!(\bot \otimes !1)$ is a conclusion of R, there is one final position ($s = *.r.*$), and three initial positions (the three stacks given in Example 5). The following subsets of $\mathtt{POS}_R$ play a crucial role in the definition of the machine:

- the set $\mathtt{INIT}_R$ of all *initial positions*;
- the set $\mathtt{FIN}_R$ of all *final positions*;
- the set $\mathtt{ONES}_R$ of positions (e, ϵ, t) where e is the conclusion of a one node;
- the set $\mathtt{DER}_R$ of positions $(e, *.\delta, t)$ where e is the conclusion of a ?d node;
- the starting positions $\mathtt{START}_R = \mathtt{INIT}_R \cup \mathtt{ONES}_R \cup \mathtt{DER}_R$;
- the set $\mathtt{STABLE}_R$ of the positions $\mathbf{p}$ for which $\mathtt{dir}(\mathbf{p}) = \leftrightarrow$.

The multitoken machine $\mathcal{M}_R$ for R consists of a set of *states* and a *transition* relation between them. These are the topics of the following two subsections.

2) States: A state of $\mathcal{M}_R$ is a snapshot description of the tokens circulating in R. We also need to keep track of the positions where the tokens started, so that the machine only uses each starting position once. Formally, a *state* $\mathbf{T} = (\mathtt{Current}_\mathbf{T}, \mathtt{Dom}_\mathbf{T})$ is a set of positions $\mathtt{Current}_\mathbf{T} \subseteq \mathtt{POS}_R$ together with a set of positions $\mathtt{Dom}_\mathbf{T} \subseteq \mathtt{START}_R$. Intuitively, $\mathtt{Current}_\mathbf{T}$ describes the current position of the tokens, and $\mathtt{Dom}_\mathbf{T}$ keeps track of which starting positions have been used[3]. A state is *initial* if $\mathtt{Current}_\mathbf{T} = \mathtt{Dom}_\mathbf{T} = \mathtt{INIT}_R$. We indicate the (unique) initial state of $\mathcal{M}_R$ by $\mathbf{I}_R$. A state $\mathbf{T}$ is *final* if all positions in $\mathtt{Current}_\mathbf{T}$ belong to either $\mathtt{FIN}_R$ or $\mathtt{STABLE}_R$. The set of all states will be denoted by $\mathcal{S}_R$. Given a state $\mathbf{T}$ of $\mathcal{M}_R$, we say that *there is a token in* $\mathbf{p}$ if $\mathbf{p} \in \mathtt{Current}_\mathbf{T}$. We use expressions such as "a token moves", "crosses a node", in the intuitive way.

3) Transitions: The transition rules of $\mathcal{M}_R$ are given by the transitions described in Fig. 6 (where $\Box$ stands for either $\otimes$ or $\parr$). The rules marked by (i)–(iii) make the machine concurrent, but the constraints they need to satisfy are rather technical and for this reason we prefer to postpone the related discussion.

Transition Rules, Graphically: The position $\mathbf{p} = (e, s, t)$ is represented graphically by marking the edge e with a bullet $\bullet$, and writing the stacks (s, t). A transition $\mathbf{T} \to \mathbf{U}$ is given by depicting only the positions in which $\mathbf{T}$ and $\mathbf{U}$ differ. It is of course intended that all positions of $\mathbf{T}$ which do not explicitly appear in the picture also belong to $\mathbf{U}$. To save space, in Fig. 6 we annotate the transition arrows with a *direction*; we mean that the rule applies (only) to positions which have that direction. We sometimes explicitly indicate the direction of a position by directly annotating it with $^\downarrow, ^\uparrow$ or $^\leftrightarrow$. Notice that no transition is defined for stable positions. We observe that tokens *changes direction* only in one of two cases: either when they move from an edge of type A to an edge of type $A^\perp$ (*i.e.*, when crossing a ax or a cut node), or when they cross a Y-node, in the case where the transitions are marked by (*): moving down from the edge A and then up to $?A^\perp$, or vice versa. Whenever a token is on the conclusion of a box, it can move into that box (graphically, the token "crosses" the border of the box) and it is modified as if it were crossing a node. For exponential boxes, in Fig. 6 we depict only the

[3]In Section III-B4 we show that $\mathtt{Dom}_\mathbf{T}$ is actually redundant; we have however decided to give it explicitly, because it makes the definition of the machine simpler.

border of the box. The transitions for the multiplicative nodes ax, cut, $\otimes$, $\parr$ are the standard ones. The rules for *exponential nodes* are mostly standard. There are however two novelties: the introduction of "dereliction tokens", *i.e.*, tokens which start their path on the conclusion of a $?d$ node, and the Y box. We discuss both below.

Some Further Comments: Certain peculiarities of our interactive machines need to be further discussed:

- *Y-boxes*. The recursive behaviour of Y-boxes is captured by the exponential signature in the form $y(\cdot,\cdot)$, which intuitively keeps track of how many times the token has entered a Y-box so far. Let us examine the transitions via the Y door. Each transition from $!A$ (conclusion of Y) or from $?A^{\perp}$ (premiss of Y) to the edge A (premiss of Y) corresponds to a recursive call. The transition from A to $?A^{\perp}$ captures the return from a recursive call; when all calls are unfolded, the token exits the box. The auxiliary doors of a Y-box have the same behaviour as those of !-boxes.
- *Dereliction Tokens*. As we have explained in section II, this is a key feature of our machine. A dereliction token is generated (according to conditions (i) below) on the conclusion of a ?d node, as depicted in Fig. 6. Intuitively, each dereliction token corresponds to a copy of a box.
- *Box Copies.* A token in a stable position is said to be *stable*. Each such token is the remains of a token which started its journey from DER or ONES, and flowed in the graph "looking for a box". It is immediate to check that a stable token can only be located inside a box, more precisely on the premiss of its principal door. This stable token that was once roaming the net therefore witnesses the fact that *an instance* of dereliction or of one "has found its box". Stable tokens play an essential role, as they keep track of box copies. We are going to formalize this immediately below.

Multitoken Rules: The rules (i)–(iii) from Fig. 6 are where the multitoken nature of the SIAM really comes into play. Those rules are subject to certain conditions, which are intimately related to box copies. Given a state $\mathbf{T}$ of $\mathcal{M}_R$, we define $\mathtt{CopyID}_{\mathbf{T}}(S)$ to be $\{\epsilon\}$ if $R = S$ (we are at depth 0). Otherwise, if S is the structure associated to a box node b of R, we define $\mathtt{CopyID}_{\mathbf{T}}(S)$ as the set of all t such that t is the box stack of a stable token at the principal door of b. Intuitively, as we discussed above, the box stack of each such a token *identifies a copy of the box* which contains S. Rules marked as (i)–(iii) only apply if certain conditions are satisfied:

(i) The position (e, ϵ, t) (resp. (e, δ, t)) does not already belong to $\mathtt{Dom}_{\mathbf{T}}$, and $t \in \mathtt{CopyID}_{\mathbf{T}}(S)$, where S is the structure to which e belongs. If both conditions are satisfied, $\mathtt{Current}_{\mathbf{T}}$ and $\mathtt{Dom}_{\mathbf{T}}$ are extended with the position $\mathbf{p}$. This is the only transition changing $\mathtt{Dom}_{\mathbf{T}}$. Intuitively, each t corresponds to a copy of the (box containing the) one (resp. $?d$) node.

