Parallelism and Synchronization in an Infinitary Context

Ugo Dal Lago
Università di Bologna & INRIA

Claudia Faggian
CNRS & Univ. Paris Diderot

Benoît Valiron
Univ. Paris Diderot

Akira Yoshimizu
Univ. of Tokyo

February 2, 2015

Abstract

We study multitoken interaction machines in the context of a very expressive 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.

1 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, 13] and the Geometry of Interaction [11] (GoI), themselves tools which were introduced with purely semantic motivations, but which have later been shown to have links to low-level formalisms like asynchronous circuits [9]. For Geometry of Interaction this is especially obvious when it is presented in its more operational form, as a token machine.

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 multiple tokens would not give any advantage. While some forms of synchronization were already present 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 for purely linear ones, whose role is relevant in quantum computation).

Multitoken GoI machines are relatively straightforward to define in a purely 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* thought logically, or *constants* if read through the lenses of functional programming. The reader will find an example in Section 2, Fig. 1.

But can all this scale to a more expressive proof theory or programming formalism? 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 or constants can possibly be erased, thus giving rise to *no* token, or copied, thus creating *more than one* token. The underlying interactive machinery, then, needs to become 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 a *conditional*. The resulting system, called MELLYS, is then strong enough to simulate universal models of functional programming: we prove that PCF can be embedded into MELLYS, 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 2.

It should be observed that the structures we introduce in this paper are considerably richer and more expressive than the ones in [4]. For this reason, it was not possible to directly transfer the proof techniques that we used in [4], and we therefore developed new ones which heavily exploit the interplay between net rewriting and the multitoken machine, and which appear to be of technical interest in their own.

Contributions

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

- *An Expressive Logical System.* We introduce MELLYS 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 MELLYS proofs, as one expects in a system with fixpoints. As main result, however, reduction is proved to be *deadlock free*, *i.e.* normal forms cannot contain surface cuts.
- *A Multitoken Interactive Machine.* MELLYS 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 (synchronization, conditionals). Our proof comes from the interplay of nets and machines: we *transfer termination* from the machine to the nets, and then *transfer back deadlock freeness* from nets to the machine.
- *A Fresh Look at CBV and CBN.* A slight variation on MELLYS nets, together with their abstract machine, is shown to be an adequate model of reduction for Plotkin's PCF. 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 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 different technique. Namely, we *transform* termination of the machine (including termination as a deadlock) into termination of the nets. This combinatorial technique is novel and uses

[1] It is worth noticing that even without fixpoints, to our knowledge there is no direct combinatorial proof of termination for surface reduction.

multiple tokens in an essential way.

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.

The fact that GoI can be turned into an implementation mechanisms for purely functional (but expressive) λ-calculi, has been observed since the beginning of the nineties [5, 16]. 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 [12, 22], 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 [7, 8], 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 [18], even though in a polarized setting. Our Y-box differs from [18] both in the typing and in the dynamics; these differences is what make it possible to build a GoI model.

2 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 results, 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

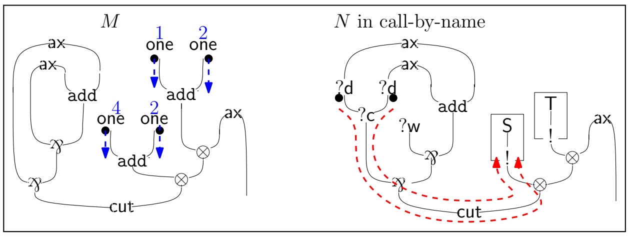


Figure 1: Actual vs. Potential Parallelism

$N = (\lambda x.\lambda y.x + x)(4 - 2)(1 + 2)$. 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 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 performs. 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 (S) 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 illustrate 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 surface level 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 Ω is a divergent term). The call-by-value evaluation of L gives rise to divergence, while in call-by-name L evaluates to 1. Something extremely interesting appears here. We give the call-by-value translation of L in Fig. 2. First of all, we observe that a standard *single-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 that the behaviour of L is not the one which we would expect in call-by-value. Our multitoken machine, instead, launches simultaneously tokens from the derelictions on the surface: the dereliction token of Ω (represented on the right in Fig. 2) reaches the Y-box, and makes the machine diverge.

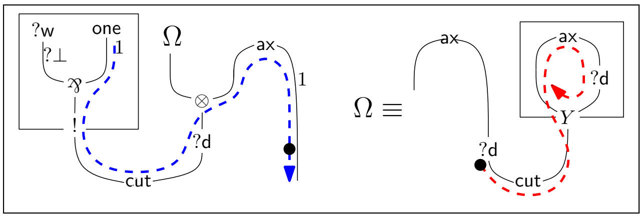


Figure 2: The CBV-translation of $(\lambda x.1)\Omega$.

We end this section by stressing that the interactive machine we use is the same, and that this machine correctly models CBN or CBV, solely depending on the chosen translation of terms into proofs. 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.

3 Nets and a Multitoken Interactive Machine

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

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

Multitoken Machines On MELLYS nets 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 exactly the case in our system. Here, however, we work with surface reduction, and we have fixpoints. For this reason, a different, 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 side, we establish that (ii) *net rewriting has no deadlock*: if no reduction is possible from R, then R is cut-and-sync free. On the machine side, we establish that (iii) if $R \rightsquigarrow R'$ then $\mathcal{M}_R$ converges/diverges/deadlocks iff $\mathcal{M}_{R'}$ does so. 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 have no deadlock.*

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.

3.1 Nets and Their Dynamics.

3.1.1 Formulas

The language of MELLYS *formulas* is identical to the one for MELL, i.e.,

$$A ::= 1 \;\big|\; \bot \;\big|\; X \;\big|\; X^\perp \;\big|\; A \otimes A \;\big|\; A \parr A \;\big|\; !A \;\big|\; ?A,$$

where X ranges over a denumerable set of propositional variables. The constants $1, \bot$ are the *units*. *Atomic formulas* are those formulas which are either propositional variables or units. 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 divided in two classes: positive and negative. In this paper instead, we define as *positive* (denoted by P) and as *negative* (denoted by N) only formulas which are built from units as follows: $P ::= 1 \;\big|\; P \otimes P$, and $N ::= \bot \;\big|\; N \parr N$. So in particular, there are formulas which are neither positive nor negative, such as e.g. $\bot \parr 1$.

3.1.2 Structures

A MELLYS *structure* is a labelled *directed* graph built over the alphabet of nodes which is represented in Fig. 3 (where the *orientation* is the top-bottom one). The edges are labelled with MELLYS formulas; the label of an edge is called its *type*. We call those edges represented below (resp. above) a node symbol *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. Each edge is a conclusion of one node and a premise of at most one node. Edges which are not premiss of any node are the *conclusions of the structure*.

The nodes $!$, Y and $\bot$ are called *boxes*, and have conclusions $\{!A, ?\Gamma\}$, $\{!A, ?\Gamma\}$, and $\{\bot, \Gamma\}$, respectively. The leftmost conclusion is called *principal*, the other ones *auxiliary*. We call the !-boxes and Y-boxes *exponential boxes*. An exponential box is *closed* if it has no auxiliary conclusions. To each box is associated a structure which is called the *content* of the box. To the !-box we associate a structure of conclusions $\{A, ?\Gamma\}$, while 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. The nodes and edges of S are said to be *inside* b. We sometimes call a crossing of the box border a *door*, and speak of premiss and conclusion of a (principal or auxiliary) door.

A node occurs at *depth 0* or *"on the 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. We assume that the maximal depth of the nodes occurring in R is always finite. The sort of each node induces constraints on the number and the labels of its premisses and conclusions, which are shown in Fig. 3. All nodes are standard except sync nodes and Y-boxes. Y-boxes allow us to handle *recursion*, while 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*.

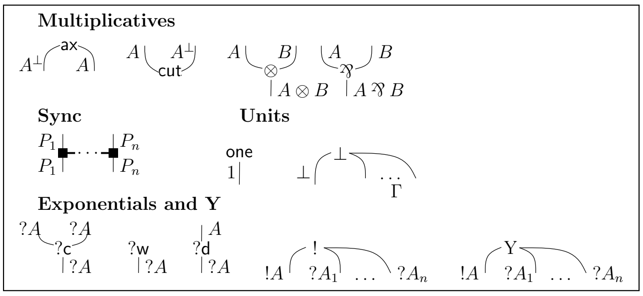


Figure 3: MELLYS Nodes

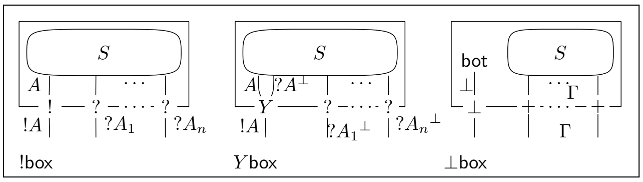


Figure 4: Boxes

W.l.o.g, we assume that *all axioms are atomic*; this hypothesis is not necessary, but it limits the number of cases in the proofs.

3.1.3 Correctness

A *net* is a structure which fulfils a *correctness criterion* defined by means of switching paths (see [14]). A *switching path* on the structure R is an undirected path[2] which

(i) for each $\parr$ and $?c$ node, the path uses at most one of the two *premisses*;

(ii) for each sync node, the path uses at most one of the *conclusions*.

A structure is *correct* if:
1. none of its switching paths is cyclic, and
2. the content of each of its boxes is itself correct.

3.1.4 Net Reduction

Reduction rules for nets are sketched in Fig. 5. The metavariables X, X_1, X_2 range over $\{!, Y\}$ and are used to uniformly specify reduction rules involving exponential boxes (the X's can independently be ! or Y). The reduction rules have two constraints: *any reduction step must be surface*, *i.e.* the step applies

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

only when the cut node (resp. the sync node) is at depth 0, and the *exponential steps are closed*, *i.e.* only take place when the $!A$ premiss of the cut is the principal conclusion of a *closed* box. We write $\rightsquigarrow$ for the rewriting relation induced by these rules.

We observe that the Y-box can be unfolded only when it faces a dereliction node; this is a key difference with [18], which is made possible by our choice of typing, and is essential to our results.

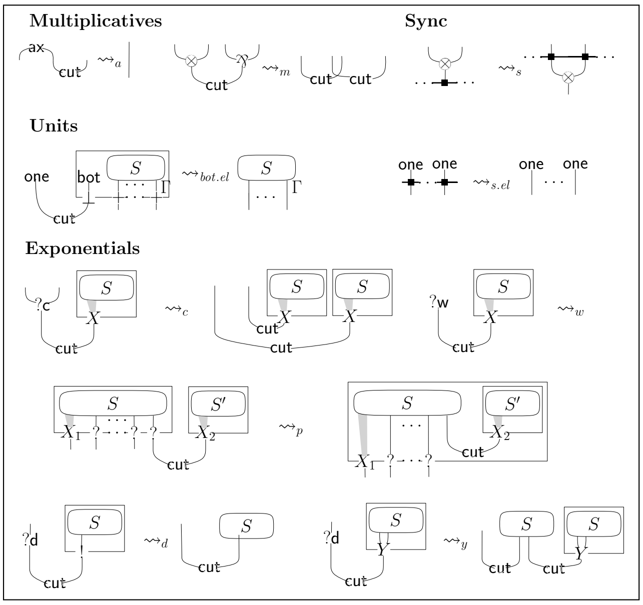


Figure 5: Net Rewriting rules

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

Proposition 1 (Confluence and Uniqueness of Normal Forms)**.** $\rightsquigarrow$ has the following properties:

1. it is confluent and normal forms are unique;
2. any net R 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 $R \rightsquigarrow R_1$ and $R \rightsquigarrow R_2$, then either $R_1 = R_2$ or there exists R_3 such that $R_1 \rightsquigarrow R_3$ and $R_2 \rightsquigarrow R_3$. (1) and (2) are direct consequences. □

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 2 (Deadlock Freeness for Nets)**.** Let R be a MELLYS net such that no $\perp$, ? or ! appears in the conclusions of R. If R contains cuts or sync, a reduction step is always possible.

The rather long proof is given in Appendix A. The key element is the definition of an order on the boxes at depth 0, whose existence relies on the correctness criterion.

Corollary 3 (Cut Elimination)**.** With the same hypothesis as above, if $R \rightsquigarrow^* R'$ and no further reduction is possible, then R' is a cut free MLL net (*i.e.* a net which only uses multiplicative nodes).

3.2 SIAM

All along this section, R indicates a MELLYS structure (with no hypothesis of correctness).

3.2.1 Definition of the SIAM

An *exponential signature* σ and a *stack* s are defined by the following BNF, where ϵ is the empty stack and . denotes concatenation:

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

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:

- $\epsilon[\alpha] = \alpha$,
- $\sigma.\delta[\mu B] = \mu$,
- $\sigma.s'[\mu B] = s'[B]$ whenever $s' \neq \delta$,
- $l.s'[B \bullet C] = s'[B]$ and $r.s'[B \bullet C] = s'[C]$, for $\bullet$ either $\otimes$ or $\parr$.

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

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 in 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, which correspond to s respectively being $*.\delta$, $*.l$ or $*.r.*.\delta$.

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 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. We write $\rightarrow$ for the relation induced by a transition step.

3.2.2 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 $\mathtt{Dom}_\mathbf{T} \subseteq \mathtt{START}_R$[3]. Intuitively, $\mathtt{Current}_\mathbf{T}$ describes the current position of the tokens, and $\mathtt{Dom}_\mathbf{T}$

[3] In section 3.2.4 we show that $\mathtt{Dom}_\mathbf{T}$ is actually redundant, however we have chosen to give it explicitly, because it makes the definition of the machine simpler.

keeps track of which starting positions have been used. 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 the position* $\mathbf{p}$*"* if $\mathbf{p} \in \mathtt{Current}_{\mathbf{T}}$. We use expressions such as "a token moves", "crosses a node", in the intuitive way. We refer to the set of all tokens as *the multitoken*.

3.2.3 Transitions

Fig. 7 describes the transition rules of $\mathcal{M}_R$. The rules marked by (i),(ii),(iii) require some technical conditions, which we prefer to postpone after having developed some intuitions on the machine.

How to read Fig. 7 To represent the position $\mathbf{p} = (e, s, t)$ we mark the edge e with a bullet $\bullet$, and write the stacks (s, t). To represent a transition $\mathbf{T} \to \mathbf{U}$, we depict 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 belong also to $\mathbf{U}$.

To save space, in Fig. 7 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 the token *changes direction* only in two possible cases: either when it moves 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 the token crosses 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 viceversa.

When a token is on the conclusion of a box, it moves into that box (graphically, the token "crosses" the border of the box) and it is modified as if it were crossing a node. We often depict only the border of the box.

The transitions for ax, cut, $\otimes$, $⅋$ are the standard ones. The rules for *exponential nodes* are mostly standard, but there are two important 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 will discuss both.

Intuition behind the definition

- *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 2, 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. 7. Intuitively, each dereliction token corresponds to the copy of a box.

- *Box Copies.* A token in a stable position is said to be *stable*. 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. In Fig. 6 we have explicitly indicated all the exponential transitions which lead to a stable position; the other transition leading to a stable position is the one on $\perp$-box. Each such token is the remains of a token which started its journey from $\mathtt{DER}$ or $\mathtt{ONES}$, and flowed in the graph "looking for a box". 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 and exploit this immediately below.

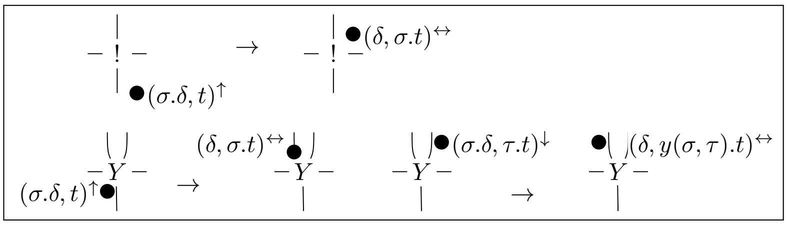

$(\delta, \sigma.t)^{\leftrightarrow}$
$(\sigma.\delta, t)^{\uparrow}$
$(\delta, \sigma.t)^{\leftrightarrow}$
$(\sigma.\delta, \tau.t)^{\downarrow}$
$(\delta, y(\sigma, \tau).t)^{\leftrightarrow}$
Y
$(\sigma.\delta, t)^{\uparrow}$

Figure 6: Exponential transitions to a stable position

Conditions We are now ready to give conditions i–iii for the rules in Fig. 7. The role of such conditions is to cope with the different copies of a same box; we implement this by exploiting the stable tokens.