(ii) The token moves inside the $\perp$-box only if its box stack t belongs to $\mathtt{CopyID}_{\mathbf{T}}(S)$, where S is the content of the $\perp$-box. (Notice that if the $\perp$-box is inside an exponential box, there could be several stable tokens at its principal door, one for each copy of the box.)

(iii) Tokens cross a sync node l only if for a certain t, there is a token on each position (e, s, t) where e is a premiss of l, and s indicates an occurrence of atom in the type of e. In this case, all tokens cross the link simultaneously. Intuitively, insisting on having the same stack t means that the tokens all belong to the same box copy. The simultaneous transition of the tokens has to be related to the $s.el$-rule, which takes place only when *all* premisses are conclusions of one nodes. Note that the tokens traverse a sync link only downwards, because all edges are positive.

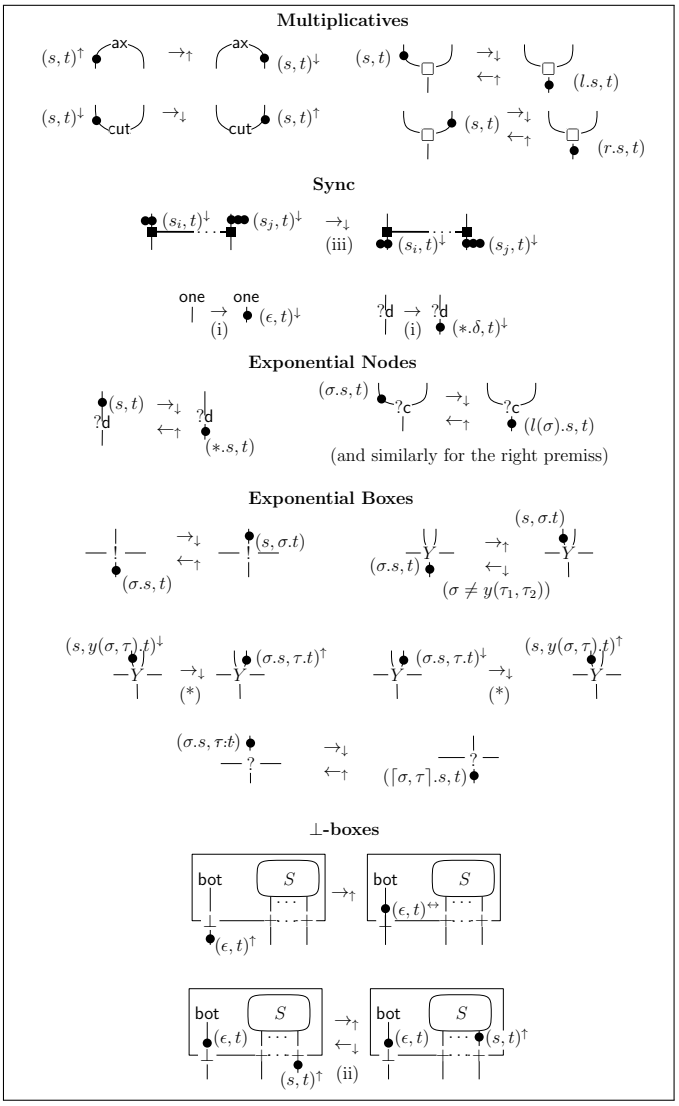


Fig. 6. SIAM Transition Rules.

A *run* of the SIAM machine of R is a *maximal* sequence of transitions $\mathbf{I}_R \to \cdots \to \mathbf{T}_n \to \cdots$ from an initial state $\mathbf{I}_R$.

4) Tracing Back: For each position $\mathbf{p}$ in R, we observe (by examining the cases in Fig. 6) that there is at most one position

from which $\mathbf{p}$ can come via a transition. When disregarding the conditions we impose on rules labelled as (i)–(iii), the transitions also apply to a single token, in isolation. By reading the transitions "backwards", we can therefore define a partial function $\mathrm{orig} : \mathtt{POS}_R \rightharpoonup \mathtt{START}_R$, where $\mathrm{orig}(\mathbf{p}) := \mathbf{s}$ if $\mathbf{p}$ traces back to $\mathbf{s}$. But there is more:

Lemma 6. *For any state* $\mathbf{T}$ *such that* $\mathbf{I}_R \to^* \mathbf{T}$*, the restriction of* orig *to* $\mathtt{Current}_\mathbf{T}$ *is a total, injective function.*

Therefore, for every position $\mathbf{p}$ which appears in a run of $\mathcal{M}_R$, $\mathrm{orig}(\mathbf{p})$ is defined.

With this in mind, $\mathtt{START}_R$ can be seen as an index set identifying each token. For most of this section (until Theorem 14) we are only interested in the "wave" of tokens, and do not need to distinguish them individually. In Section IV, however, we heavily rely on orig to associate values and operations to tokens.

5) Basic Properties: In this and next section, we study some properties of the SIAM. We write $\mathbf{T} \not\to$ if no reduction applies from $\mathbf{T}$. A non final state $\mathbf{T} \not\to$ is called a *deadlock* state. If $\mathbf{I}_R \to \mathbf{T}_1 \to ... \to \mathbf{T}_n \not\to$ is a run of $\mathcal{M}_R$ we say that the run *terminates* (in the state $\mathbf{T}_n$). A run of $\mathcal{M}_R$ *diverges* if it is infinite, *converges* (resp. *deadlocks*) if it terminates in a final (resp. non final) state.

Proposition 7 (Confluence and Uniqueness of Normal Forms)**.** The relation $\to$ enjoys the following properties:

- it is confluent and normal forms are unique;
- if a run of the machine $\mathcal{M}_R$ terminates, then all runs of $\mathcal{M}_R$ terminate.

Proof. By checking each pair of transition rules we observe that $\to$ has the diamond property, because the transitions do not interfere with each other. □

6) State Transformation: Our central tool to relate net rewriting and the SIAM is a mapping of states to states. More precisely, if $R \rightsquigarrow S$, we define a *transformation* as a partial function $\mathrm{trsf}_{R\rightsquigarrow S} : \mathtt{POS}_R \rightharpoonup \mathtt{POS}_S$, which extends to a transformation on states $\mathrm{trsf}_{R\rightsquigarrow S} : \mathcal{S}_R \rightharpoonup \mathcal{S}_S$ in the obvious way, point-wise. We will omit the subscript $R \rightsquigarrow S$ of $\mathrm{trsf}_{R\rightsquigarrow S}$ whenever it is obvious.

Assume $R \rightsquigarrow_a S$ (axiom step), and $\mathbf{p} = (d, s, \epsilon) \in \mathtt{POS}_R$. If $d \in \{e, f, g\}$ as shown in Fig. 7(a), then $\mathrm{trsf}_{R\rightsquigarrow S}(\mathbf{p}) := (h, s, \epsilon) \in \mathtt{POS}_S$. For the other edges, $\mathrm{trsf}_{R\rightsquigarrow S}(\mathbf{p}) := \mathbf{p}$. This definition can rigorously be described as in Fig. 7(b), where the mapping is shown by the dashed arrows. We give some other cases of reductions in Fig. 8. trsf acts as the identity on all positions $\mathbf{p}$ relative to those edges which are not modified by the reduction rule, *i.e.*, $\mathrm{trsf}(\mathbf{p}) = \mathbf{p}$. The cross symbols $\times$ serves to indicate that the source position has no corresponding target in S (remember that the mapping is partial). Intuitively, the token on that position is *deleted* by the mapping. It is important to observe that in the case of steps $bot.el$ and d (the only rules which open a box), a stable token is always deleted. The cases of d and y deserve some further discussion:

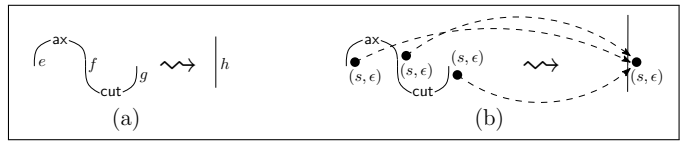


Fig. 7. $\mathrm{trsf}_{R\rightsquigarrow S}$, Formally and as a Drawing.