Given a state $\mathbf{T}$ of $\mathcal{M}_R$, we define $\texttt{CopyID}_{\mathbf{T}}(S) := \{\epsilon\}$ if $R = S$ (we are at depth 0). Otherwise, if S is the structure associated to a box node b, we define $\texttt{CopyID}_{\mathbf{T}}(S)$ as the set of all t such that t is the box stack of a stable tokens 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.

The rules marked by (i),(ii),(iii) only apply under the following conditions:

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

(ii) The token moves inside the $\bot$-box only if its box stack t belongs to $\texttt{CopyID}_{\mathbf{T}}(S)$. (Notice that if the $\bot$-box is inside exponential boxes, in general there are 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 node simultaneously (*synchronization*). Intuitively, t says that the tokens all belong to the same copy of the same box.

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 $\mathbf{I}_R$.

3.2.4 Tracing Back

For each position $\mathbf{p}$ in R, we observe (by examining the cases in Fig. 7) that there is at most one position from which $\mathbf{p}$ can come via a transition. As a matter of fact, if we ignore conditions i–iii, the rules in Fig. 7 define a bi-deterministic automaton which acts on *single positions* (with this reading, all rules are bi-deterministic). By reading the transition rules "backwards" we can *"trace back"* any position (as in [6]). We can therefore define a partial function

$$\text{orig}_R : \texttt{POS}_R \rightharpoonup \texttt{START}_R$$

where $\text{orig}_R(\mathbf{p}) := \mathbf{s}$ if $\mathbf{p}$ traces back to $\mathbf{s}$. Moreover, for all positions which appear in a run of $\mathcal{M}_R$, $\text{orig}_R(\mathbf{p})$ is always defined, as states the following Lemma.

Lemma 4. *For any state $\mathbf{T}$ such that $\mathbf{I}_R \to^* \mathbf{T}$, the restriction of orig_R to $\texttt{Current}_{\mathbf{T}}$ is a total, injective function.*

Proof. By induction on the length n of the sequence of transitions $\mathbf{I}_R \to^* \mathbf{T} = \mathbf{T}_n$. If $n = 0$, then $\mathbf{T} = \mathbf{I}_R$, and orig_R is the identity on $\texttt{Current}_{\mathbf{T}}$. If $n \geq 1$, it is immediate that for all positions $\mathbf{p} \in \texttt{Current}_{\mathbf{T}}$, either $\mathbf{p} \in \texttt{START}_R$ (hence $\text{orig}_R(\mathbf{p}) = \mathbf{p}$) or the token in $\mathbf{p}$ comes from the position $\mathbf{q} \in \mathbf{T_{n-1}}$, and tracing back from $\mathbf{p}$ is the same as tracing back from $\mathbf{q}$.

□

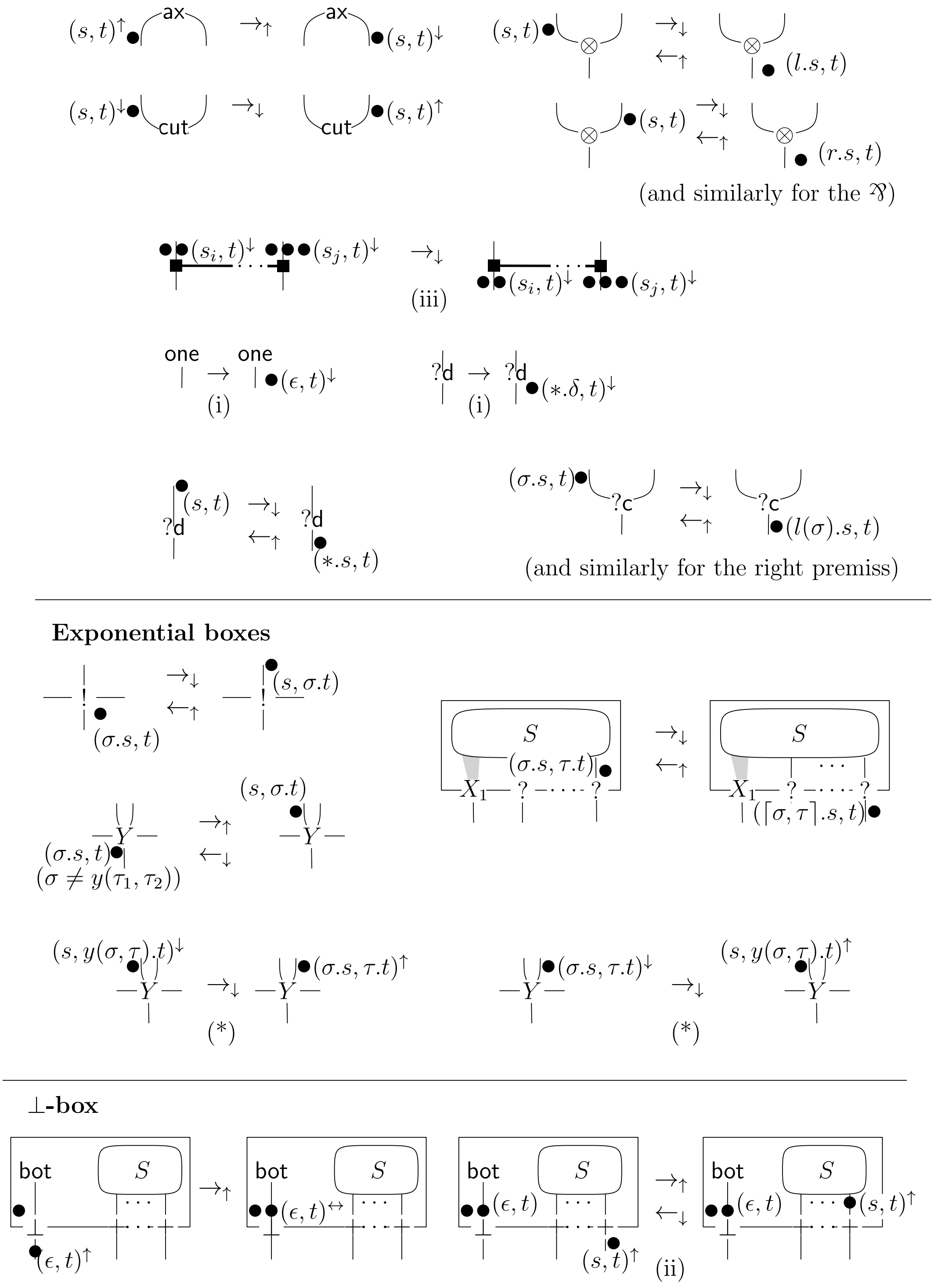

(s, t)↑ ax →↑ ax (s, t)↓
(s, t)↓ cut →↓ cut (s, t)↑
(s, t) ⊗ →↓ ←↑ ⊗ (l.s, t)
⊗ (s, t) →↓ ←↑ ⊗ (r.s, t)
(and similarly for the ⅋)
(s_i, t)↓ (s_j, t)↓ →↓ (iii) (s_i, t)↓ (s_j, t)↓
one → one (ϵ, t)↓
(i)
?d → ?d (∗.δ, t)↓
(i)
(s, t) ?d →↓ ←↑ ?d (∗.s, t)
(σ.s, t) ?c →↓ ←↑ ?c (l(σ).s, t)
(and similarly for the right premiss)
Exponential boxes
! (σ.s, t) →↓ ←↑ ! (s, σ.t)
S (σ.s, τ.t) X1 ? · · · ? →↓ ←↑ S · · · X1 ? · · · ? (⌈σ, τ⌉.s, t)
(s, σ.t)
Y (σ.s, t) (σ ≠ y(τ1, τ2)) →↑ ←↓ Y
(s, y(σ, τ).t)↓ Y →↓ Y (σ.s, τ.t)↑
(*)
Y (σ.s, τ.t)↓ →↓ (s, y(σ, τ).t)↑ Y
(*)
⊥-box
bot S ⊥ (ϵ, t)↑ →↑ bot S (ϵ, t)↔ ⊥
bot S (ϵ, t) ⊥ (s, t)↑ →↑ ←↓ (ii) bot S (ϵ, t) ⊥ (s, t)↑

Figure 7: SIAM transition rules

As a consequence of this Lemma 4, $\mathtt{START}_R$ can be seen as an index set and can be used to identify the individual tokens. However, for most of this section (until Theorem 15) we are only interested in the "wave" of tokens, and do not need to distinguish them individually. In Section 4 we will instead heavily rely on orig_R to associate values and operations to tokens. In the rest of the paper, we will omit the subscript R whenever obvious from the context.

Remark 5. Tracing back from $\mathtt{Current}_{\mathbf{T}}$ allows us to reconstruct $\mathtt{Dom}_{\mathbf{T}}$ from the set of current positions. We have preferred to carry along $\mathtt{Dom}_{\mathbf{T}}$ in the definition of state to make it more immediate, since the definition of orig is rather technical. Similarly, in order not to trace back all the way each time we need the starting position, one can also make the choice to carry the function along with the state. We made a similar choice in our previous work [4], where a state was defined as a function $\mathtt{Dom}_{\mathbf{T}} \to \mathtt{POS}_R$ The two definitions are of course equivalent for all states which can be reached from the initial state, thanks to Lemma 4.

3.2.5 Basic Properties

In this and next section, we study some properties of the SIAM. R indicates a structure, $\mathcal{M}_R$ is its multitoken machine, and $\to$ the relation induced by a transition rule.

We write $\mathbf{T} \not\to$ if no transition applies from the state $\mathbf{T}$. A non final state $\mathbf{T} \not\to$ is called a *deadlock* state. If $\mathbf{I}_R \to \mathbf{T}_1 \to \ldots \to \mathbf{T}_n \not\to$ 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 6 (Confluence and Uniqueness of Normal Form)**.** The relation $\to$ enjoys the following properties:

1. it is confluent and normal forms are unique;

2. 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. (1) and (2) are immediate consequences. □

3.2.6 States Transformation

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

Assume $R \rightsquigarrow_a R'$ (axiom step), and $\mathbf{p} = (e, s, \epsilon) \in \mathtt{POS}_R$. If $e \in \{e_1, e_2, e_3\}$ as shown in Fig. 8(a), then $\mathrm{trsf}_{R\rightsquigarrow R'}(\mathbf{p}) := (e', s, \epsilon) \in \mathtt{POS}_{R'}$. For the other edges, $\mathrm{trsf}_{R\rightsquigarrow R'}(\mathbf{p}) := \mathbf{p}$. This definition can rigorously be described as Fig. 8(b), where the mapping is shown by the dashed arrows. We give the other cases of reductions in Fig. 9 (For the $\otimes/\invamp$ rule, we only give the transformation for the positions above $\otimes$ since the transformation for positions above $\invamp$ are analogous). For the edges which are not modified by the reduction, we identify the corresponding edges in R and R', and have that $\mathrm{trsf}(\mathbf{p}) = \mathbf{p}$. The cross ("×") means that the source position has not corresponding target in R' (remember that the mapping is partial). Intuitively, the token on that position is *deleted* by the mapping. It is important to observe that

Fact 7. If $R \rightsquigarrow R'$ via a $bot.el, d$ or y step, the action of trsf always delete a stable token.

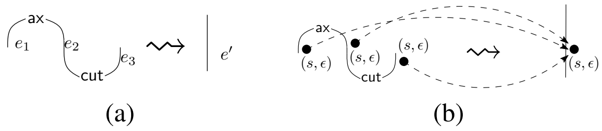


Figure 8: trsf, formally and as a drawing

The cases of d and y reduction deserve some discussion.

- *d reduction*. The token generated from the $?d$ node disappears in R'. For the other tokens, those outside the !-box are modified by deleting from the formula stack the signature $*$ (which was acquired while crossing that $?d$ node), the tokens (e, s, t) inside the !-box are modified by deleting 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 the bottom of the stack t.
- *y reduction*. The token generated from the $?d$ node disappears in R' (similarly to what happens in the d step). More interesting is what happens to the tokens inside the Y-box. This depends on the bottom element of their box stack, which is the signature corresponding to *this* Y-box. If the bottom signature is $*$, the token has entered the Y-box only once (*i.e.* it belongs to the first recursive call) and it is mapped onto the copy of S outside the box. Otherwise, the token remains in the Y-box, but it loses one $y(\_,\_)$ symbol (*i.e.* it does one iteration less).

Each statement below can be proved by case analysis.

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

1. *If* $\mathbf{T} \to \mathbf{U}$ *in* $\mathcal{M}_R$ *then* $\mathrm{trsf}(\mathbf{T}) \to^* \mathrm{trsf}(\mathbf{U})$ *in* $\mathcal{M}_{R'}$.
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* $\mathcal{M}_{R'}$.
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.*

Proof. In Section B, Lemma 35 and 36.

□

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. 9 and, and in particular checking which tokens are deleted (see Fact 7)

Lemma 9. *Assume* $R \rightsquigarrow R'$. *We have that* $\mathtt{weight}(\mathbf{T}) \geq \mathtt{weight}(\mathrm{trsf}(\mathbf{T}))$. *Moreover, if* $R \rightsquigarrow R'$ *via a step* d, y *or bot.el, then* $\mathtt{weight}(\mathbf{T}) > \mathtt{weight}(\mathrm{trsf}(\mathbf{T}))$.

3.3 The Interplay of Nets and Machines

We know that if a net R reduces to a normal form S, then S is an MLL net (Corollary 3). We also know that in this case, every run of the machine $\mathcal{M}_S$ terminates in a final state, because there are no sync nodes and no boxes, and therefore each token in the initial state reaches a final position. 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.

Theorem 10 (Mutual Termination)**.** Let R be as in Theorem 2. 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. (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 $R \rightsquigarrow R'$, trsf maps the run of $\mathcal{M}_R$ into a run of $\mathcal{M}_{R'}$ which terminates in the state $\mathrm{trsf}(\mathbf{T})$. By Lemma 9, $\mathtt{weight}(\mathrm{trsf}(\mathbf{T})) \leq \mathtt{weight}(\mathbf{T})$, hence $\mathtt{weight}(R') \leq \mathtt{weight}(R)$. Using Lemma 9, we prove that is not possible to have an infinite sequence of $\rightsquigarrow$ reductions starting from R, because: (i) each rewriting step which opens a box (d, y, 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 formalize this.

Given a formula A, its *size* $size(A)$ is the number of connectives in the formula. The size $size(l)$ of a node l is 1 if l is not a sync node; if l is a sync node of premisses $P_1, \ldots, P_n$, we define its size as $size(l) := \sum_1^n size(P_i)$.

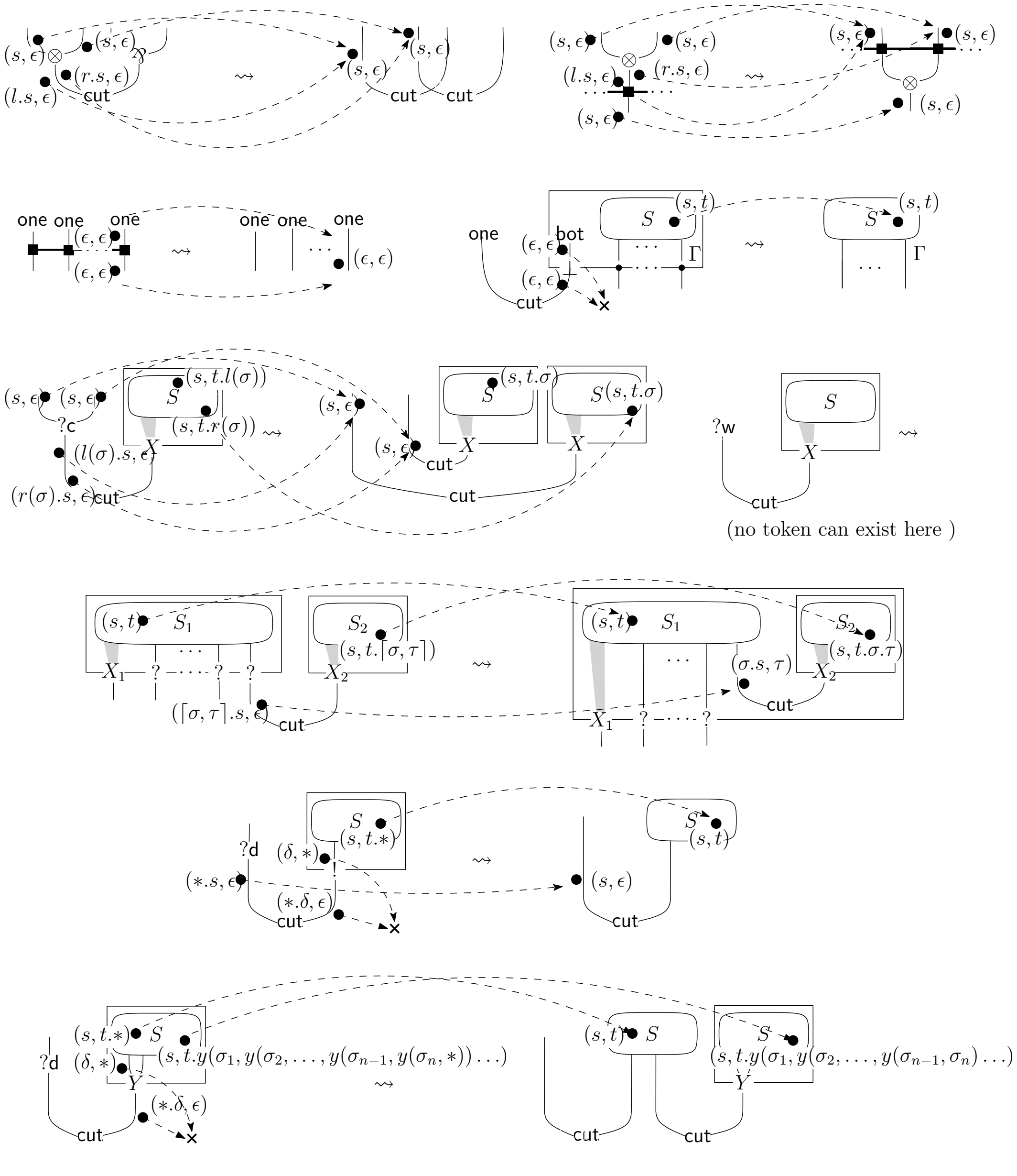


Figure 9: $\mathrm{trsf}_{R \leadsto R'}$

We prove that each sequence of reductions starting from R terminates by induction on the parameter $(\texttt{weight}(R), N_R)$, where N_R is the sum of the size of all the nodes at depth 0. Each step d, y, or $bot.el$ strictly decreases $\texttt{weight}(R)$ (while N_R may increase); each other reduction step strictly decreases N_R (without changing $\texttt{weight}(R)$).

(2.) By hypothesis, R reduces to a cut free MLL net S. 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). □

Theorem 10 entails deadlock freeness. It should be noticed that in the statement below, R has no constraints about $\perp$ or ! appearing in its conclusions, unlike what is needed to guarantee the reduction of nets (Theorem 2).

Theorem 11 (Deadlock Freeness of the SIAM)**.** Let R be a MELLYS 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.

Proof. If R has no $\perp$ and no ! in its conclusions, deadlock freeness is immediate consequence of Theorem 10. However, the result is true also without this constraint, because we can always "close" the net R into a net $\overline{R}$ in a way that cannot create any new deadlocks.

$\overline{R}$ is the net obtained from R when we cut each conclusion A of R with the conclusion $A^\perp$ of the net $S_{A^\perp}$ which is defined as follows. $S_{A^\perp}$ has the direct encoding of the formula tree of $A^\perp$ above the conclusion $A^\perp$ (each modality ? is introduced by a $?d$ node); the atomic leaves are conclusion of an axiom in the case of $X, X^\perp, \perp$, or of a one node in the case of 1. Therefore, $S_{A^\perp}$ has only conclusions $X^\perp, X, 1$, *i.e.* the other side of the axioms.

To conclude we observe that the SIAM deadlocks in $\overline{R}$ iff it deadlocks in R. □

It is worth observing also another direct implication of Theorem 10. We have *an alternative proof technique* for Proposition 1 (ii) and 6 (ii) (neither of which we used so far), since:

Corollary 12. Mutual termination implies that weak and strong normalization are equivalent, this for both $\rightsquigarrow$ and $\rightarrow$.

3.4 Computational Semantics

The machine $\mathcal{M}_R$ implicitly gives a semantics to R. By Proposition 6, all runs of $\mathcal{M}_R$ have the same behaviour. We can therefore say that $\mathcal{M}_R$ *converges* (to the unique final state) or *diverges*. We write $\mathcal{M}_R \Downarrow$ if all runs of the machine converge. Similarly, 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 (Theorem 10) that

Corollary 13 (Adequacy)**.** $\mathcal{M}_R \Downarrow$ if and only if $R \Downarrow$.

We also already proved that:

Corollary 14 (Invariance)**.** Assume $R \rightsquigarrow R'$. $\mathcal{M}_R \Downarrow$ if and only if $\mathcal{M}_{R'} \Downarrow$.

We now introduce an equivalence on the machines which is finer than the one given by convergence. To the machine of a net R we associate a partial function $[\![R]\!]$ and show that the interpretation $[\![R]\!]$ is sound. This way we have a fine computational model for MELLYS, 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 in the state $\mathbf{T}$, $[\![R]\!]$ is the partial function $[\![R]\!] : \texttt{INIT}_R \rightharpoonup \texttt{FIN}_R$ where $[\![R]\!](\mathbf{s}) := \mathbf{p}$ if $\mathbf{p}$ is a final position in $\mathbf{T}$ (*i.e.* $\mathbf{p} \in \texttt{Current}_\mathbf{T} \cap \texttt{FIN}_R$) and $\mathrm{orig}(\mathbf{p}) = \mathbf{s}$.

Theorem 15 (Soundness)**.** *If $R \rightsquigarrow R'$, then $[\![R]\!] = [\![R']\!]$.*

Proof. In Appendix B. □

4 Beyond Nets: Interpreting Programs

MELLYS nets as defined and studied in Section 3 are purely "logical". In this section we introduce *program nets*, which are a (small) variation of MELLYS nets in which external data can be manipulated. This allows us to interpret PCF-like languages in nets. 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 the following. Assume a language with a single base type. The base type is mapped to the formula 1, and values of this type are stored in a *register*. Elementary operations of the base type are modelled using sync nodes, while recursion is modelled by Y-boxes; conditional tests are captured by a generalization of the $\perp$-box. Arrow and product types, and all the regular lambda-calculus constructions are encoded by means of one of the well-known mappings of intuitionistic logic into linear logic [10, 15, 17], depending on the chosen evaluation strategy.

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

4.1 PCF

We consider a PCF language with flat natural numbers, whose *terms* (M, N, P) and *types* (A, B) are defined as follows:

$$\begin{array}{rcl} M, N, P & ::= & x \mid \lambda x.M \mid MN \mid \pi_l(M) \mid \pi_r(M) \mid \langle M, N\rangle \mid \\ & & \overline{n} \mid S(M) \mid P(M) \mid \texttt{if}\, P\, \texttt{then}\, M\, \texttt{else}\, N \mid \\ & & \texttt{let}\,\texttt{rec}\, f\, x = M\, \texttt{in}\, N, \\ A, B & ::= & \mathbb{N} \mid A \to B \mid A \times B, \end{array}$$

where 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 a typing judgement is written as

$$\Delta \vdash M : A$$

A typing judgement is *valid* if it can be derived from the following set of typing rules.

$$\frac{}{\Delta, x : A \vdash x : A} \qquad \frac{\Delta, x : A \vdash M : B}{\Delta \vdash \lambda x.M : A \to B} \qquad \frac{\Delta \vdash M : A \to B \quad \Delta \vdash N : A}{\Delta \vdash MN : B}$$

$$\frac{\Delta \vdash M : A \times B}{\Delta \vdash \pi_l(M) : A} \qquad \frac{\Delta \vdash M : A \times B}{\Delta \vdash \pi_r(M) : B} \qquad \frac{\Delta \vdash M : A \quad \Delta \vdash N : B}{\Delta \vdash \langle M, N\rangle : A \times B}$$

$$\frac{}{\Delta \vdash \overline{n} : \mathbb{N}} \quad \frac{\Delta \vdash M : \mathbb{N}}{\Delta \vdash S(M) : \mathbb{N}} \quad \frac{\Delta \vdash M : \mathbb{N}}{\Delta \vdash P(M) : \mathbb{N}} \quad \frac{\Delta \vdash P : \mathbb{N} \quad \Delta \vdash M : A \quad \Delta \vdash N : A}{\Delta \vdash \texttt{if}\, P\, \texttt{then}\, M\, \texttt{else}\, N : A}$$

$$\frac{\Delta, f : A \to B, x : A \vdash M : B \quad \Delta, f : A \to B \vdash N : C}{\Delta \vdash \texttt{let}\,\texttt{rec}\, f\, x = M\, \texttt{in}\, N : C}$$

Call-by-name and call-by-value evaluation can be defined as usual, giving rise to two rewrite relations over closed, typable terms, called $\to_{cbn}$ and $\to_{cbv}$, respectively.

4.1.1 Call-by-name reduction

A value in the call-by-name setting is defined from the following grammar

$$U, V \quad ::= \quad x \mid \lambda x.M \mid \langle M, N\rangle \mid \overline{n}.$$

In call-by-name, M rewrites to N, written as $M \to_{cbn} N$, is defined according to the following rules.

$$\frac{}{(\lambda x.M)N \to_{cbn} M\{x := N\}} \qquad \frac{}{\pi_l\langle M, N\rangle \to_{cbn} M} \qquad \frac{}{\pi_r\langle M, N\rangle \to_{cbn} N}$$

$$\frac{}{S(\overline{n}) \to_{cbn} \overline{n+1}} \qquad \frac{}{P(\overline{n+1}) \to_{cbn} \overline{n}} \qquad \frac{}{P(\overline{0}) \to_{cbn} \overline{0}}$$

$$\frac{}{\texttt{if}\,\overline{0}\,\texttt{then}\,M\,\texttt{else}\,N \rightarrow_{cbn} M} \qquad \frac{}{\texttt{if}\,\overline{n+1}\,\texttt{then}\,M\,\texttt{else}\,N \rightarrow_{cbn} N}$$

$$\frac{}{\texttt{let}\,\texttt{rec}\,f\,x = M\,\texttt{in}\,N \rightarrow_{cbn} N\{f := \lambda x.\texttt{let}\,\texttt{rec}\,f\,x = M\,\texttt{in}\,f\,x\}}$$

and various congruence rules, as follows. A *call-by-name reduction context* $C[-]$ is defined with the following grammar:

$$[-] \mid C[-]N \mid \pi_l C[-] \mid \pi_r C[-] \mid S(C[-]) \mid P(C[-]) \mid \texttt{if}\,C[-]\,\texttt{then}\,M\,\texttt{else}\,N.$$

Then, provided that $M \rightarrow_{cbn} M'$, we have $C[M] \rightarrow_{cbn} C[M']$.

4.1.2 Call-by-value reduction

A value in the call-by-value setting is defined from the following grammar

$$U, V \quad ::= \quad x \mid \lambda x.M \mid \langle U, V\rangle \mid \overline{n}$$

In call-by-value, M rewrites to N, written as $M \rightarrow_{cbv} N$, is defined according to the following rules.

$$\frac{}{(\lambda x.M)U \rightarrow_{cbv} M\{x := U\}} \qquad \frac{}{\pi_l\langle U, V\rangle \rightarrow_{cbv} U} \qquad \frac{}{\pi_r\langle U, V\rangle \rightarrow_{cbv} V}$$

$$\frac{}{S(\overline{n}) \rightarrow_{cbv} \overline{n+1}} \qquad \frac{}{P(\overline{n+1}) \rightarrow_{cbv} \overline{n}} \qquad \frac{}{P(\overline{0}) \rightarrow_{cbv} \overline{0}}$$

$$\frac{}{\texttt{if}\,\overline{0}\,\texttt{then}\,M\,\texttt{else}\,N \rightarrow_{cbv} M} \qquad \frac{}{\texttt{if}\,\overline{n+1}\,\texttt{then}\,M\,\texttt{else}\,N \rightarrow_{cbv} N}$$

$$\frac{}{\texttt{let}\,\texttt{rec}\,f\,x = M\,\texttt{in}\,N \rightarrow_{cbv} N\{f := \lambda x.\texttt{let}\,\texttt{rec}\,f\,x = M\,\texttt{in}\,f\,x\}}$$

and various congruence rules, as follows. A *call-by-value reduction context* $C[-]$ is defined with the following grammar:

$$\begin{array}{l}[-] \mid C[-]N \mid VC[-] \mid \langle C[-], N\rangle \mid \langle V, C[-]\rangle \mid \pi_l C[-] \mid \pi_r C[-] \mid \\ S(C[-]) \mid P(C[-]) \mid \texttt{if}\,C[-]\,\texttt{then}\,M\,\texttt{else}\,N.\end{array}$$

Then, provided that $M \rightarrow_{cbv} M'$, we have $C[M] \rightarrow_{cbv} C[M']$.

4.2 Program Nets and Register Machines

In the rest of this paper, we assume that all atomic formulas are units $(1, \bot)$. The language of formulas is therefore $A ::= 1 \;\big|\; \bot \;\big|\; A \otimes A \;\big|\; A \parr A \;\big|\; !A \;\big|\; ?A$.

Before going on, we need the definition of registers:

Definition 16. Let I be a (possibly) infinite non empty set whose elements are called *addresses*. Let `Syncnames` be a finite set of names, where to each name we associate a positive number that we call *arity*. A *register* is defined to be a structure *reg* together with the following operations:

$$\begin{array}{rcl} \mathit{test} & : & I \times \mathit{reg} \rightarrow \mathit{Bool} \times \mathit{reg}; \\ \mathit{update} & : & \texttt{Syncnames} \times (I^{\times\,\mathit{arity}}) \times \mathit{reg} \rightarrow \mathit{reg}; \\ \mathit{init} & : & I \times \mathit{reg} \rightarrow \mathit{reg}. \end{array}$$

Intuitively, *test* is used to query the value of an address of the register, *update* to update its value, and *init* to (re)initialize an address of the register.

4.2.1 Program Nets

Program nets are obtained as a slight and natural extension of MELLYS nets, as follows.

- $\perp$-boxes are replaced by multi-$\perp$-boxes, which are meant to handle *tests* on the values which are associated to the principal $\perp$. A *multi-$\perp$-box* is a $\perp$ node to which we associate *two* structures with the same conclusions Γ, as shown in Fig. 10. An *extended MELLYS net* is a MELLYS net where multi-$\perp$-boxes[4] are used in place of $\perp$-boxes.
- Given an (extended) net R, let $\texttt{Surfone}(R)$ be the set of all one nodes at surface level, and $\texttt{Syncnode}(R)$ be the set of *all* sync-nodes of the extended net R, whether at surface level or not. A *decoration* of R with names Syncnames consists of the following three pieces of data:
  1. An injective, partial map $\texttt{ind}(R) : \texttt{Surfone}(R) \rightharpoonup I$, *i.e.* one nodes are not necessarily decorated.
  2. A *total* map $\texttt{mkname}(R) : \texttt{Syncnode}(R) \to \texttt{Syncnames}$, where Syncnames is a finite set of names. This map is simply naming the sync nodes appearing in the extended net R. We assume that all the sync nodes of a given name have the same arity, where the arity of a sync node is the total number of 1's in its premisses.
  3. A function $\texttt{fresh}\colon \mathcal{P}_{fin}(I) \to I$ such that for all finite subset J of I, $\texttt{fresh}(J) \notin J$.

Definition 17. A *program net* is a pair $\mathbf{R} = (R, reg_R)$, where R is a decorated, extended net and reg_R is a register.