- In the d rule, the token generated on the $?d$ node is deleted, and disappears in S. For the other tokens, those outside the !-box are modified by removing the signature $*$ (which was acquired while crossing that $?d$ node) from the formula stack. The tokens (e, s, t) inside the !-box are modified by removing the signature $*$ from the *bottom* of the box stack t, which is coherent with the invariant on the size of t (its size is its exponential depth). Why from the bottom of the stack? Because the box b which disappears is at depth 0 in R, therefore for each position (e, s, t) inside the box, the signature corresponding to b is at the bottom of t.
- In the y rule, things are slightly more complicated. What happens to the tokens lying inside a Y-box depends on the bottom element of their box stack, which is the signature corresponding to the Y-box. If the signature at the bottom of the stack is not of the form $y(\cdot, \cdot)$, the token has entered the Y-box only once (*i.e.*, it belongs to the first recursive call) and hence the token is mapped onto a token in the copy of S outside the Y-box. Otherwise, the token is mapped onto a token in the Y-box; it loses one $y(\cdot, \cdot)$ symbol (*i.e.*, it does one iteration less), but the box stack becomes longer (which is coherent with the increase in depth). We show an example in Fig. 9. The (stable) token with a stack $(\delta, *)$ on the premise of Y-node is mapped onto a token on the premise of the ! node, with the same stack. In contrast, the token with a stack $(\delta, y(*, y(*, *)))$ is mapped onto a token on a premise of the Y node on the right-hand side, now with a stack $(\delta, y(*, *).*)$ — it loses a y symbol.

Each statement below can be proved by case analysis.

Lemma 8 (Properties of trsf)**.** *Assume* $R \rightsquigarrow S$.

1. *If* $\mathbf{T} \to \mathbf{U}$ *in* $\mathcal{M}_R$ *then* $\mathrm{trsf}(\mathbf{T}) \to^* \mathrm{trsf}(\mathbf{U})$ *in* $\mathcal{M}_S$.
2. *If* $I_R \to \cdots \to \mathbf{T}_n \cdots$ *is a run of* $\mathcal{M}_R$*, then* $\mathrm{trsf}(I_R) \to^* \cdots \to^* \mathrm{trsf}(\mathbf{T}_n) \cdots$ *is a run of the machine* $\mathcal{M}_S$.
3. $I_R \to \cdots \to \mathbf{T}_n \cdots$ *diverges/converges/deadlocks iff* $\mathrm{trsf}(I_R) \to^* \cdots \to^* \mathrm{trsf}(\mathbf{T}_n) \cdots$ *does.*

We end this section by looking at the number of circulating tokens. We observe that the number of tokens, and stable tokens in particular, in any state $\mathbf{T}$ which is reached in a run of $\mathcal{M}_R$ is finite. We denote by $\mathtt{weight}(\mathbf{T})$ the number of stable tokens in $\mathbf{T}$ (*i.e.*, $\mathtt{Current}_\mathbf{T} \cap \mathtt{STABLE}_R$). The following is immediate by analyzing Fig. 8 and checking which tokens are deleted.

Lemma 9. *Assume* $R \rightsquigarrow S$*. We have that* $\mathtt{weight}(\mathbf{T}) \geq \mathtt{weight}(\mathrm{trsf}(\mathbf{T}))$*. Moreover, if* $R \rightsquigarrow S$ *via the* d*-rule or* $bot.el$*-rule, then* $\mathtt{weight}(\mathbf{T}) > \mathtt{weight}(\mathrm{trsf}(\mathbf{T}))$.

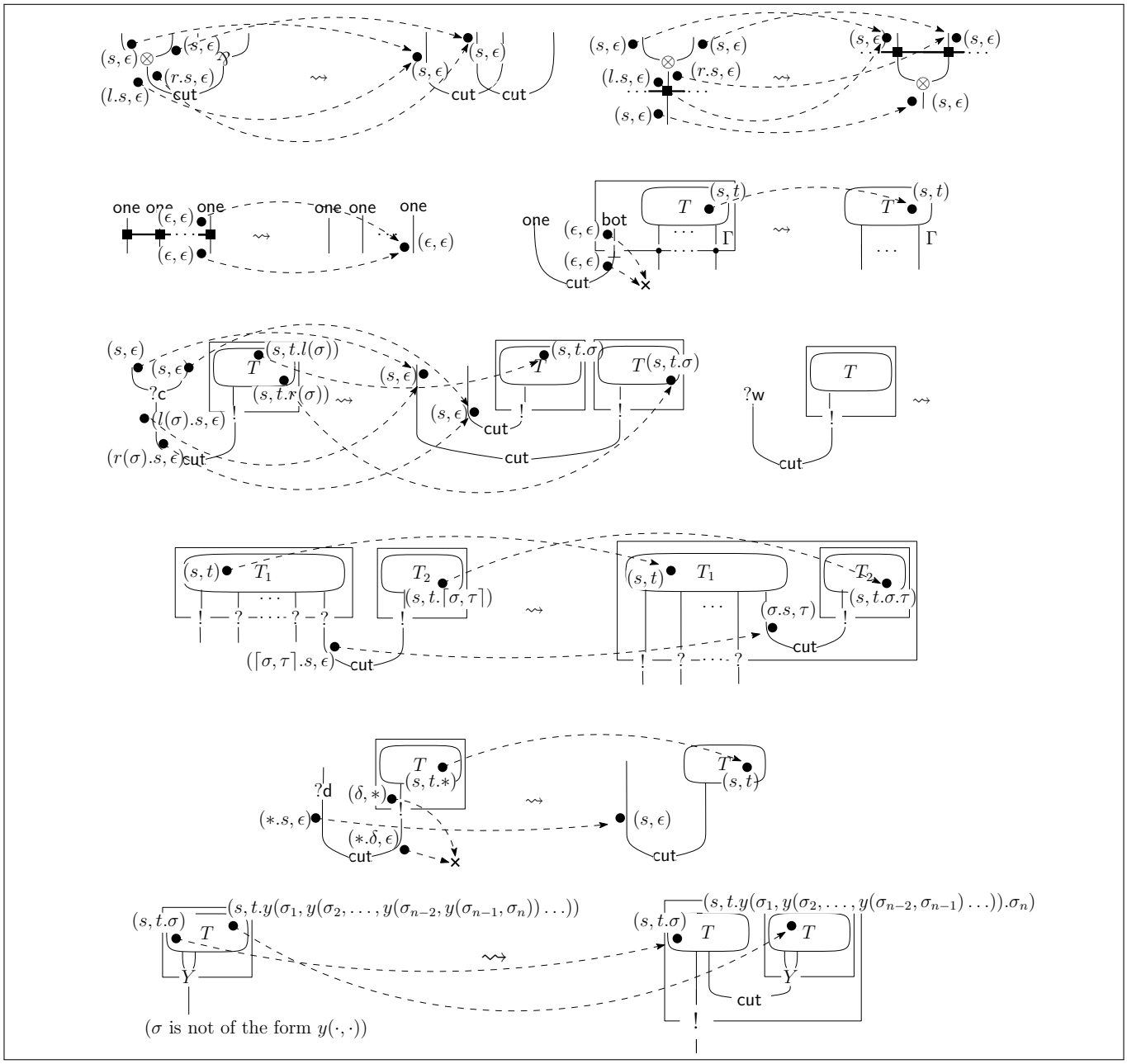


Fig. 8. The Function $\mathrm{trsf}_{R\rightsquigarrow S}$.