The rewriting relation on MELLYS nets easily extends to program nets. In Fig. 10 we illustrate only the novelties. We add a new rewriting rule $decor$ which associates to a surface node one an address $i \in I$; when doing this, we are *linking* the one node to the register. The $bot.el$ rule is modified to handle multi-$\perp$-boxes; it now depends on the register. In the other reduction rules, the underlying net is rewritten exactly as in MELLYS nets. The register can however be modified along reduction; more precisely, it is updated at the $s.el$ step, by performing $update$ according to the operation which is associated to the sync node. In detail:

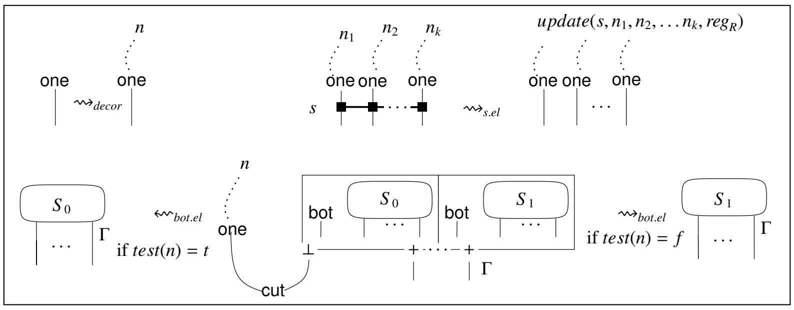


Figure 10: Program Net Rewriting

- The $decor$ rule leaves unchanged the underlying net, but adds an index to an undecorated one-node, with a fresh index created using fresh. The value corresponding to i in the register is initialized by using the operation $init$.
- In all other steps, the decoration is preserved. In the case of the rules $bot.el$, d and y, new one-nodes are possibly created: they are introduced as undecorated nodes.
- The rule $bot.el$ is updated to handle multi-$\perp$-boxes, as shown in Figure 10. The operation $test$ provided by Definition 16 is used to decide which sub-net of the multi-$\perp$-box is used.
- The rule $s.el$ only holds when the corresponding one-nodes are decorated. In that case the decoration is preserved, but the register is updated depending on the name of the sync-node, using the operation $update$.

[4] 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.

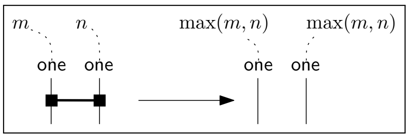


Figure 11: The max-node.

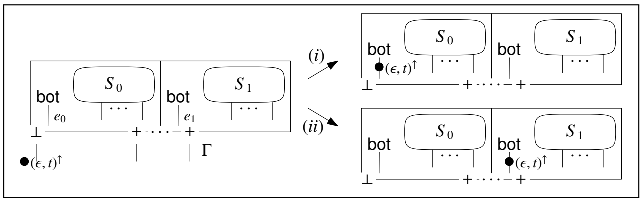


Figure 12: Multi-$\perp$-box transition for a Register Machine

By managing extra information we gain expressive power, while all good computational properties will still be guaranteed by the underlying nets.

4.2.2 PCF nets

In order to encode PCF programs, we use a class of program nets with a special kind of register.

Definition 18. A *PCF net* is a program net where the register is a function $I \to \mathbb{N}$, the set of sync-names is $\{\mathtt{max}, P, S\}$, where $\mathtt{max}$ is binary while P and S are unary, and the operations on the register are as follows:

- $update$ depends on the sync-names. P simply takes the predecessor of the corresponding register, S takes the successor, the sync node of label $\mathtt{max}$ acts as described in Figure 11: it turns the value of the two one-nodes into the maximum of their original value.
- $test$ takes an address i and returns True if the corresponding value in the register is zero and False otherwise.
- $init$ takes an address i and sets to zero the value of the register corresponding to i.

In Sections 4.3 and 4.4 we encode a typing judgement $x_1 : A_1, \dots, x_n : A_n \vdash M : B$ as a PCF net. Two possible encodings will be considered: one for call-by-value, one for call-by-name, which correspond to the two translations of intuitionistic logic into linear logic [15, 17].

4.2.3 Register Machines

The SIAM, as we defined it in Section 3.2, is readily adapted to interpret program nets, hence in particular PCF nets. The novelty is that the notion of state now includes a register. The dynamics of the machine is mostly inherited from the SIAM. To a program net $\mathbf{R} = (R, reg_R)$ is associated the machine $\mathcal{M}_\mathbf{R}$ whose states and transitions are defined as follows.

States The definition of position and set of positions is the same as in Section 3.2. A state of $\mathcal{M}_\mathbf{R}$ is a pair $(\mathbf{T}, \mathtt{reg}_\mathbf{T})$, where $\mathbf{T}$ is a state in the sense of Section 3.2, and $\mathtt{reg}_\mathbf{T}$ is a register whose set of addresses I is the set of positions $\mathtt{INIT}_R \cup \mathtt{ONES}_R$. The access to the register is defined for all positions for which $\mathrm{orig}(\mathbf{p}) = \mathbf{s} \in \mathtt{INIT}_R \cup \mathtt{ONES}_R$; in this case, we say that $\mathbf{p}$ (resp. a token in position $\mathbf{p}$) *points to* $\mathtt{reg}_\mathbf{T}[\mathbf{s}]$, the value of the register associated to $\mathbf{s}$.

Transitions The transitions are the same as in 3.2, except in the following cases, which are of course defined only if the access to the register is defined.

- Multi-$\bot$-box. Let the box be as in Fig. 12, 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 the the test on $\mathtt{reg}_{\mathbf{T}}[\mathrm{orig}(\mathbf{p})]$ gives true (arrow (i) in Fig. 12) and to (e_0, ϵ, t) if the test on $\mathtt{reg}_{\mathbf{T}}[\mathrm{orig}(\mathbf{p})]$ returns false (arrow (ii) in Fig. 12). If a token (e', s, t) is on an auxiliary conclusion e', it moves to the corresponding conclusion in S_0 (resp. S_1) if $t \in \mathtt{CopyID}_{\mathbf{T}}(S_0)$ (resp. $t \in \mathtt{CopyID}_{\mathbf{T}}(S_1)$).
- Sync. When k tokens cross a sync node with label s, the operation s is applied to the elements of the register pointed to by those tokens.

4.2.4 PCF Machines

A PCF machine is a register machine where the register and the operations on it are defined as for PCF nets (Section 4.2.2). 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 3.3 and 3.4.

Lemma 19. *Let $\mathbf{R}$ be a PCF net where all conclusions have type 1. The machine $\mathcal{M}_{\mathbf{R}}$ terminates in a final state (say $(\mathbf{T}, \mathtt{reg}_{\mathbf{T}})$) iff $\mathbf{R}$ reduces to a cut and sync free net (say $\mathbf{S} = (S, reg_S)$). Moreover*

$$reg_S = [\![\mathtt{reg}_{\mathbf{T}}]\!]$$

where $[\![\mathtt{reg}_{\mathbf{T}}]\!]$ is the restriction of $\mathtt{reg}_{\mathbf{T}}$ to the elements pointed to by final positions.

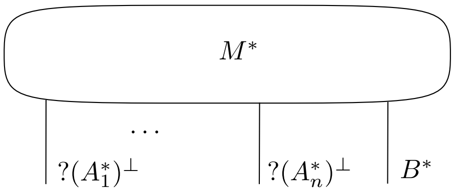
Assume $\mathbf{R}$ is a PCF net of conclusion 1. We write $\mathbf{R} \Downarrow n$ if $\mathbf{R}$ reduces to $\mathbf{S}$, where the value in the register corresponding to the unique one node in S is n. Similarly we write $\mathcal{M}_{\mathbf{R}} \Downarrow n$, where n is the value pointed to by the unique final position in the final state of $\mathcal{M}_{\mathbf{R}}$.

Theorem 20 (Adequacy)**.** $\mathbf{R} \Downarrow n$ if and only if $\mathcal{M}_{\mathbf{R}} \Downarrow n$.

4.3 The Call-by-Name Encoding

Program nets have the expressive power to encode both call-by-value and call-by-name PCF. The call-by-name coding is rather standard; we give more details and the proofs in Appendix C.2.

The proof net corresponding to $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$ has the shape



where R is a net and where $(-)^*$ is a mapping of types to MELLYS formulas, defined as follows:

$$\begin{array}{lcl} \mathbb{N}^* & := & 1 \\ (A \to B)^* & := & ?(A^*)^\perp \parr B^* \\ (A \times B)^* & := & !(A^*) \otimes !(B^*) \end{array}$$

Typing judgements are mapped to PCF nets essentially in the standard way, as shown in Fig. 13.

With this, we can relate the call-by-name encoding in PCF nets and the call-by-name reduction strategy for terms.

Theorem 21 (Adequacy)**.** *Let M be a closed term of type $\mathbb{N}$. Then $M \to_{cbn} \overline{n}$ if and only if $M^* \Downarrow n$.*

Proof. The details of the proof are given in Appendix C.2. □

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

Corollary 22. *Let M be a closed term of type $\mathbb{N}$. Then M converges in call-by-name if and only if the token machine $\mathcal{M}_{M^*}$ itself converges.*

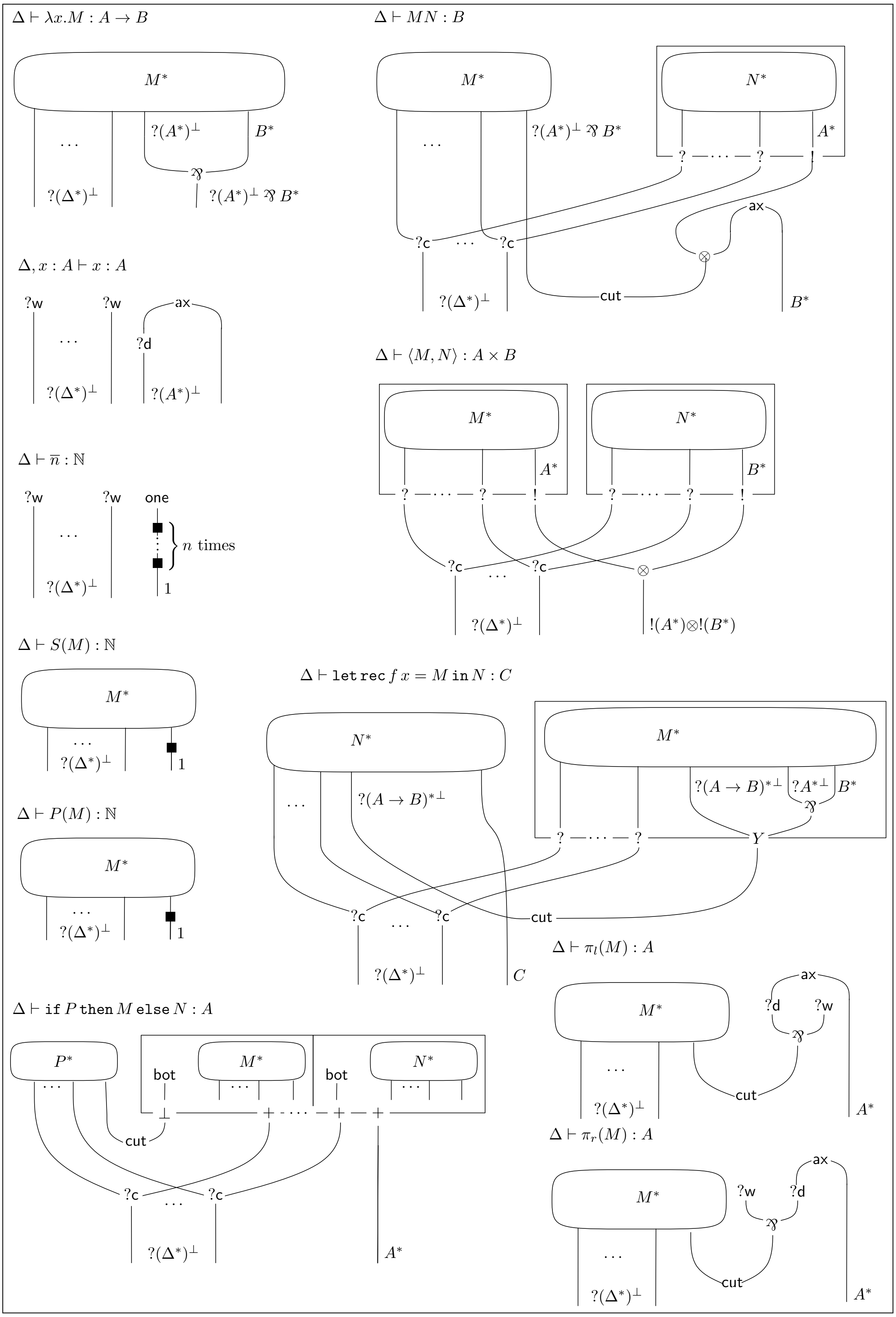


Figure 13: Call-by-name translation of PCF into PCF nets.

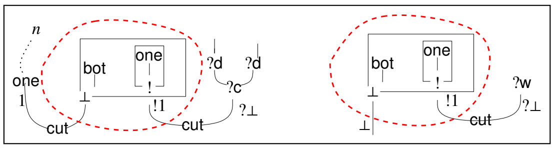


Figure 14: A proof of $1 \vdash !1$

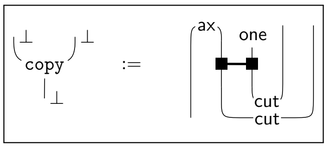
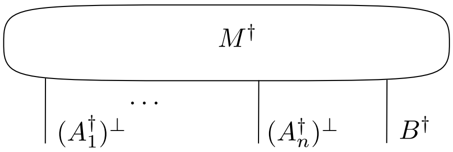


Figure 15: The copy node.

4.4 The Call-by-Value Encoding

Our call-by-value encoding is *not* standard. We devote this section to illustrate it in detail.

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 $(-)^\dagger$ is a mapping of types to MELLYS formulas, defined as follows:

$$\begin{array}{lcl} \mathbb{N}^\dagger & := & 1 \\ (A \to B)^\dagger & := & !(A^{\dagger\perp} \parr B^\dagger) \\ (A \times B)^\dagger & := & A^\dagger \otimes B^\dagger \end{array}$$

In our translation, we have chosen to adopt an *efficient* encoding, rather than the usual call-by-value encoding. Namely we follow Girard's optimized translation of intuitionistic into linear logic, which relies on properties of positive formulas [10][5]. We feel that this encoding is closer to the actual call-by-value computation; it however raises a small issue. Notice in fact that we map natural numbers into the type 1, not $!1$. What about duplication? We 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 guarantee duplication in our setting, because we need to correctly duplicate also the values in the register. We illustrate this in Fig. 14. 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 when reducing this redex, we do duplicate the node one, *but* not the value n which is associated to it. The value n is not transmitted from the conclusion 1 of one to the $!1$ which is going to be duplicated. The logical encoding however still correctly models weakening (r.h.s. of Fig. 14).

Exponential Rules and the Units The formula $\perp$ does not support contraction, weakening and promotion "out of the box" in MELLYS but it is nonetheless possible to encode them in 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 `copy` operation defined in Fig. 15. It indeed 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.

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

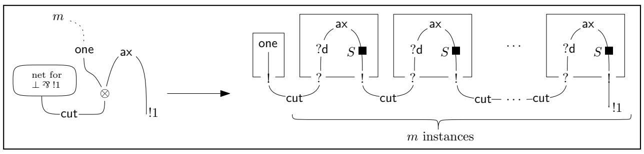


Figure 16: Desired behavior for the mapping of 1 to !1.

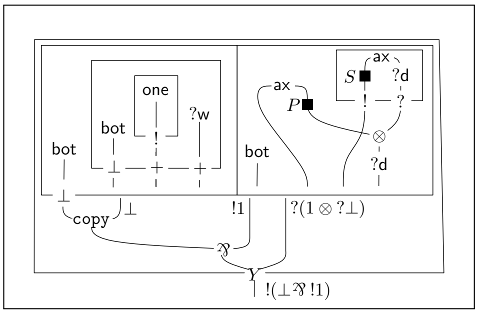
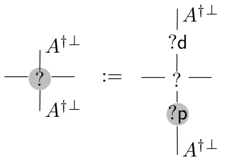


Figure 17: PCF net computing $\bot \parr !1$.

- *Promotion.* We aim at the transformations shown in Fig. 16: a one node with register set to n is sent to a frozen computation (inside a !-box) computing the same one node. Since MELLYS 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 $\bot \parr !1$, as shown in Fig. 17.
- *Weakening.* We can directly use the coding given on the right-hand-side of Fig. 14.

Exponential Rules for the Image $A^\dagger$ of any Type A Our goal is to construct nets which behave like the nodes $?c$, $?w$ and $?p$ of linear logic, this for any edge of type $A^{\dagger\bot}$. 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 of the type, as presented in Fig. 18.

Interpreting Typing Judgements Typing judgements are mapped to PCF nets as shown in Fig. 19. The grey nodes ?c and ?w were defined in Fig. 18 (the case of $?\bot$ has been discussed above). The grey node "?" is a shortcut for the following construction:



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

Theorem 23. *Let M be a closed term of type $\mathbb{N}$. Then $M \to_{cbv} \overline{n}$ if and only if $M^\dagger \Downarrow n$.*

Proof. The proof proceeds in a similar fashion as in the call-by-name setting. □

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

Corollary 24. *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.*

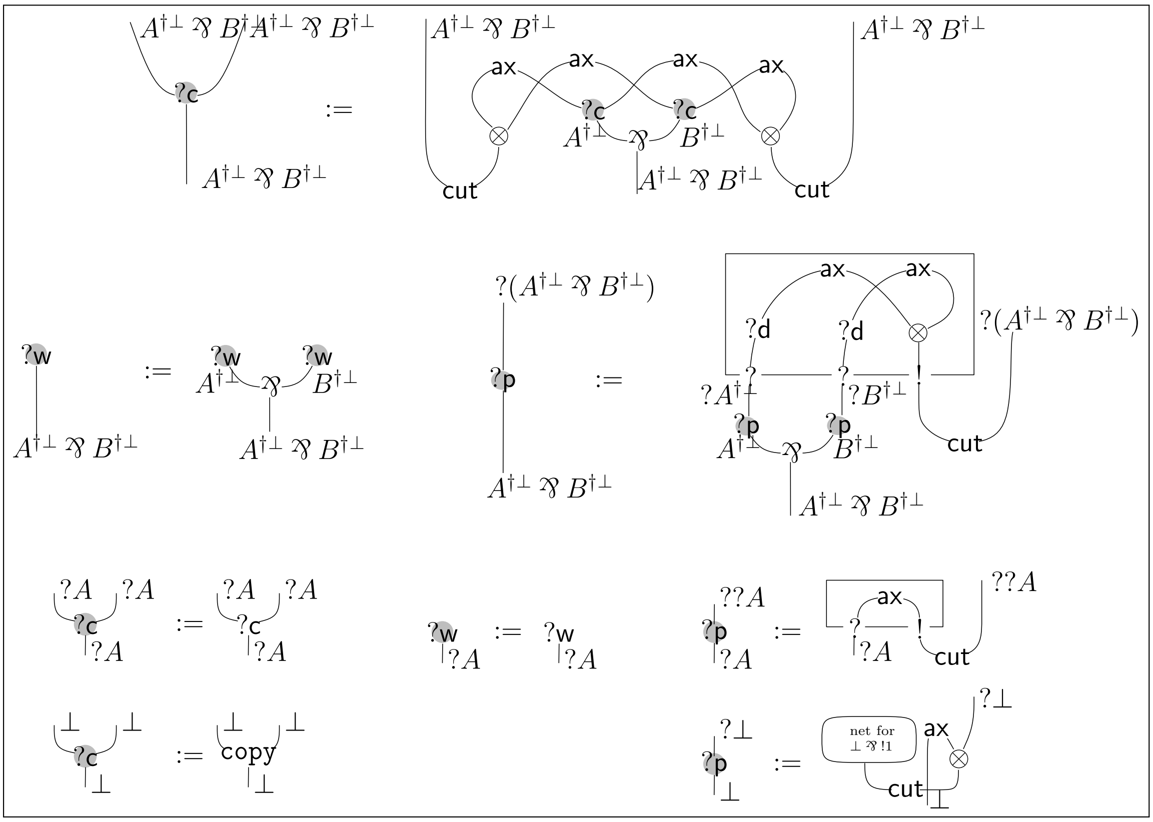


Figure 18: Inductive definition of contraction, weakening and promotion nodes.

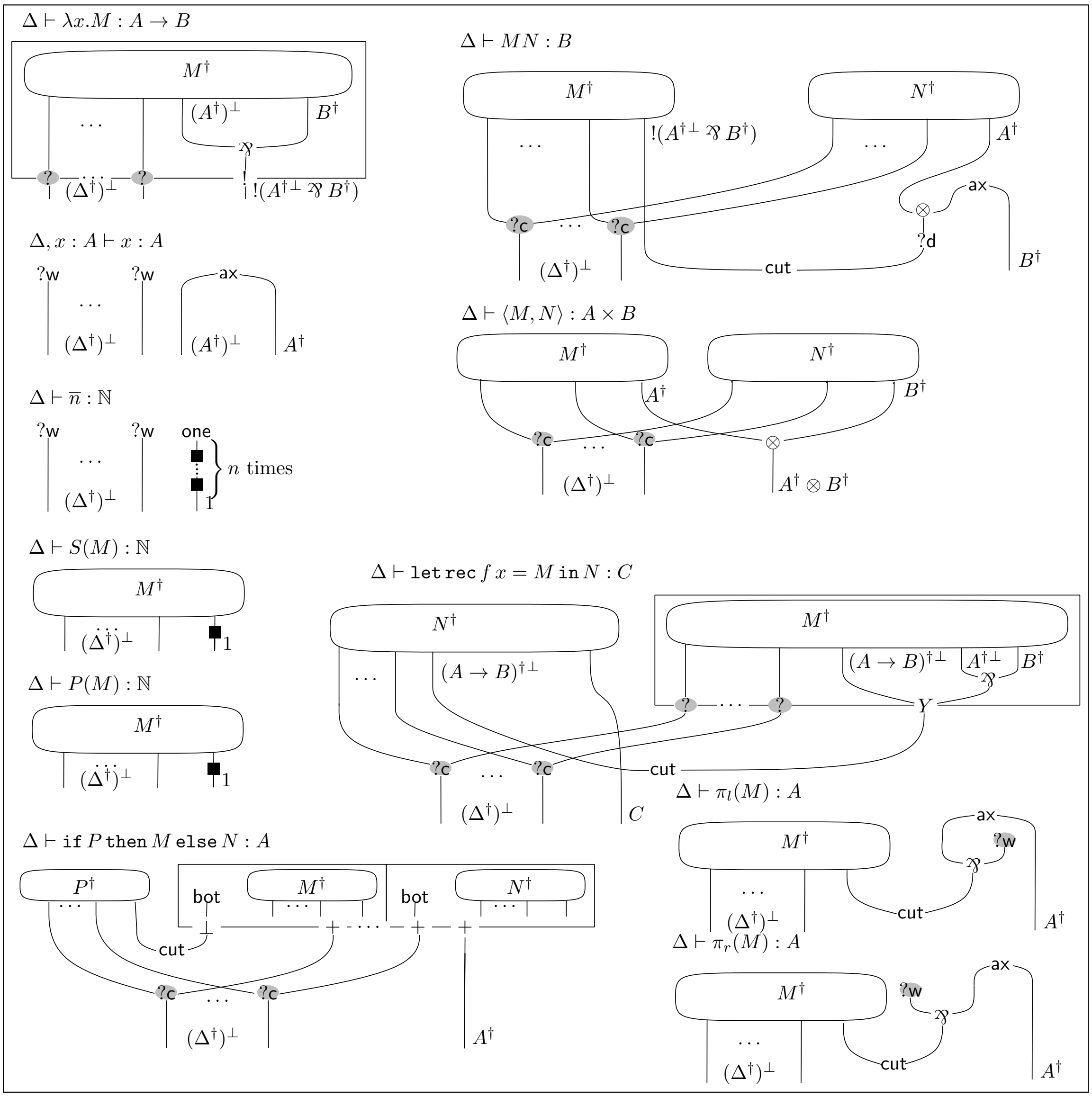

$\Delta \vdash \lambda x.M : A \to B$
$M^\dagger$
$(A^\dagger)^\perp$
$B^\dagger$
$(\Delta^\dagger)^\perp$
$!(A^{\dagger\perp} \parr B^\dagger)$
$\Delta \vdash MN : B$
$N^\dagger$
$!(A^{\dagger\perp} \parr B^\dagger)$
$A^\dagger$
ax
?c
?d
cut
$B^\dagger$
$\Delta, x : A \vdash x : A$
?w
$A^\dagger$
$\Delta \vdash \langle M, N\rangle : A \times B$
$A^\dagger \otimes B^\dagger$
$\Delta \vdash \overline{n} : \mathbb{N}$
one
n times
1
$\Delta \vdash S(M) : \mathbb{N}$
$\Delta \vdash P(M) : \mathbb{N}$
$\Delta \vdash \texttt{let rec} \, f \, x = M \, \texttt{in} \, N : C$
$(A \to B)^{\dagger\perp}$
$A^{\dagger\perp}$
Y
C
$\Delta \vdash \pi_l(M) : A$
$\Delta \vdash \pi_r(M) : A$
$\Delta \vdash \texttt{if} \, P \, \texttt{then} \, M \, \texttt{else} \, N : A$
$P^\dagger$
bot
$\perp$
+

Figure 19: Call-by-value translation of PCF into PCF nets.

4.5 Need for Multiple Tokens: Quantum Computation

In a former paper [4], multitoken machines on SMLL nets were used to encode a *strictly linear* quantum λ-calculus. As it turns out, the additional constructs provided by program nets allow us to capture a more general PCF-style calculus featuring duplicability of classical subterms, and fixpoints [19]. Program nets and register machines specialize to quantum nets and a quantum SIAM by choosing a normalized vector on a finite-dimensional Hilbert space as the underlying register. As set of sync-names we fix a set of *unitary operators*, which act on the register via $update$. The $test$ operation is, simply, *measurement*. We expect to be able to push this till a result of adequacy; however, since both the calculus and the machine are probabilistic, the full development is more involved and we leave it for future work.

5 Conclusions

We have shown how the multitoken paradigm not only works well in the presence of exponential and fixpoints, but that it 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 explore further the potential of such an use, in particular in view of efficient implementations.

A key feature of MELLYS nets rewriting is that it is *surface*. Surface reduction allows us to interpret recursion, as usual, but *we exploit it also in a novel way*: it is in fact the main ingredient which allows us to interpret a quantum lambda calculus with exponentials and recursion, without contradicting the *no cloning axiom*. Indeed, while rewriting a net the only objects being potentially copied are exponential boxes. Since the reduction is surface, and since the one-nodes sitting inside boxes are not initialized, it implies that no *initialized* one-nodes will ever be copied.

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 *Computer Science Logic, Selected Papers*, 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 *Proceedings of the Joint Meeting CSL-LICS 2014*, pages 35:1–35:10, 2014.

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

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

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

[8] 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.

[9] 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.

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

[11] J.-Y. Girard. Geometry of interaction 1: Interpretation of system F. *Logic Colloquium 88*, 1989.

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

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

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

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

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

[17] 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.

[18] R. Montelatici. Polarized proof nets with cycles and fixpoints semantics. In *Proceedings of TLCS,Typed Lambda Calculi and Applications,*, pages 256–270, 2003.

[19] M. Pagani, P. Selinger, and B. Valiron. Applying quantitative semantics to higher-order quantum computing. In *Symposium on Principles of Programming Languages, POPL '14*, pages 647–658, 2014.

[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] U. Schöpp. Call-by-value in a basic logic for interaction. In *Proceedings of Programming Languages and Systems*, pages 428–448, 2014.

A MELLYS: Proof of Theorem 2

In this section we prove Theorem 2. We focus on MELLYS nets, however it is important to observe that *the exact same constructions, results and proofs hold also for extended nets and program nets* as defined in Section 4, because the number of structures associated to a $\bot$-node plays no role at any point; the proof that a reduction is always possible is only concerned with the nodes at the surface.

All along this section, we assume that R is a MELLYS net such that no $\bot$, ? or ! appears in the conclusions. W.l.o.g, we assume that *all axioms are atomic*; this hypothesis is not necessary, but it limits the number of cases in the proofs. We say that an axiom is *polarized* if its conclusions are polarized formulas, *i.e.* the axiom has the form $\{\bot, 1\}$. We say that *a net R is in SM normal form* if no sync reduction and no multiplicative reduction is possible.

The proof of Thoeorem 2 relies on the definition of a strict partial order on the set $\mathcal{T}_R$, which we define as follows.

$\mathcal{T}_R :=$ the set of the following nodes at depth 0 in R: *sync nodes, boxes, and polarized axioms*.

We first observe the following two facts (both are immediate because of the typing of the nodes).

Lemma 25 (Sync Normal Forms)**.** *If no s reduction applies, then the only nodes which can be above a $sync$ node are of type: sync, $\bot$-box, polarized axiom or* one.

Lemma 26. *Each edge of type $!A$ is conclusion of a box (exponential or $\bot$-box).*

The notion of *non bouncing* path is well known in the literature of proof nets, and immediately extends to our case. Given a net R, a directed path is *non bouncing* if for each node on the path the following hold:

- *cuts and axioms*: r enters and exits from different edges;
- *boxes*: if r enters from an auxiliary conclusion, then exits from the principal conclusion, and viceversa;
- $\otimes, \parr$, *$sync$ nodes*: if r enters from a premiss, then exits from a conclusion, and viceversa.

Definition 27 (Priority path)**.** We say that a non bouncing path r is a *priority path* if r starts from a node $a \in \mathcal{T}_R$ *at depth 0* as follows:

- if a is a $sync$ node: r exits a from a premiss;
- if a is a $\bot$-box or a polarized axiom $\{\bot, 1\}$: r exits a from the principal conclusion $\bot$;
- if a is an exponential box : r exits a from an auxiliary conclusion.

We observe that

Fact 28. A priority path has constant depth 0, as it never enters boxes.

We now study which nodes a priority path can reach, and how.

Lemma 29. *Let R be a net in SM normal form, and r a priority path.*

- *When going* downwards*, r can reach only the following nodes, in the following way.*
  - *$\parr, \otimes, ?c, ?d$ nodes: r enters from a premiss, exits from the conclusion.*
- *When going* upwards*, r can reach only the following nodes, in the following way.*
  - *$sync$ nodes: r enters from a conclusion, exits from a premiss;*
  - *$\bot$ nodes: r enters from an auxiliary conclusion (whose type is not $?A$), and exits from the principal conclusion;*

- *$!$ and Y nodes: r enters from the principal conclusion, exits from an auxiliary conclusion;*
- one *nodes: r enters from the conclusion;*
- *r reaches an* axiom *iff it is of the form $\{\bot, 1\}$: r enters from 1, exits from $\bot$;*
- *No $?w$ node can be reached.*

- *Moreover, going downwards, no edge of r has type of the form $!A$, going upwards, no edge of r has type $?A$.*

Proof. We verify the lemma by induction on the length of r. Let us follow r, starting from its origin a, until either r ends (in a conclusion, or in a node one), or r reaches a node $l \in \mathcal{T}_R$. We distinguish two cases.

1. If a is *a box or an axiom, then r starts downwards* with an edge of type F where F is either $\bot$ or $?A$ (for some formula A). While descending, r may traverse $\otimes, \parr, ?c, ?d$ nodes (from a premiss to the conclusion); r cannot traverse any sync node, because F is subformula of the type of each edge below F. We observe that no edge may have type $!B$, because of Lemma 26. Descending, r eventually reaches either a conclusion (in such a case the lemma is proved), or a cut c on which r changes direction. Let C be the premiss of c which contains F, and $C^\bot$ its dual. It is immediate that $C^\bot \neq ?B$ (for any formula B), because otherwise we would have $C = !B^\bot$. Therefore, $C^\bot$ cannot be conclusion of any $?d, ?c, ?w$ node. $C^\bot$ cannot be conclusion of a node ax, $\otimes, \parr$, otherwise it is immediate to see that a multiplicative reduction would apply. As a consequence we have:
   (a) either $C^\bot$ is conclusion of a one node (hence the Lemma is verified);
   (b) or $C^\bot$ is conclusion of a sync node (which r enters from a conclusion);
   (c) or $C^\bot$ is conclusion of a box b. If b is a $\bot$-box, $C^\bot$ must be an auxiliary conclusion (because $C^\bot$ contains $F^\bot$, *i.e.* either 1 or $!A^\bot$); if b is an exponential box, $C^\bot$ must be the principal conclusion (because $C^\bot \neq ?B$).
2. If a is a *sync node, then r starts upwards.* By Lemma 25, any node l above a is either a one node (hence the Lemma is verified), or a node which belongs to $\mathcal{T}_R$: either a sync node (which r enters from a conclusion), or a $\bot$-box, which r enters from an auxiliary conclusion (because the edge is positive), or a polarized axiom.