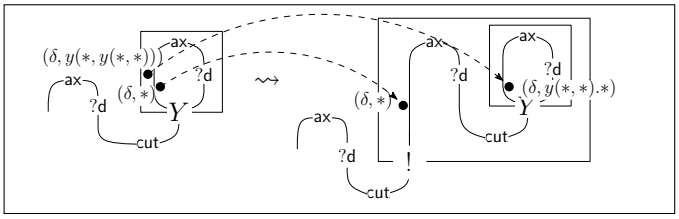


Fig. 9. $\mathrm{trsf}_{R\rightsquigarrow S}$ on y-reduction.

C. The Interplay of Nets and Machines

We already know that if a net R reduces to a normal form S, then S is an MLL net (Corollary 4), actually a very simple one. It is immediate that in this case, every run of the machine $\mathcal{M}_S$ terminates in a final state: each token in the initial state flows to a final position (the net has neither sync nor boxes to stop them). Given an arbitrary net R, we of course do not know if it reduces to a normal form, but we are still able to use the facts above to prove that $\mathcal{M}_R$ is deadlock-free.

Lemma 10 (Mutual Termination)**.** *Let R be a positive net, as in Theorem 3. We have:*

1. *if a run of $\mathcal{M}_R$ terminates, then each sequence of reductions starting from R terminates;*
2. *if a sequence of reductions starting from R terminates, then each run of $\mathcal{M}_R$ terminates in a* final *state.*

Proof. Let us first consider Point 1. By hypothesis, there is a run of $\mathcal{M}_R$ which terminates in a state $\mathbf{T}$. We define $\mathtt{weight}(R) := \mathtt{weight}(\mathbf{T})$. By Lemma 8, if $R \rightsquigarrow S$, trsf maps the run of $\mathcal{M}_R$ into a run of $\mathcal{M}_S$ which terminates in the state $\mathrm{trsf}(\mathbf{T})$. By Lemma 9, $\mathtt{weight}(\mathrm{trsf}(\mathbf{T})) \leq \mathtt{weight}(\mathbf{T})$, hence $\mathtt{weight}(S) \leq \mathtt{weight}(R)$. Using Lemma 9, we prove that it is not possible to have an infinite sequence of $\rightsquigarrow$ reductions starting from R, because: (i) each rewriting step which opens a box (d, or $bot.el$) strictly decreases $\mathtt{weight}(R)$; (ii) there is only a finite number of rewriting steps which can be performed without opening a box. Let us then consider Point 2. By hypothesis, R reduces to a cut free net S, which has the form described in Corollary 4. On such a net, all runs of $\mathcal{M}_S$ terminate in a final state. If $\mathcal{M}_R$ has a run which is infinite (resp. deadlocks), by Lemma 8 trsf would map it into a run of $\mathcal{M}_S$ which is infinite (resp. deadlocks). □

Lemma 10 entails deadlock-freeness as an immediate consequence:

Theorem 11 (Deadlock-Freeness of the SIAM)**.** Let R be a SMEYLL net such that no ? appears in its conclusions. If a run of $\mathcal{M}_R$ terminates in the state $\mathbf{T}$, then $\mathbf{T}$ is a final state.

It should be noticed that in the statement above there is *no assumption of positivity* (unlike in Lemma 10, or Theorem 3). This holds because we can always "close" the general net R by opportunely cutting it, and so obtain a positive net S. If the machine has a deadlock in S, the deadlock must already be in the original net R. Details are given in the long version [5]. The constraint that the conclusions are required not to contain the ? modality is instead a real limit, which is intrinsic to most presentations of GoI (see, *e.g.*, [12]).

D. Computational Semantics

For the rest of this section, we assume the nets to be positive nets; this is the class of nets to which we are going to give computational meaning in the Section IV. The machine $\mathcal{M}_R$ implicitly gives a semantics to R. By Proposition 7, all runs of $\mathcal{M}_R$ have the same behaviour. We can therefore say that $\mathcal{M}_R$ either *converges* (to a unique final state) or *diverges*. We write $\mathcal{M}_R \Downarrow$ if all runs of the machine converge. We write $R \Downarrow$ if all sequences of reductions starting from R terminate in the (unique) normal form. In the previous section we have established (Lemma 10) that

Corollary 12 (Adequacy)**.** $\mathcal{M}_R \Downarrow$ if and only if $R \Downarrow$.

We also already know that:

Corollary 13 (Invariance)**.** Assume $R \rightsquigarrow S$. $\mathcal{M}_R \Downarrow$ if and only if $\mathcal{M}_S \Downarrow$.

We now introduce an equivalence on the machines which is finer than the one induced by convergence. We associate a partial function $[\![R]\!]$ to each net R through the machine $\mathcal{M}_R$, and show that $[\![R]\!]$ is a sound interpretation. This way we have a finer computational model for SMEYLL, on which we will build in the next sections. The *interpretation* $[\![R]\!]$ of a net R is defined as follows

- if $\mathcal{M}_R$ diverges, $[\![R]\!] := \Omega$,
- if $\mathcal{M}_R$ converges, $[\![R]\!]$ is the partial function $[\![R]\!] : \mathtt{INIT}_R \rightharpoonup \mathtt{FIN}_R$ where $[\![R]\!](\mathbf{s}) := \mathbf{p}$ if $\mathbf{p}$ is a final position in the final state $\mathbf{T}$ of the machine (*i.e.*, $\mathbf{p} \in \mathtt{Current}_{\mathbf{T}} \cap \mathtt{FIN}_R$) and $\mathrm{orig}(\mathbf{p}) = \mathbf{s}$.

Theorem 14 (Soundness)**.** *If* $R \rightsquigarrow S$, $[\![R]\!] = [\![S]\!]$.

IV. Beyond Nets: Interpreting Programs

SMEYLL nets as defined and studied in Section III are purely "logical". In this section we introduce *program nets*, which are a (slight) variation on SMEYLL nets in which external data can be manipulated. This allows us to interpret PCF-like languages. The machine running on these nets will be a very simple extension of the SIAM, of which it inherits all properties.

The intuition behind program nets is as follows. Assume a language with a single base type. The base type is mapped to the formula 1; values of the base type are stored in a *memory*. Elementary operations of the base type are modeled using sync nodes, recursion is modeled by Y-boxes, conditional tests are captured by a generalization of the $\perp$-box. Arrow and product types (and all the usual λ-calculus constructions) are encoded by means of one of the well-known mappings of intuitionistic logic into linear logic [11], [16], [18], depending on the chosen evaluation strategy.

Before introducing program nets and interactive machines for them, let us fix a language which will also be our main application.