   Let us indicate by r_1 the prefix of r until the first node $l \in \mathcal{T}_R$, and by r_2 its continuation starting from l. We have verified that r_1 satisfies the property. We observe also that r_2 is again a priority path (of shorter length), and hence it satisfies the property by induction.

□

The following observation is immediate

Lemma 30. *Each priority path is a switching paths.*

Proof. A priority path is a path at constant depth. For each $\parr$ or $?c$ node, r only uses one premiss, because r can only enter from a premiss and exit from the conclusion. The dual is true for sync nodes.

□

We are now able to set our main tool.

Proposition 31 (Priority Order)**.** Let R be a net in SM normal form. The relation $a \prec b$ for a, b in $\mathcal{T}_R$ is defined if there is a priority path r from a to b. This relation defines a strict partial order on $\mathcal{T}_R$, which we call *priority order*.

Proof. We prove that the relation is

1. Irreflexive: $a \prec a$ does *not* hold for any a in $\mathcal{T}_R$.

2. Transitive: $a \prec b$ and $b \prec c$ implies $a \prec c$.

(1). By Lemma 30, $a \prec a$ would imply that there is a cyclic switching path.

(2). Let $r = k_1....k_n$ be the priority path from $a = k_1$ to $b = k_n$ and $r' = n_1...n_m$ be the priority path from $b = n_1$ to $c = n_m$. We claim that b is the only node that the two paths have in common, and we can hence concatenate them and obtain a priority path from a to c. Otherwise, assume that $l = n_j = k_i$ is the first node belonging to r' which belongs also to r. We follow r' from b to l and r from l to b. Let us call this path p, and check that it is non bouncing on l. Therefore p is a priority path, in contradiction with the fact that $b \prec b$ cannot hold.

We observe that p enters l as r' and exits l as r. l cannot be a cut, otherwise l would not be the first node which belongs to both paths. For all the other cases, Lemma 29 guarantees that, if l is a node of type $\otimes, \parr, ?c, ?d$, then r' enters from a premiss, and r exits from a conclusion. The exact opposite is true if l is a sync node. If l is a $\perp$-box r' enters from an auxiliary conclusion, r exits from the principal conclusion. The opposite is true in case l is an exponential box. If l is an axiom, it is polarized; r' enters from a the positive conclusion, while r exits from the negative conclusion.

□

In order to prove Theorem 2 we still need some technical lemmas.

Lemma 32. *If b is an exponential box, and b is maximal for the priority order, then b is a closed box.*

Proof. Each auxiliary conclusion $?A$ needs to be hereditary premiss of a cut. The path r descending from $?A$ to the cut node c is a priority path; the extension of r with the other premiss C of c is still a priority path, which now is ascending . By Lemma 29, the source of C could be either a one, which is not possible because of the type, or a node in $\mathcal{T}_R$, against maximality of b. Therefore, b cannot have any auxiliary conclusion.

□

Lemma 33. *Let R be a net in SM normal form. $\mathcal{T}_R$ is empty iff there are no cuts.*

Proof. Assume $\mathcal{T}_R$ is empty, then R is an MLL net (with one nodes as leaves); if there is a cut, we could perform a multiplicative reduction. Assume $\mathcal{T}_R$ is not empty. If there is a box or a polarized axiom, its principal conclusion needs to be cut, because it does not appear in the conclusions. If there are sync nodes, but no boxes or axioms, we could apply an s reduction.

□

Proof of Theorem 2

Proof. Let R be as in Theorem 2. If R is not in SM normal form, a sync or multiplicative reduction is possible by definition. If R is in SM normal form, and contains cuts, by Lemma 33, $\mathcal{T}_R$ is non empty. We find a valid reduction step by case analysis.

- If $\mathcal{T}_R$ contains a *maximal* node l which is not an exponential box, we focus on it.
  - *l is a sync node.* Any path moving upward from l is a priority path. By using lemma 25 and the fact that l is maximal in $\mathcal{T}_R$, we know that above l there can only be one nodes. An $s.el$ reduction hence applies.
  - *l is a $\perp$-box or a polarized axiom.* Let $\perp$ be the principal conclusion of the box, or the negative conclusion of the axiom. Since it cannot appear in the conclusions, $\perp$ must be hereditary premiss of a cut c. We find c by descending from $\perp$. Since the path descending from l to the node c is a priority path, by Lemma 29 we know that the first node entered by r after the cut can only be conclusion of a one, because any other possibility would belong to $\mathcal{T}_R$ and is excluded by the maximality of l. Hence a reduction applies (either $bot.el$ or $\mathsf{ax/cut}$)
- Otherwise, we choose a node l as follows.

- If $\mathcal{T}_R$ contains only exponential boxes, we observe that all cuts have premisses of type $?A, !A$, and the $!A$ premiss is principal conclusion of an exponential box. Let l be such a box.
- If $\mathcal{T}_R$ contains nodes which are not exponential boxes, let l be any such node.

If l is already a maximal exponential box, let $b_{max} := l$, otherwise we choose a maximal exponential box b_{max} such that $l \prec b_{max}$. The key properties that this careful construction guarantees is that for each exponential box b in the priority path from l to b_{max}:

i. the principal conclusion $!A$ of b is premiss of a cut c;

ii. the other premiss $?A^{\perp}$ of c *is not auxiliary conclusion* of a $\perp$-box.

(i.) is true for l by construction; moreover, for every exponential box b which is reached by a priority path r, r can enter b only from the principal door, ascending from a cut (see case (1.c) in the proof of Lemma 29). (ii.) is true for any cut c which is reached by a priority path r, because if the cut has premisses $!A, ?A^{\perp}$, r can only use the edge $?A^{\perp}$ to *descend* in c, and must do so from a node which cannot be a $\perp$-box (because by Lemma 29, r exits a $\perp$-box only from the $\perp$ conclusion). We are now able to conclude.

By Lemma 32, b_{max} is a closed box. Because of (i.), the principal conclusion $!A$ of b_{max} is premiss of a cut c. The other premiss of c has type $?A^{\perp}$, and because of (ii.), $?A^{\perp}$ can only be conclusion of a node of type $?d, ?c, ?w$, or auxiliary conclusion of an exponential box. In each case a closed reduction applies.

□

B SIAM: Proofs

Fact 34 (Parametricity)**.** Every transition rule is defined parametrically with respect to box stacks. That means, if a transition
$\{(e_1, s_1, t), (e_2, s_2, t), \dots (e_n, s_n, t)\} \cup T \to \{(e'_1, s'_1, t'), (e'_2, s'_2, t'), \dots (e'_m, s'_m, t')\} \cup T$ is possible with box stack t, then the transition
$\{(e_1, s_1, \bar{t}), (e_2, s_2, \bar{t}), \dots (e_n, s_n, \bar{t})\} \cup T \to \{(e'_1, s'_1, \bar{t}'), (e'_2, s'_2, \bar{t}'), \dots (e'_m, s'_m, \bar{t}')\} \cup T$ is also possible with box stack $\bar{t}$.

Lemma 35 (Properties of trsf)**.** *For any reduction $R \rightsquigarrow R'$,*

1. *If* $\mathbf{T} \to \mathbf{U}$ *in* $\mathcal{M}_R$ *then* $\mathrm{trsf}(\mathbf{T}) \to^* \mathrm{trsf}(\mathbf{U})$ *in* $\mathcal{M}_{R'}$.
2. *If* $\mathbf{T}$ *is an initial state in* $\mathcal{M}_R$*, then so is* $\mathrm{trsf}(\mathbf{T})$ *in* $\mathcal{M}_{R'}$.
3. *If* $\mathbf{T} \not\to$ *in* $\mathcal{M}_R$ *then* $\mathrm{trsf}(\mathbf{T}) \not\to$ *in* $\mathcal{M}_{R'}$.
4. *If* $\mathbf{T}$ *is a final state in* $\mathcal{M}_R$*, then so is* $\mathrm{trsf}(\mathbf{T})$ *in* $\mathcal{M}_{R'}$.
5. *If* $\mathbf{T}$ *is a deadlock state in* $\mathcal{M}_R$*, then so is* $\mathrm{trsf}(\mathbf{T})$ *in* $\mathcal{M}_{R'}$.

Proof. Each statement can be proved by case analysis. Note that statement 1. includes $\mathrm{trsf}(\mathbf{T}) = \mathrm{trsf}(\mathbf{U})$.

1. If the transition $\mathbf{T} \to \mathbf{U}$ is not on the redex of $R \rightsquigarrow R'$ then the claim holds, because the positions of tokens and the structure are the same except around the redex. Else we examine each case of reductions, where we have to consider only such a transition $\mathbf{T} \to \mathbf{U}$ that moves a token on the redex:

   - $\underline{\rightsquigarrow_a}$ The states $\mathbf{T}$ and $\mathbf{U}$ are mapped to $\mathrm{trsf}(\mathbf{T}) = \mathrm{trsf}(\mathbf{U})$ by definition of trsf.
   - $\underline{\rightsquigarrow_m}$ Similarly we verify that $\mathrm{trsf}(\mathbf{T}) = \mathrm{trsf}(\mathbf{U})$ (if the transition crosses $\otimes$ or $⅋$ node) or $\mathrm{trsf}(\mathbf{T}) \to \mathrm{trsf}(\mathbf{U})$ (if the transition crosses cut).

- $\underline{\rightsquigarrow_s}$ If the transition crosses the $\otimes$ node then $\mathrm{trsf}(\mathbf{T}) = \mathrm{trsf}(\mathbf{U})$. If the transition crosses the sync node then $\mathrm{trsf}(\mathbf{T}) \to \mathbf{T}'_1 \to \ldots \to \mathbf{T}'_n \to \mathrm{trsf}(\mathbf{U})$, where the first transition crosses the sync node and each of the other transitions crosses the $\otimes$ node one by one.
- $\underline{\rightsquigarrow_{s.el}}$ Similar to the case of $\rightsquigarrow_a$.
- $\underline{\rightsquigarrow_{bot.el}}$ If the transition crosses the cut node or enters the box then $\mathrm{trsf}(\mathbf{T}) = \mathrm{trsf}(\mathbf{U})$. Else, for the transition $\mathbf{T} \to \mathbf{U}$ in $\mathcal{M}_R$ inside the box allowed by a stable token, there is always a transition $\mathrm{trsf}(\mathbf{T}) \to \mathrm{trsf}(\mathbf{U})$ since the structure contained in R in the box is now surface in R'.
- $\underline{\rightsquigarrow_c}$ If the transition crosses the $?c$ node then $\mathrm{trsf}(\mathbf{T}) = \mathrm{trsf}(\mathbf{U})$. Else $\mathrm{trsf}(\mathbf{T}) \to \mathrm{trsf}(\mathbf{U})$ since the transition tules are defined parametrically with respect to box stacks (thus if $\mathbf{T} \to \mathbf{U}$ is done with a box stack $t.l(\sigma)$, $\mathrm{trsf}(\mathbf{T}) \to \mathrm{trsf}(\mathbf{U})$ can be done with box stack $t.\sigma$.)
- For the other exponential rules the situation is similar: if the transition is on a dereliction token the states collapse, else $\mathrm{trsf}(\mathbf{T}) \to \mathrm{trsf}(\mathbf{U})$ is possible by the same rule as $\mathbf{T} \to \mathbf{U}$ with a different box stack.

2. Immediate. Any token in an initial position is mapped to an initial position.

3. First we observe that is impossible for tokens in $\mathbf{T}$ which are outside the redex to become able to move in $\mathrm{trsf}(\mathbf{T})$, since their positions are not modified. Thus we examine only the tokens in the redex. By case analysis, and by Fact 34, the existence of some state $\mathbf{U}$ s.t. $\mathrm{trsf}(\mathbf{T}) \to \mathbf{U}$ contradicts to $\mathbf{T} \not\to$.

4. Immediate. Any token in a final position is mapped to a final position.

5. Immediate consequence of items 3. and 4.

□

Lemma 36.

i. *If* $\mathbf{T}_0 \to \cdots \to \mathbf{T}_n \to \cdots$ *is a run of* $\mathcal{M}_R$*, then* $\mathrm{trsf}(\mathbf{T}_0) \to^* \cdots \to^* \mathrm{trsf}(\mathbf{T}_n) \to \cdots$ *is a run of* $\mathcal{M}_{R'}$*. Moreover*

ii. $\mathbf{T}_0 \to \cdots \to \mathbf{T}_n \to \cdots$ *is infinite/converges/deadlocks iff* $\mathrm{trsf}(\mathbf{T}_0) \to^* \cdots \to^* \mathrm{trsf}(\mathbf{T}_n) \to \cdots$ *does.*

Proof. (i) is direct consequence of Lemma 35. To prove (ii), we first prove (1.) and (2.) below:

1. *If* $\mathbf{T}_0 \to \cdots \to \mathbf{T}_n \to \cdots$ *is an infinite run then* $\mathrm{trsf}(\mathbf{T}_0) \to^* \cdots \to^* \mathrm{trsf}(\mathbf{T}_n) \to \cdots$ *is an infinite run*

   Proof. First, note that in every state $\mathbf{T}$ in a run, the set $\mathtt{Current}_{\mathbf{T}}$ is finite. We call a transition $\mathbf{T} \to \mathbf{U}$ that satisfies $\mathrm{trsf}(\mathbf{T}) = \mathrm{trsf}(\mathbf{U})$ (resp. $\mathrm{trsf}(\mathbf{T}) \to^+ \mathrm{trsf}(\mathbf{U})$) a *collapsing transition* (resp. a *non-collapsing transition*). We show the following:

   - *For any state* $\mathbf{T}$ *s.t.* $\mathbf{T}_0 \to^* \mathbf{T}$*, an infinite sequence of transitions* $\mathbf{T} \to \cdots$ *contains infinitely many non-collapsing transitions.*

     Let $\mathbf{R} \rightsquigarrow_a \mathbf{R}'$, and e_1, e_2, e_3 be the edges in Figure 20. Since $\mathtt{Current}_{\mathbf{T}}$ is finite, the set

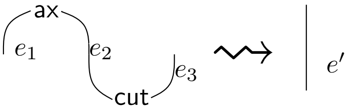


Figure 20: Edges of the redex

$\{(e_i, s, \epsilon) \mid i \in \{1, 2, 3\}\} \cap \mathtt{Current}_{\mathbf{T}}$ is finite. Let n be the number of elements in this set of positions. It is straightforward to check that the length of a sequence of transitions from $\mathbf{T}$ only using collapsing transitions is bounded by $2n$. (We cannot apply more than two collapsing transitions on each token.) Thus an infinite sequence $\mathbf{T} \rightarrow \cdots$ of transitions must contain a non-collapsing transition. By repeating this argument for all states in the run, we see that the infinite sequence $\mathbf{T} \rightarrow \cdots$ contains infinitely many non-collapsing transitions.

We conclude that there are infinitely many transitions $\mathbf{T}_i \rightarrow \mathbf{T}_{i+1}$ s.t. $\mathrm{trsf}(\mathbf{T}_i) \rightarrow^{+} \mathrm{trsf}(\mathbf{T}_{i+1})$, and therefore the run $\mathrm{trsf}(\mathbf{T}_0) \rightarrow^{*} \cdots \rightarrow^{*} \mathrm{trsf}(\mathbf{T}_n) \rightarrow \cdots$ is infinite. A similar argument applies for all other reduction steps.