A. PCF

The language we shall consider in this section is nothing more than Plotkin's PCF, whose *terms* (M, N, P) and *types* (A, B) are defined as follows:

$$
\begin{array}{lcl}
M, N, P & ::= & x \mid \lambda x.M \mid MM \mid \pi_l(M) \mid \pi_r(M) \mid \\
& & \langle M, M\rangle \mid \overline{n} \mid \mathtt{s}(M) \mid \mathtt{p}(M) \mid \\
& & \mathtt{if}\, P\, \mathtt{then}\, M\, \mathtt{else}\, M \mid \mathtt{letrec}\, f\, x = M\, \mathtt{in}\, M, \\
A, B & ::= & \mathbb{N} \mid A \to A \mid A \times A,
\end{array}
$$

Here, n ranges over the set of non-negative natural numbers. A *typing context* Δ is a (finite) set of typed variables $\{x_1 : A_1, \ldots, x_n : A_n\}$, and *typing judgements* are in the form $\Delta \vdash M : A$. We say that a typing judgement is *valid* if it can be derived from a standard set of typing rules (see [5]). Most term constructs are self-explanatory: we only give a few words on the `letrec` construction. In standard PCF, the fixpoint is represented with a Y-combinator: while this is fine in call-by-name evaluation, it does not behave well in the context of call-by-value reduction. As the `letrec` makes sense in both situations, we use it instead. Moreover, we only want to allow recursive definitions of *functions*. To syntactically enforce this, we consider a `letrec` binding two variables: one for the function to be defined, and one for its argument.

B. Program Nets and Register Machines

First of all, we need the definition of a *memory*:

Definition 15. Let I be a (possibly) infinite set whose elements are called *addresses*. Let $\mathtt{SyncNames}$ be a finite set of names, where to each name we associate a positive number that we call *arity*. Given a set of *values* $\mathbb{X}$, we define Mem as the set $\mathrm{I} \to \mathbb{X}$ of all functions from I to $\mathbb{X}$, equipped with the following operations, where the partial function update is defined on a triple $(l, x, \mathbf{m})$ iff the length of x equals the arity of l:

$$\begin{array}{rcl} \mathrm{test} & : & \mathrm{I} \times \mathrm{Mem} \to \mathrm{Bool} \times \mathrm{Mem}; \\ \mathrm{update} & : & \mathtt{SyncNames} \times (\mathrm{I}^*) \times \mathrm{Mem} \rightharpoonup \mathrm{Mem}; \\ \mathrm{init} & : & \mathrm{I} \times \mathrm{Mem} \to \mathrm{Mem}. \end{array}$$

A *memory*[4] is any element $\mathbf{m}$ of Mem, and we say that $\mathbf{m}$ *has values in* $\mathbb{X}$.

Intuitively, $\mathbf{m}$ represents a set of *registers* which are referenced by the elements of I (the addresses). The operation test is used to query the value of a register, update to update its value, and init to set a register of the memory to a default value. Some comments on the operations on Mem are useful. The reason why we have Mem in the codomain of the operation test, is that we aim at a general model where test might have a non-local effect on the memory, such as in a quantum setting (see e.g. [26]), though its implementation is beyond the scope of this paper. Notice also that the type of update is really a dependent-type.

1) Program Nets: Program nets are obtained as a light and natural extension of SMEYLL nets, as follows:

- $\perp$-boxes are replaced by multi-$\perp$-boxes, which are meant to handle *tests*. A *multi-$\perp$-box* is a $\perp$ node to which we associate *two* structures with the same conclusions Γ, as shown in Fig. 10 and 11 (these figures are fully explained later on). An *extended SMEYLL net* is a SMEYLL net where multi-$\perp$-boxes[5] are used in place of $\perp$-boxes.
- Given an (extended) net R, let $\mathtt{SurfOne}(R)$ be the set of all one nodes at the surface, and $\mathtt{SyncNode}(R)$ be the set of *all* sync-nodes of the extended net R, whether at surface or not. A *decoration* of R with names $\mathtt{SyncNames}$ consists of the following two pieces of data:
  1. An injective partial map $\mathtt{ind}(R) : \mathtt{SurfOne}(R) \rightharpoonup \mathrm{I}$, *i.e.*, one nodes are not necessarily decorated;
  2. A *total* map $\mathtt{mkname}(R) : \mathtt{SyncNode}(R) \to \mathtt{SyncNames}$, where $\mathtt{SyncNames}$ is a finite set of names. This map is simply naming the sync nodes appearing in the extended net R. We assume that given a name of arity k, all the sync nodes decorated with that name have the same arity k, where the arity of a sync node is the total number of 1's in its premisses.

[4]An even more fitting name would be *memory states*, but we do not want to overload too much the term "state".

[5]In some example pictures, it is still convenient to use simple $\perp$-boxes; they can be seen as a short-cut for multi-$\perp$-boxes with the same net in both places.

Definition 16. Given a set Mem as in Definition 15, a *program net* is a pair $\mathbf{R} = (R, \mathbf{m}_R)$, where R is a decorated, extended net and $\mathbf{m}_R \in \mathrm{Mem}$ is a memory.

Rewriting on SMEYLL nets easily extends to program nets as shown in Fig. 10 (where we adopt the convention that the memory associated to the net is $\mathbf{m}_1$ before reduction, and $\mathbf{m}_2$ after reduction). The rules are as follows. Rule $decor$ is a new rewriting rule which associates to a surface node one an address $r \in I$; when doing this, we are *linking* the one node to the memory. Rule $bot.el$ is modified to reflect the use of multi-$\perp$-boxes. As shown in Fig. 10, the reduction depends on the memory, and is determined by the result of the operation test. For the other reduction rules, the underlying net is rewritten exactly as for SMEYLL nets. Concerning the memory, only the rule $s.el$ modifies it, as follows: $\mathbf{m}_2 = \mathrm{update}(l, (r_1, r_2, \ldots r_k), \mathbf{m}_1)$, where k is the arity of l. In all the remaining cases $\mathbf{m}_1 = \mathbf{m}_2$ (i.e. the memory is not changed) What we have introduced so far is a

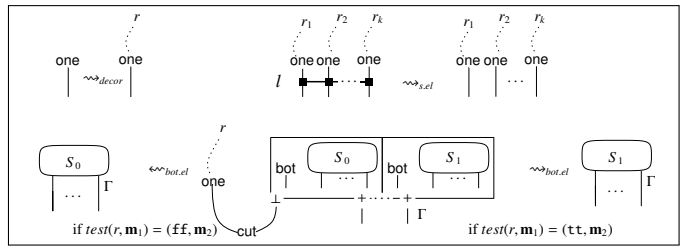


Fig. 10. Program Net Rewriting.

general schema for program nets; in order to capture specific properties, we need to define Mem and the operations on it. In the following section, we specialize the construction to PCF.

2) PCF nets: To encode PCF programs, we use a class of program nets. Once Mem and the operations on it are appropriately defined, we are able to gain more expressive power than in SMEYLL, while good computational properties will be still guaranteed by the underlying nets. A PCF *net* is a program net where Mem has values in $\mathbb{N}$, that is, $\mathrm{Mem} := \mathrm{I} \to \mathbb{N}$. The set of sync-names is $\{\mathtt{max}, \mathtt{p}, \mathtt{s}\}$: $\mathtt{max}$ is binary while $\mathtt{p}$ and $\mathtt{s}$ are unary. The operation update is defined as follows. The sync node of label $\mathtt{p}$ acts as the predecessor, that is $\mathrm{update}(, r, \mathbf{m}_1) = \mathbf{m}_2$ where $\mathbf{m}_2(r) = \mathbf{m}_1(r) - 1$ and $\mathbf{m}_2(k) = \mathbf{m}_1(k)$ if $k \neq r$. The node of label $\mathtt{s}$ acts as the successor, that is $\mathrm{update}(\mathtt{s}, n, \mathbf{m}_1) = \mathbf{m}_2$ where $\mathbf{m}_2(r) = \mathbf{m}_1(r) + 1$ and $\mathbf{m}_2(k) = \mathbf{m}_1(k)$ if $k \neq r$. Finally, the sync node of label $\mathtt{max}$ acts as follows: $\mathrm{update}(\mathtt{max}, r, q, \mathbf{m}_1) = \mathbf{m}_2$ where $\mathbf{m}_2(r) = \mathbf{m}_2(q) = \max(\mathbf{m}_1(r), \mathbf{m}_1(q))$ and $\mathbf{m}_2(k) = \mathbf{m}_1(k)$ if $k \neq r$ and $k \neq q$. For the other operations, $\mathrm{test}(r, \mathbf{m})$ is defined to be $(\mathtt{tt}, \mathbf{m})$ if $\mathbf{m}(r) = 0$, and $(\mathtt{ff}, \mathbf{m})$ otherwise; $\mathrm{init}(r, \mathbf{m})$ is defined to be the memory $\mathbf{n}$ where $\mathbf{n}(r) = 0$ and $\mathbf{n}(k) = \mathbf{m}(k)$ for $k \neq r$. Any typing derivation is encoded as a PCF net. Two possible encodings will be considered: one for call-by-value, one for call-by-name, which correspond to two translations of intuitionistic logic into linear logic [16], [18].

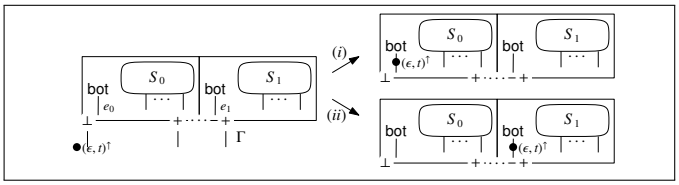


Fig. 11. Multi-$\perp$-box Transition for a Register Machine.

3) Register Machines: The SIAM, as we defined it in Section III-B, is readily adapted to interpret PCF nets. Let us first sketch a general construction for the machine which is associated to a program net. The dynamics of the machine is mostly inherited from the SIAM; the novelty is that the notion of state now includes a memory. Let us fix a set of memories Mem. To a program net $\mathbf{R} = (R, \mathbf{m}_R)$ ($\mathbf{m}_R \in \mathrm{Mem}$) is associated the machine $\mathcal{M}_\mathbf{R}$ whose memories, states and transitions are defined as follows. The definition of position and set of positions is the same as in Section III-B.

Memories: Mem and the operations on it are the same as for the program net $\mathbf{R}$. To illustrate the machine, we need however to make the set of addresses I precise. We take I to be the set of positions $\mathtt{INIT}_R \cup \mathtt{ONES}_R$. We say that the access to the memory is defined for all positions for which $\mathrm{orig}(\mathbf{p}) \in \mathtt{INIT}_R \cup \mathtt{ONES}_R$.

States: A state of $\mathcal{M}_\mathbf{R}$ is a pair $(\mathbf{T}, \mathbf{m_T})$, where $\mathbf{T}$ is a state in the sense of Section III-B, and $\mathbf{m_T} \in \mathrm{Mem}$ is a memory. An *initial* state of $\mathcal{M}_\mathbf{R}$ is a pair $(\mathbf{I}, \mathbf{m_I})$, where $\mathbf{m_I}$ coincides with $\mathbf{m}_R$ for the positions corresponding to decorated one nodes, is arbitrary on $\mathtt{INIT}_R$, and is 0 anywhere else.

Transitions: The transitions are the same as in III-B, except in the following cases, which are defined only if the access to the memory is defined.

- Sync nodes. When the tokens $\mathbf{p}_1, \ldots, \mathbf{p}_k$ cross a sync node with label l and arity k, the operation $\mathrm{update}(l, \mathbf{p}_1, \ldots, \mathbf{p}_k, \mathbf{m})$ opportunely modifies the memory $\mathbf{m}$.
- Multi-$\perp$-box. Let the box be as in Fig. 11, where S_0 and S_1 are the two nets associated to it, and the edges e_0, e_1 are as indicated. When a token is in position $\mathbf{p} = (e, \epsilon, t)$ on the principal conclusion of the box, it moves to (e_0, ϵ, t) if $\mathrm{test}(\mathrm{orig}(\mathbf{p}), \mathbf{m_T})$ returns the boolean `ff` (arrow (i) in Fig. 11) and it moves to (e_1, ϵ, t) if $\mathrm{test}(\mathrm{orig}(\mathbf{p}), \mathbf{m_T})$ returns `tt` (arrow (ii) in Fig. 11). If a token (f, s, t) is on an auxiliary conclusion f, it moves to the corresponding conclusion in S_0 (resp. S_1) if $t \in \mathtt{CopyID}_\mathbf{T}(S_0)$ (resp. $t \in \mathtt{CopyID}_\mathbf{T}(S_1)$).

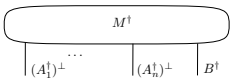
State Transformations: Let $\mathbf{R} = (R, \mathbf{m}_R)$ be a program net and $\mathbf{R} \rightsquigarrow \mathbf{S} = (S, \mathbf{m}_S)$. The transformation trsf described in Fig. 8 associates *positions* of R to *positions* of S; this allows us also to specify the transformation of the memory, hence allowing us to map a memory of $\mathcal{M}_\mathbf{R}$ into a memory of $\mathcal{M}_\mathbf{S}$. More precisely, each state $(\mathbf{T}, \mathbf{m_T})$ of $\mathcal{M}_\mathbf{R}$ is mapped into a state $(\mathrm{trsf}(\mathbf{T}), \mathrm{trsf}(\mathbf{m_T}))$ of $\mathcal{M}_\mathbf{S}$.

4) PCF Machines: A PCF machine is a register machine where Mem and the operations on it are defined as for PCF nets (Section IV-B2). As for the SIAM, we have that trsf maps each run of $\mathcal{M}_\mathbf{R}$ into a run of $\mathcal{M}_{\mathbf{R}'}$ which converges/diverges/deadlocks iff the run on $\mathcal{M}_R$ does. By combining PCF nets and the PCF machine, it is possible to establish similar results to those in Section III-C and III-D. Assume $\mathbf{R}$ is a PCF net of conclusion 1. We write $\mathbf{R} \Downarrow n$ if $\mathbf{R}$ reduces to $\mathbf{S}$, where the value in the memory corresponding to the unique one node in $\mathbf{S}$ is n. Similarly we write $\mathcal{M}_\mathbf{R} \Downarrow n$, where n is the value pointed to by the unique final position in the final state of $\mathcal{M}_\mathbf{R}$.

Theorem 17 (Adequacy)**.** $\mathbf{R} \Downarrow n$ if and only if $\mathcal{M}_\mathbf{R} \Downarrow n$.