■

2. *If* $\mathbf{T}_0 \rightarrow \cdots \rightarrow \mathbf{T}_n$ *is a run which terminates then* $\mathrm{trsf}(\mathbf{T}_0) \rightarrow^{*} \cdots \rightarrow^{*} \mathrm{trsf}(\mathbf{T}_n)$ *is a run which terminates. In this case,*

   (a) if $\mathbf{T}_n$ *is final, so is* $\mathrm{trsf}(\mathbf{T}_n)$

   (b) if $\mathbf{T}_n$ *is not final, so is* $\mathrm{trsf}(\mathbf{T}_n)$

   Proof. By using Lemma 35, items 3., 4. and 5. □

We conclude by noticing that (1.) and (2.) together are equivalent to either of the following:

- $\mathbf{T}_0 \rightarrow \cdots \rightarrow \mathbf{T}_n \cdots$ is an infinite run iff $\mathrm{trsf}(\mathbf{T}_0) \rightarrow^{*} \cdots \rightarrow^{*} \mathrm{trsf}(\mathbf{T}_n) \cdot$ is an infinite run
- $\mathbf{T}_0 \rightarrow \cdots \rightarrow \mathbf{T}_n$ is a run which terminates iff $\mathrm{trsf}(\mathbf{T}_0) \rightarrow^{*} \cdots \rightarrow^{*} \mathrm{trsf}(\mathbf{T}_n)$ is a run which

Similarly, (2.a) and (2.b) are equivalent to either of the following:

- $\mathbf{T}_n$ is final iff $\mathrm{trsf}(\mathbf{T}_n)$ is final;
- $\mathbf{T}_n$ is a deadlock iff $\mathrm{trsf}(\mathbf{T}_n)$ is a deadlock.

□

Proof of Theorem 15 (Soundess). Let R be a net of conclusions $A_1, \ldots, A_n$, and $R \rightsquigarrow R'$; we adopt the following convention: we identify each conclusion of a net with the occurrence of formula A_i typing it, so that there is no ambiguity. In particular, R and R' have *the same initial and final positions*. We now show that the interpretation of a net is preserved by all normalization steps.

We first observe the following.

Lemma 37. *Let* $\mathbf{I}_R \rightarrow \ldots \rightarrow \mathbf{T}$ *a run in* $\mathcal{M}_R$ *and* $\mathbf{I}_{R'} \rightarrow \ldots \rightarrow \mathrm{trsf}(\mathbf{T})$ *the corresponding sequence of transitions in* $\mathcal{M}_{R'}$*. For each* $\mathbf{p} \in \mathtt{Current}_{\mathbf{T}}$*, we have the following:*

- $\mathrm{orig}_R(\mathbf{p}) \in \mathtt{INIT}_R$ *iff* $\mathrm{orig}_{R'}(\mathrm{trsf}(\mathbf{p})) \in \mathtt{INIT}_{R'} = \mathtt{INIT}_R$
- *if* $\mathrm{orig}_R(\mathbf{p}) \in \mathtt{INIT}_R$ *then* $\mathrm{orig}_R(\mathbf{p}) = \mathrm{orig}_{R'}(\mathrm{trsf}(\mathbf{p}))$

Proof. By induction on the length of the run. If $\mathbf{T} = \mathbf{I}_R$ it is immediate by definition of trsf. If $\mathbf{I}_R = \mathbf{T}_0 \rightarrow \ldots \rightarrow \mathbf{T}_{n-1} \rightarrow \mathbf{T}_n$, it is immediate to check that tracing back from the positions in $\mathrm{trsf}(\mathbf{T}_n)$ reaches the same positions as $\mathrm{trsf}(\mathbf{T}_{n-1})$. □

We now can prove that $[\![R]\!] = [\![R']\!]$ as partial functions.

Proof. We know that $\mathtt{INIT}_R = \mathtt{INIT}_{R'}$ (let us call this set $\mathtt{INIT}$) and $\mathtt{FIN}_R = \mathtt{FIN}_{R'}$ (let us call this set $\mathtt{FIN}$). Assume that $\mathcal{M}_R$ terminates in the state $\mathbf{T}$ and $\mathcal{M}_{R'}$ in the state $\mathbf{T}'$. We know that trsf $T = T'$ (because trsf T is a terminal final state of $\mathcal{M}_{R'}$, and such a state is unique); therefore $\mathtt{Current}_{\mathbf{T}} \cap \mathtt{FIN} = \mathtt{Current}_{\mathbf{T}}{}' \cap \mathtt{FIN}$. Let us call this set X. It is immediate that if $\mathbf{p} \in X$ then $\mathbf{p} = \mathrm{trsf}\ \mathbf{p}$. Finally, by Lemma 37, for each $\mathbf{s} \in \mathtt{INIT}$ and each $\mathbf{p} \in X$, $\mathrm{orig}_R(\mathbf{p}) = \mathbf{s}$ iff $\mathrm{orig}_{R'}(\mathbf{p}) = \mathbf{s}$. From this we conclude that $[\![R]\!](\mathbf{s}) = [\![R']\!](\mathbf{s})$.

□

C PCF: Proofs

C.1 The PCF Machine

Let us first make precise the notion of transformation.

Transformation. Let $\mathbf{R} = (R, reg_R)$ be a program net and $\mathbf{R} \rightsquigarrow \mathbf{R}' = (R', reg_{R'})$. The transformation trsf described in Fig. 9 associates *positions* of R to *positions* of R'; this allows us also to specify the transformation of the register, and to map a register of $\mathcal{M}_R$ into a register of $\mathcal{M}_{R'}$.

More precisely, each state $(\mathbf{T}, \mathtt{reg}_\mathbf{T})$ of $M_\mathbf{R}$ is mapped into a state $(\mathrm{trsf}(\mathbf{T}), \mathrm{trsf}(\mathtt{reg}_\mathbf{T}))$ of $M_{\mathbf{R}'}$ in the following way. Positions $\mathbf{p} \in \mathbf{T}$ are mapped as described in Figure 9 (see §3.2.6), which directly defines $\mathrm{trsf}(\mathbf{T})$. For the register, let the register ρ (with addresses $\mathtt{ONES}_{R'} \cup \mathtt{INIT}_{R'}$) be defined as follows:

$$\rho[\mathrm{trsf}\ \mathbf{s}] := \mathtt{reg}_\mathbf{T}[\mathbf{s}], \text{ for each } \mathbf{s} \in \mathtt{ONES}_R \cup \mathtt{INIT}_R.$$

We define $\mathrm{trsf}(\mathtt{reg}_\mathbf{T}) := \rho$ in all cases, except when the reduction is an $s.el$ step. In this case, let s be the sync node to be reduced, and $\mathbf{o}_1, \ldots, \mathbf{o}_n$ the positions associated to its premisses; each $\mathbf{o}_i$ belongs to $\mathtt{ONES}_R$, as it is conclusions of a one node. If the multitoken has crossed s (*i.e.* $\{\mathbf{o}_1, \ldots, \mathbf{o}_n\} \subseteq \mathtt{Dom}_\mathbf{T}$ and $\{\mathbf{o}_1, \ldots, \mathbf{o}_n\} \not\subseteq \mathtt{Current}_\mathbf{T}$), we set $\mathrm{trsf}(\mathtt{reg}_\mathbf{T}) := \rho$. Otherwise, if the multitoken has *not* crossed s, we set $\mathrm{trsf}(\mathtt{reg}_\mathbf{T}) := update(s, \mathbf{o}_1, \ldots, \mathbf{o}_n, \rho)$, *i.e.* the transformation applies s to ρ.

Example 38. Let $\mathbf{R} \rightsquigarrow \mathbf{R}'$ be as shown in the first row in Fig. 21 and $\mathbf{o}$ be the (unique) element in $\mathtt{ONES}_\mathbf{R} = \mathtt{ONES}_{\mathbf{R}'}$. The runs in $\mathcal{M}_\mathbf{R}$ and $\mathcal{M}_{\mathbf{R}'}$ are as shown in the second and third row in Fig. 21 respectively, where $\mathbf{T_0}$ and $\mathbf{T'_0}$ are initial states. In this example $\mathrm{trsf}(\mathbf{T_0}) = \mathbf{T'_0}$, $\mathrm{trsf}(\mathbf{T_1}) = \mathrm{trsf}(\mathbf{T_2}) = \mathbf{T'_1}$, and $\mathrm{trsf}(\mathbf{T_3}) = \mathbf{T'_2}$. Observe the following: if the token has not yet crossed the sync node in a state (namely $\mathbf{T_0}$ or $\mathbf{T_1}$), trsf applies update to the register of the state. Otherwise the register of the state is simply copied.

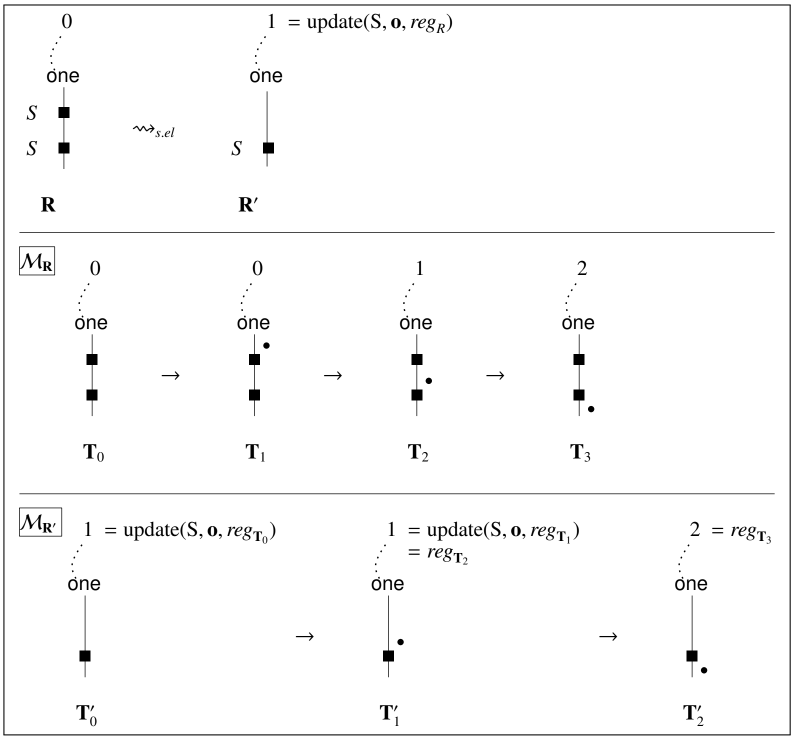


Figure 21: Example of reduction and transition in PCF net

Lemma 39. *Let* $\mathbf{R}$ *be a PCF net. For any reduction* $\mathbf{R} \rightsquigarrow \mathbf{R}'$*,*

1. *If* $\mathbf{T} \to \mathbf{U}$ *in* $\mathcal{M}_{\mathbf{R}}$ *then* $\mathrm{trsf}(\mathbf{T}) \to^* \mathrm{trsf}(\mathbf{U})$ *in* $\mathcal{M}_{\mathbf{R}'}$*.*
2. *If* $\mathbf{T}$ *is an initial state in* $\mathcal{M}_{\mathbf{R}}$*, then so is* $\mathrm{trsf}(\mathbf{T})$ *in* $\mathcal{M}_{\mathbf{R}'}$*.*
3. *If* $\mathbf{T} \not\to$ *in* $\mathcal{M}_{\mathbf{R}}$ *then* $\mathrm{trsf}(\mathbf{T}) \not\to$ *in* $\mathcal{M}_{\mathbf{R}'}$*.*
4. *If* $\mathbf{T}$ *is a final state in* $\mathcal{M}_{\mathbf{R}}$*, then so is* $\mathrm{trsf}(\mathbf{T})$ *in* $\mathcal{M}_{\mathbf{R}'}$*.*
5. *If* $\mathbf{T}$ *is a deadlock state in* $\mathcal{M}_{R}$*, then so is* $\mathrm{trsf}(\mathbf{T})$ *in* $\mathcal{M}_{R'}$*.*

Proof. Each statement can be proved by case analysis.

1. If the transition $\mathbf{T} \to \mathbf{U}$ is not on the redex of $R \rightsquigarrow R'$ then the claim holds, because the positions of tokens and the structure are the same except around the redex. For the reductions except $\rightsquigarrow_{s.el}$, $\rightsquigarrow_{bot.el}$ and $\rightsquigarrow_{decor}$ the proof is the same as that in MELLYS since registers are not modified by the reduction and trsf. For those three rules of reduction relevant to registers:
   - $\underline{\rightsquigarrow_{s.el}}$ We only have to consider the transition $\mathbf{T} \to \mathbf{U}$ that crosses the sync node reduced; by definition of the transformation $\mathrm{trsf}(\mathbf{U}) = \mathrm{trsf}(\mathbf{U})$ holds.
   - $\underline{\rightsquigarrow_{bot.el}}$ If the transition crosses the cut node or enters the box then $\mathrm{trsf}(\mathbf{T}) = \mathrm{trsf}(\mathbf{U})$ by definition. Since no transition can modifiy the value of the register pointed by the one node, the contents chosen by the reduction and by the transitions coincide. Thus for all transitions $\mathbf{T} \to \mathbf{U}$ in the box $\mathrm{trsf}(\mathbf{T}) \to \mathrm{trsf}(\mathbf{U})$ holds.
   - $\underline{\rightsquigarrow_{decor}}$ Since the machines $\mathcal{M}_{\mathbf{R}}$ and $\mathcal{M}_{\mathbf{R}'}$ are the same by definition, for every transition $\mathbf{T} \to \mathbf{U}$ we have $\mathrm{trsf}(\mathbf{T}) \to \mathrm{trsf}(\mathbf{U})$.
2. Almost the same as MELLYS: in the case of $\rightsquigarrow_{s.el}$, the register in the initial state in $\mathcal{M}_{\mathbf{R}}$ is mapped to that in $\mathcal{M}_{\mathbf{R}'}$ by definition of trsf.

3.–5. Same as MELLYS: the definition of terminal/final/deadlock state does not rely on registers.

□

As for the SIAM, we have that

Lemma 40. trsf *maps each run of* M_R *into a run of* $M_{R'}$ *which converges/diverges/deadlocks iff the run on* M_R *does.*

Proof. The proof proceeds in exactly the same way as that of SIAM, because registers do not matter to these properties. □

Fact 41. Let $\mathbf{R}$ be a PCF net. The following holds by definition of trsf:

- Let $\mathbf{R} \rightsquigarrow \mathbf{R}'$ be not a $s.el$ step, let $(\mathbf{T}, \mathtt{reg}_{\mathbf{T}})$ be a state of $\mathcal{M}_{\mathbf{R}}$.
  Then $\mathrm{trsf}(\mathtt{reg}_{\mathbf{T}})[\mathrm{orig}_{R'}(\mathrm{trsf}(\mathbf{p}))] = \mathtt{reg}_{\mathbf{T}}[\mathrm{orig}_{R}(\mathbf{p})]$.
- Let $\mathbf{R} \rightsquigarrow \mathbf{R}'$ be a $s.el$ step, $(\mathbf{T}_1, \mathtt{reg}_{\mathbf{T}_1}) \to (\mathbf{T}_2, \mathtt{reg}_{\mathbf{T}_2}) \to \ldots \to (\mathbf{T}_n, \mathtt{reg}_{\mathbf{T}_n})$ be a run in $\mathcal{M}_{\mathbf{R}}$ containing $(\mathbf{T}_k, \mathtt{reg}_{\mathbf{T}_k}) \to (\mathbf{T}_{k+1}, \mathtt{reg}_{\mathbf{T}_{k+1}})$ crossing the sync node that is reduced. Then $\mathrm{trsf}(\mathtt{reg}_{\mathbf{T}_i})[\mathrm{orig}_{R'}(\mathrm{trsf}(\mathbf{p}))] = \mathtt{reg}_{\mathbf{T}_i}[\mathrm{orig}_{R}(\mathbf{p})]$ holds for $k < i$.

Moreover, similarly to what we have seen in Section 3.3

Lemma 42. *Let* $\mathbf{R}$ *be a PCF net where all conclusions have type* 1*. The machine* $M_{\mathbf{R}}$ *terminates in a final state (say* $(\mathbf{T}, \mathtt{reg}_{\mathbf{T}})$*) iff* $\mathbf{R}$ *reduces to a cut and sync free net (say* $\mathbf{S} = (S, reg_S)$*). Moreover*

$$reg_S = [\![\mathtt{reg}_{\mathbf{T}}]\!]$$

where $[\![\mathtt{reg}_{\mathbf{T}}]\!]$ *is the restriction of* $\mathtt{reg}_{\mathbf{T}}$ *to the elements pointed to by final positions.*

Proof. if Every run in the cut- and sync-free net converges, hence by Lem. 40 any run in $\mathcal{M}_{\mathbf{R}}$ also converges.

only if Similar argument as in the case of the SIAM: if $\mathcal{M}_{\mathbf{R}}$ terminates then the reduction of $\mathbf{R}$ terminates, and the normal form is cut- and sync-free. We observe that –as mentioned in Appendix A, the counterpart of Theorem 2 holds for extended nets and program net, because because the number of structures associated to a *bot* node is irrelevant to the proof of the Theorem, and the register do not affect the possibility of a reduction.

$reg_S = [\![\mathtt{reg}_{\mathbf{T}}]\!]$ Follows from Fact 41. □

Let $\mathbf{R}$ be a PCF net of conclusion 1. We write $\mathbf{R} \Downarrow n$ if $\mathbf{R}$ reduces to $\mathbf{S}$, and n is the value in the register corresponding to the single one node of S. Similarly we write $M_{\mathbf{R}} \Downarrow n$. The next theorem is a corollary of Lem. 42

Theorem 43. $\mathbf{R} \Downarrow n$ if and only if $M_{\mathbf{R}} \Downarrow n$.

C.2 Call-by-name: adequacy (proof of Theorem 21)

In this section we prove Theorem 21, relating the call-by-name encoding in PCF nets and the call-by-name reduction strategy for terms. In order to relate terms and nets rewriting, we introduce an intermediate calculus called extPCF. Its purpose is to decompose the substitution and make it part of the rewrite procedure. The substitution process is decomposed into local term rewrites: the term rewriting of extPCF can then more easily compared to nets rewriting.

Formally, the intermediate language extPCF is defined as follows.

- Terms are extended with the contructor $\mathtt{subst}\, x_1, \dots x_n\, \mathtt{by}\, N_1, \dots N_n\, \mathtt{in}\, M$. Instead of lists $x_1, \dots, x_n$ and $N_1, \dots, N_n$ we shall write $\vec{x}$ and $\vec{N}$.
- The typing rules for this new construct is as follows.

$$\frac{\Delta, x_1 : A_1, \dots x_n : A_n \vdash M : B \quad \forall i,\ \vdash N_i : A_i}{\Delta \vdash \mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M : B}$$

  So in particular, all substituting terms are closed.
- Reductions rules are updated as follows. The new rewriting relation is $\rightarrow_{cbnext}$
  - The beta-redex now reduces with

$$(\lambda x.M)N \rightarrow_{cbnext} \mathtt{subst}\, x\, \mathtt{by}\, N\, \mathtt{in}\, M$$

    and the let-rec reduces as

$$\mathtt{let\,rec}\, f\, x = M\, \mathtt{in}\, N \rightarrow_{cbnext} \mathtt{subst}\, f\, \mathtt{by}\, (\lambda x.\mathtt{let\,rec}\, f\, x = M\, \mathtt{in}\, f\, x)\, \mathtt{in}\, N$$

  - Call-by-name contexts are not touched (i.e. one does not reduce under $\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M$).
  - New reduction rules are added as follows. Alpha-conversion is applied accordingly. We assume that y is not amongst the x_i's.

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, x_i \rightarrow_{cbnext} N_i$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, y \rightarrow_{cbnext} y$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, MP \rightarrow_{cbnext} (\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M)(\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, P)$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, \lambda y.M \rightarrow_{cbnext} \lambda y.(\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M)$$

$$\mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, \langle M, P\rangle \rightarrow_{cbnext} \langle \mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, M, \mathtt{subst}\, \vec{x}\, \mathtt{by}\, \vec{N}\, \mathtt{in}\, P\rangle$$

$$\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,\pi_l(M) \rightarrow_{cbnext} \pi_l(\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,M)$$

$$\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,\pi_r(M) \rightarrow_{cbnext} \pi_r(\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,M)$$

$$\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,\overline{n} \rightarrow_{cbnext} \overline{n}$$

$$\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,S(M) \rightarrow_{cbnext} S(\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,M)$$

$$\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,P(M) \rightarrow_{cbnext} P(\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,M)$$

$$\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,\mathtt{if}\,P\,\mathtt{then}\,M\,\mathtt{else}\,M' \rightarrow_{cbnext} \mathtt{if}\,(\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,P)\,\mathtt{then}\,(\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,M)\,\mathtt{else}\,(\mathtt{subs}$$

$$\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,\mathtt{let\,rec}\,f\,y = M\,\mathtt{in}\,P \rightarrow_{cbnext} \mathtt{let\,rec}\,f\,y = (\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,M)\,\mathtt{in}\,\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,P$$

$$\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,\mathtt{subst}\,y\,\mathtt{by}\,P\,\mathtt{in}\,M \rightarrow_{cbnext} \mathtt{subst}\,\vec{x},y\,\mathtt{by}\,\vec{N},P\,\mathtt{in}\,M$$

We call these additional rules the *substitution reduction.*

- The interpretation $(\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,M)^*$ is simply defined with a cut between M^* and the N_i^*'s.

We finally define a map $\downarrow$ from extPCF to PCF, recursively collapsing in *all* subterms the new construct to the actual substitution:

$$\downarrow(\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,M) \;:=\; (\downarrow M)\{x_i := \downarrow N_1 \mid i = 1\cdots n\}$$

Lemma 44. *Let M be a term in extPCF such that $M \rightarrow_{cbnext} N$. If it is a substitution reduction, then $\downarrow M = \downarrow N$. Otherwise, $\downarrow M \rightarrow_{cbn}^{+} \downarrow N$.*

Proof. Proof by structural induction on the derivation of $M \rightarrow_{cbnext} N$. □

Lemma 45. *If M is a PCF term, and if $M \rightarrow_{cbnext}^{*} \overline{n}$, then $M \rightarrow_{cbn}^{*} \overline{n}$.*

Proof. Since M is a PCF-term, it does not contain any subst-term, and $\downarrow M = M$. Also, $\downarrow \overline{n} = \overline{n}$. We then prove the result by simple induction on the size of the rewrite sequence $M \rightarrow_{cbnext}^{*} \overline{n}$, invoking Lemma 44 in the induction case. □

Lemma 46. *Let P be a term in extPCF such that $\downarrow P \rightarrow_{cbn} M$. Then there exists a term P' in extPCF such that $\downarrow P' = M$ and $P \rightarrow_{cbnext}^{+} P'$.*

Proof. Proof by induction on the size of P. □

Lemma 47. *Suppose that N does not substitution-reduce. Suppose furthermore that $\downarrow M = \downarrow N$, and that $M \rightarrow_{cbnext} M'$ is not a substitution-reduction step. Then there exists N' such that $\downarrow M' = \downarrow N'$ and $N \rightarrow_{cbnext}^{+} N'$.*

Proof. The proof is done by structural induction on M, and by case distinction on $M \rightarrow_{cbnext} M'$. □

Definition 48. We define the substitution-size $ss(M)$ of a term M inductively as follows.

- $ss(\mathtt{subst}\,\vec{x}\,\mathtt{by}\,\vec{N}\,\mathtt{in}\,M)$ is defined as

$$\left(\prod_i (1 + ss(N_i))\right)^{ss(M)}$$

- the substitution-size of terms built from any other constructors is 1 plus the sum of the substitution size of their constituants. For example:

$$ss(x) = ss(\overline{n}) = 1$$

$$ss(\pi_l M) = 1 + ss(M)$$

$$ss(MN) = ss(\langle M, N\rangle) = 1 + ss(M) + ss(N)$$

$$ss(\mathtt{let\,rec}\,f\,x = M\,\mathtt{in}\,N) = 1 + ss(M) + ss(N)$$

$$ss(\mathtt{if}\,P\,\mathtt{then}\,M\,\mathtt{else}\,N) = 1 + ss(M) + ss(N) + ss(P)$$

Lemma 49. *If P is a strict subterm of M, then $ss(P) < ss(M)$. If M substitution-reduces to N, then $ss(N) < ss(M)$.*

Proof. The first part of the lemma is easy to check by structural induction on M, realizing that for all M, $ss(M) \geq 1$. The second part of the lemma is shown by induction on the derivation of $M \to_{cbnext} N$.

- $\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,MP \to_{cbnext} (\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,M)(\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,P)$.

  Provided that we set $x := \prod_i(1 + ss(N_i))$, we have to show
  $$x^{1+ss(M)+ss(P)} > 1 + x^{ss(M)} + x^{ss(P)}.$$
  With $a := x^{ss(M)}$ and $b := x^{ss(P)}$, this can be rewritten as
  $$x \cdot a \cdot b > 1 + a + b.$$
  Since $x \geq 2$, it is enough to show that
  $$2 \cdot a \cdot b > 1 + a + b.$$
  This is equivalent to
  $$a \cdot (2 \cdot b - 1) > b + 1.$$
  Since $a \geq 2$, it is enough to show that
  $$2 \cdot (2 \cdot b - 1) > b + 1,$$
  that is,
  $$3 \cdot b - 3 > 0.$$
  Since $b \geq 2$, this is always verified.

- $\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,\lambda y.M \to_{cbnext} \lambda y.(\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,M)$.

  Provided that we set $x := \prod_i(1 + ss(N_i))$, we have to show
  $$x^{1+ss(M)} > 1 + x^{ss(M)}.$$
  With $a := x^{ss(M)}$, and since $x \geq 2$, it is enough to show
  $$2 \cdot a > 1 + a.$$
  This inequality is valid since $a \geq 2$.

- $\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,\texttt{if}\,P\,\texttt{then}\,M\,\texttt{else}\,M' \to_{cbnext}$
  $\texttt{if}\,(\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,P)\,\texttt{then}\,(\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,M)\,\texttt{else}\,(\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,M)$.

  Provided that we set $x := \prod_i(1 + ss(N_i))$, we have to show
  $$x^{1+ss(P)+ss(M)+ss(M')} > 1 + x^{ss(P)} + x^{ss(M)} + x^{ss(M')}.$$
  With $a := x^{ss(P)}$, $b := x^{ss(M)}$ and $c := x^{ss(M')}$ this can be rewritten as
  $$x \cdot a \cdot b \cdot c > 1 + a + b + c.$$
  Since $x \geq 2$, it is enough to show that
  $$2 \cdot a \cdot b \cdot c > 1 + a + b + c.$$
  Since b and c are larger or equal to 2, we have $b \cdot c > b + c$, and writing $d := b + c$, it is enough to show
  $$a \cdot (2 \cdot d - 1) > d + 1.$$
  We are back to the situation of first bullet point: this inequality is then valid.

- $\texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{N}\,\texttt{in}\,\texttt{subst}\,\vec{y}\,\texttt{by}\,\vec{M}\,\texttt{in}\,P \rightarrow_{cbnext} \texttt{subst}\,\vec{x},\vec{y}\,\texttt{by}\,\vec{N},\vec{M}\,\texttt{in}\,P$.

  Provided that we set $x := \prod_i(1+ss(N_i))$, $y := \prod_i(1+ss(M_i))$ and that we write $a := ss(P)$, we have to show
  $$x^{y^a} > (x \cdot y)^a.$$
  Since $y \geq 2$ and $a \geq 1$, we have $y^a > y \cdot a$. It is then enough to show
  $$x^{y \cdot a} > (x \cdot y)^a.$$
  This is equivalent to
  $$(x^y)^a > (x \cdot y)^a,$$
  and it is correct since $x^y > x \cdot y$.

The other cases are treated similarly. □

Lemma 50. *Suppose that $\downarrow M = \downarrow N$, and that $M \rightarrow^*_{cbnext} b$, where b is either a term variable or a constant $\overline{n}$. Then $N \rightarrow^*_{cbnext} b$.*

Proof. The proof is done by induction on the substitution-size of N.

- Base case: $N = c$, where c is a term constant or a term variable. We need to show that $c = b$.

  We proceed by induction on the size of the sequence of reduction from M to b.

  - If $M = b$, then clearly $c = b$.
  - Suppose that for all terms reducing in n steps to b then $c = b$. Consider a sequence of reduction $M \rightarrow_{cbnext} M' \rightarrow^*_{cbnext} b$ of size $n+1$. Since $N = c$, we have $\downarrow M = c$. There are two cases:
    - Either $M = c$, in which case we get a contradiction since M cannot reduce to any M'.
    - Or $M = \texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{P}\,\texttt{in}\,P'$. In this case, M' is necessarily coming from a substitution-reduction, meaning that $\downarrow M' =\downarrow M = c$. By induction hypothesis we conclude that $c = b$.

  Therefore, in the case where N is a constant or a variable, it is indeed equal to b.

- Now, suppose that it is neither a constant nor a variable, and that the result is correct for all terms of smaller substitution-size.

  We again proceed by induction on the size of the sequence of reduction from M to b.

  - If $M = b$, then $\downarrow N = b$. There are two cases.
    - Either $N = b$, and we are done.
    - Or $N = \texttt{subst}\,\vec{x}\,\texttt{by}\,\vec{P}\,\texttt{in}\,P'$. It then reduces through a substitution-reduction to some term that is substitution-smaller than N, from Lemma 49. We can then invoke the induction hypothesis, and deduce that N reduces to b.
  - Otherwise, suppose tht $M \rightarrow_{cbnext} M' \rightarrow^*_{cbnext} b$, and that the result is true for M'. We know that $\downarrow M = \downarrow M'$. Without loos of generality one can suppose that N does not substitution-reduce: otherwise, we could invoke Lemma 49 and the outer-most induction hypothesis as above to conclude.

    We proceed by case distinction on $M \rightarrow_{cbnext} M'$.
    - If it is a substitution-reduction step, then $\downarrow M = \downarrow M' = \downarrow N$, and the inner induction hypothesis tells us that $N' \rightarrow^*_{cbnext} b$.
    - Otherwise, we first apply Lemma 47, conclude to the existence of N' such that $N \rightarrow_{cbnext} N'$ with $\downarrow N' = \downarrow M'$, and apply the inner induction hypothesis to conclude.

This closes the proof of the lemma. □

Lemma 51. *If N is a term in extPCF such that $\downarrow N = b$, where b is either a term variable or a constant $\overline{n}$, then $N \to^*_{cbnext} b$.*

Proof. This is a corollary of Lemma 50, when setting $M = b$. Indeed, in that case we trivially have that $\downarrow M = \downarrow N$ and that $M \to^*_{cbnext} b$, so the lemma applies. □

Lemma 52. *If M is a PCF term, then $M \to^*_{cbn} \overline{n}$ if and only if $M \to^*_{cbnext} \overline{n}$.*

Proof. The right-to-left direction is Lemma 45. For the left-to-right direction, remark that $\downarrow M = M$ and $\downarrow \overline{n} = \overline{n}$. Then consider the reduction sequence

$$M \to_{cbn} M_1 \to_{cbn} \cdots \to_{cbn} M_n = \overline{n}.$$

Applying Lemma 46 on each step of this sequence, one construct a sequence of reduction showing that $M \to^*_{cbnext} M'_n$ where $\downarrow M'_n = \overline{n}$. We finally conclude with Lemma 51 □

Lemma 53. *Suppose that M and N are closed terms of type A. If $M \to_{cbnext} N$, then $M^* \to^+ N^*$.*

Proof. The proof is done by structural induction on the derivation of the reduction $M \to_{cbnext} N$. □

Lemma 54. *Suppose that a net R converges to a normal form. Then any sequence of reduction starting from R is finite and terminates on the same normal form.*

Proof. As in Proposition 1. □

Lemma 55. *Suppose that M is a closed term of type $\mathbb{N}$. Then $M \to^*_{cbnext} \overline{n}$ if and only if $M^* \to^* \overline{n}^*$.*

Proof. The left-to-right direction is proven by simple induction on the size of the rewrite sequence $M \to^*_{cbnext} \overline{n}$, using Lemma 53.

For the right-to-left direction, suppose that $M^* \to^* \overline{n}^*$ but that there exists an infinite sequence $\{M_i\}_{i \in \mathbb{N}}$ such that $M = M_0$ and such that for all i, $M_i \to_{cbnext} M_{i+1}$. Then using Lemma 53 we can conclude that there is an infinite net-rewrite sequence starting with M^*. From Lemma 54 this contradicts the fact that M^* converges. □

Proof of Theorem 21. The desired adequacy result simply follows from the use of Lemma 52 to fall back on the intermedidate PCF and Lemma 55.