C. The Call-by-Value Encoding

In the call-by-value encoding of PCF into PCF nets, the shape of the net corresponding to $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$ is



where $M^\dagger$ is a net and where $(\cdot)^\dagger$ is a mapping of types to SMEYLL formulas, defined as follows:

$$\mathbb{N}^\dagger := \mathbf{1}; \qquad (A \to B)^\dagger := !(A^{\dagger\perp} \parr B^\dagger); \qquad (A \times B)^\dagger := A^\dagger \otimes B^\dagger.$$

In our translation, we have chosen to adopt an *efficient* encoding, rather than the usual call-by-value encoding. In other words, we follow Girard's optimized translation of intuitionistic into linear logic, which relies on properties of positive formulas [11][6]. We feel that this encoding is closer to call-by-value computation than the non-efficient one; it however raises a small issue. Notice in fact that we map natural numbers into the type 1, not !1. How about duplication and erasure, then? We will handle this in the next section, by using sync nodes, but let us first better clarify what the issue is.

Girard's translation relies on the fact that 1 and !1 are logically equivalent (*i.e.*, they are equivalent for provability). However, this in itself is not enough to capture duplication in our setting, because we need to also duplicate the values in the memory, and not only the underlying net. We illustrate this in Fig. 12. The portion inside the dashed line corresponds to a proof of $1 \vdash !1$; when we look at an example of its use (l.h.s. of the figure), we see that by using it we do duplicate the node one, but *not* the value n which is associated to it. The value n is not transmitted from the 1 to the !1 which is going to be duplicated. The logical encoding however still correctly models weakening (r.h.s. of Fig. 12).

Exponential Rules and the Units: The formula $\perp$ does not support contraction, weakening and promotion "out of the box" in SMEYLL but it is nonetheless possible to encode them as PCF nets with the help of the binary sync node `max`.

- *Contraction.* We encode contraction on $\perp$ by using a sync node `max` and the syntactic sugar `copy` defined in Fig. 13.

[6]A good summary of the different translations is given at the address http://llwiki.ens-lyon.fr/mediawiki/index.php/Translations_of_intuitionistic_logic

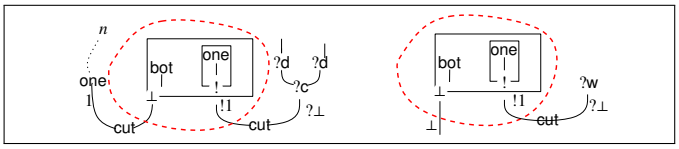


Fig. 12. A Proof of $1 \vdash !1$.

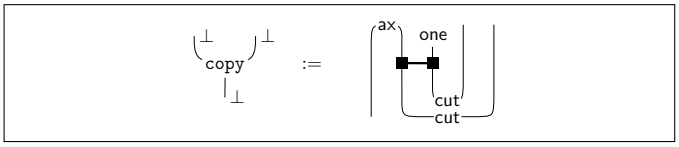


Fig. 13. Syntactic Sugar: Copying $\perp$.

It duplicates the value associated to the incoming edge, and it does so in a call-by-value manner: it will only copy a one node (i.e. a result), not a whole computation. In particular, it should be noted that the rules of net rewriting are not modified.

- *Promotion.* We aim at the reduction(s) shown in Fig. 14: a one node with memory set to n is sent to a frozen computation (inside a !-box) computing the same one node. Since SMEYLL features recursion in the form of the Y-box, together with the copy operation already defined it is possible to write a net for the formula $\perp \parr !1$, as shown in Fig. 15.
- *Weakening.* We can directly use the coding given on the r.h.s. of Fig. 12.

Exponential Rules for the Image $A^\dagger$ of any Type A: The goal of this paragraph is to construct nets which behave like the nodes $?c$, $?w$ and $?p$ of linear logic, this for any edge of type $A^{\dagger\perp}$. For any type A, the formula $A^\dagger$ is a multi-tensor of 1's and !-ed types. We therefore construct the grey contraction, weakening and promotion nodes inductively on the structure

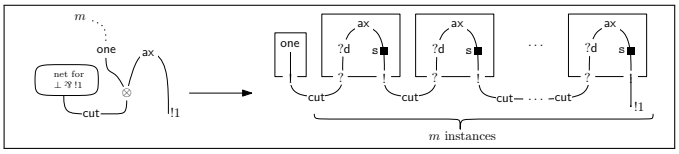


Fig. 14. Desired Behavior for the Mapping of 1 to !1.

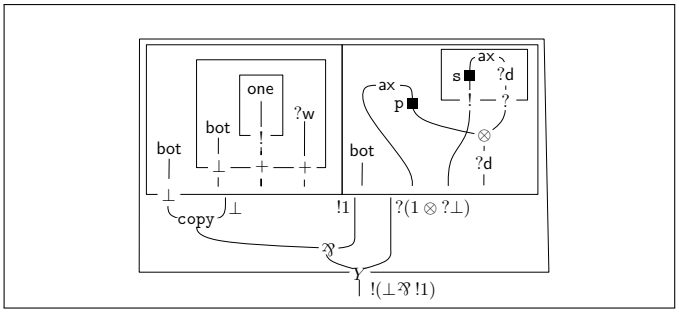


Fig. 15. PCF Net Computing $\perp \parr !1$.

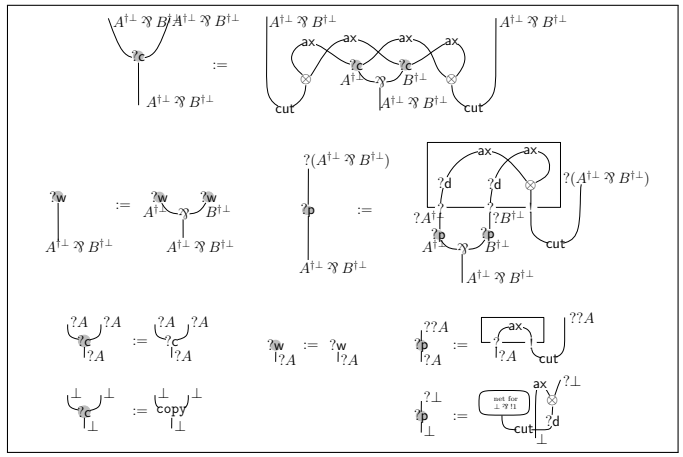


Fig. 16. Inductive Definition of Contraction, Weakening and Promotion Nodes.

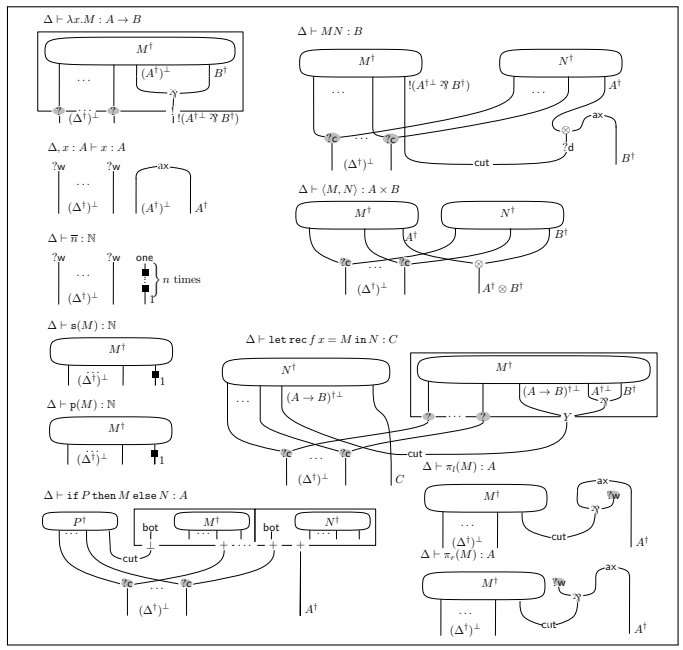
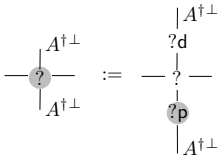


Fig. 17. Call-by-Value Translation of PCF into PCF Nets.

of the type, as presented in Fig. 16.

Interpreting Typing Judgements: Typing derivations are inductively mapped to PCF nets as shown in Fig. 17. The grey nodes ?c and ?w were defined in Fig. 16 (the case of $?\perp$ has been discussed above). The grey node "?" is a shortcut for the following construction:



Adequacy: We prove the following result, which relates the call-by-value encoding into PCF nets and the call-by-value reduction strategy for terms:

Theorem 18. *Let M be a closed term of type $\mathbb{N}$. Then $M \to_{cbv} \overline{n}$ if and only if $M^\dagger \Downarrow n$.*

As a corollary, we conclude that the machine on $M^\dagger$ behaves as M in call-by-value.

Corollary 19. *Let M be a closed term of type $\mathbb{N}$. Then M call-by-value converges if and only if $\mathcal{M}_{M^\dagger}$ itself converges.*

D. The Call-by-Name Encoding

Besides the encoding of call-by-value PCF, which is non-standard, and has thus been described in detail, program nets also have the expressive power to encode call-by-*name* PCF. The encoding is the usual one: a proof net corresponding to $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$ has conclusions $\{?A_1^{*\perp}, \ldots, ?A_n^{*\perp}, B^*\}$, where $(\cdot)^*$ is a mapping of types to SMEYLL formulas:

$$\mathbb{N}^* := 1; \quad (A \to B)^* := ?(A^*)^\perp \parr B^*; \quad (A \times B)^* := !(A^*) \otimes !(B^*).$$

Typing derivations are mapped to PCF nets essentially in the standard way. Then, as in the last section, one can relate the call-by-name encoding in PCF nets and the call-by-name reduction strategy for terms.

Theorem 20 (Adequacy)**.** *Let M be a closed term of type $\mathbb{N}$. Then $M \to_{cbn} \overline{n}$ if and only if $M^* \Downarrow n$.*

As a corollary, one can show that the machine on M^* behaves as M in call-by-name.

Corollary 21. *Let M be a closed term of type $\mathbb{N}$. Then M converges in call-by-name if and only if the register machine $\mathcal{M}_{M^*}$ itself converges.*

V. Conclusions

We have shown how the multitoken paradigm not only works well in the presence of exponential and fixpoints, but also allows us to treat different evaluation strategies in a uniform way. Some other interesting aspects which emerged along the last section are worth being mentioned.

In the call-by-value encoding of PCF, we have used binary *sync nodes* in an essential way, to duplicate values in the register: without them, the efficient encoding of natural numbers would not have been possible. This shows that sync nodes can indeed have an interesting computational role besides reflecting entanglement in quantum computation [4]. In the future, we plan to further the potential of such an use, in particular in view of efficient implementations.

A key feature of SMEYLL nets rewriting is that it is *surface*. Surface reduction allows us to interpret recursion, but how much do we lose by considering surface reduction instead of usual cut-elimination? We think that a simple way to understand the limitations of surface reduction is to consider an analogy to Plotkin's weak reduction. In PCF, $\lambda x.\Omega$ is a normal form. As a consequence one loses, *e.g.*, some nice results about the shape of normal forms in the λ-calculus (which, in logic, corresponds to the subformula property). In presence of fixpoints, however, this is a necessary price to pay. Otherwise, any term including a fixpoint would diverge. Of course there is much more to be said about all this, and we refer the reader to, *e.g.*, the work by Simpson [25].

References

[1] S. Abramsky, R. Jagadeesan, and P. Malacaria. Full abstraction for PCF. *Inf. Comput.*, 163(2):409–470, 2000.

[2] S. Abramsky and G. McCusker. Call-by-value games. In *CSL*, pages 1–17, 1997.

[3] A. Asperti and G. Dore. Yet another correctness criterion for multiplicative linear logic with mix. In *Logical Foundations of Computer Science*, pages 34–46. Springer, 1994.

[4] U. Dal Lago, C. Faggian, I. Hasuo, and A. Yoshimizu. The geometry of synchronization. In *CSL-LICS*, pages 35:1–35:10, 2014.

[5] U. Dal Lago, C. Faggian, B. Valiron, and A. Yoshimizu. Parallelism and synchronization in an infinitary context (long version). Available at http://arxiv.org/abs/1505.03635, 2015.

[6] V. Danos and L. Regnier. The structure of multiplicatives. *Archive for Mathematical logic*, 28(3):181–203, 1989.

[7] V. Danos and L. Regnier. Reversible, irreversible and optimal lambda-machines. *Theor. Comput. Sci.*, 227(1-2):79–97, 1999.

[8] D. R. Ghica. Geometry of synthesis: a structured approach to VLSI design. In *POPL*, pages 363–375, 2007.

[9] D. R. Ghica and A. Smith. Geometry of synthesis II: From games to delay-insensitive circuits. *Electr. Notes Theor. Comput. Sci.*, 265:301–324, 2010.

[10] D. R. Ghica, A. Smith, and S. Singh. Geometry of synthesis iv: compiling affine recursion into static hardware. In M. M. T. Chakravarty, Z. Hu, and O. Danvy, editors, *ICFP*, pages 221–233. ACM, 2011.

[11] J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.

[12] J.-Y. Girard. Geometry of interaction I: Interpretation of system F. *Logic Colloquium 88*, 1989.

[13] N. Hoshino, K. Muroya, and I. Hasuo. Memoryful geometry of interaction: from coalgebraic components to algebraic effects. In *CSL-LICS*, page 52, 2014.

[14] J. M. E. Hyland and C. L. Ong. On full abstraction for PCF: I, II, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[15] O. Laurent. An introduction to proof nets. Available at http://perso.ens-lyon.fr/olivier.laurent/pn.pdf.

[16] I. Mackie. *Applications of the Geometry of Interaction to language implementation*. Phd thesis, University of London, 1994.

[17] I. Mackie. The geometry of interaction machine. In *POPL*, pages 198–208, 1995.

[18] J. Maraist, M. Odersky, D. N. Turner, and P. Wadler. Call-by-name, call-by-value, call-by-need and the linear lambda calculus. *Electr. Notes Theor. Comput. Sci.*, 1:370–392, 1995.

[19] R. Montelatici. Polarized proof nets with cycles and fixpoints semantics. In *TLCA*, pages 256–270, 2003.

[20] M. Pedicini and F. Quaglia. PELCR: parallel environment for optimal lambda-calculus reduction. *ACM Trans. Comput. Log.*, 8(3), 2007.

[21] J. S. Pinto. Parallel implementation models for the lambda-calculus using the geometry of interaction. In *TLCA*, pages 385–399, 2001.

[22] G. D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[23] U. Schöpp. Call-by-value in a basic logic for interaction. In *ESOP*, pages 428–448, 2014.

[24] P. Selinger and B. Valiron. A lambda calculus for quantum computation with classical control. In *TLCA*, pages 354–368, 2005.

[25] A. K. Simpson. Reduction in a linear lambda-calculus with applications to operational semantics. In *RTA 2005*, pages 219–234, 2005.

[26] A. Yoshimizu, I. Hasuo, C. Faggian, and U. Dal Lago. Measurements in proof nets as higher-order quantum circuits. In *ESOP*, pages 371–391, 2014.