

Par Means Parallel: Multiplicative Linear Logic Proofs as Concurrent Functional Programs

FEDERICO ASCHIERI*, TU Wien, Austria

FRANCESCO A. GENCO†, Université Paris 1 Panthéon-Sorbonne, France

Along the lines of Abramsky’s “Proofs-as-Processes” program, we present an interpretation of multiplicative linear logic as typing system for concurrent functional programming. In particular, we study a linear multiple-conclusion natural deduction system and show it is isomorphic to a simple and natural extension of λ -calculus with parallelism and communication primitives, called $\lambda_{\mathcal{N}}$. We shall prove that $\lambda_{\mathcal{N}}$ satisfies all the desirable properties for a typed programming language: subject reduction, progress, strong normalization and confluence.

CCS Concepts: • **Theory of computation** → **Lambda calculus**; **Type theory**.

Additional Key Words and Phrases: Proofs-as-programs, concurrency, classical multiplicative linear logic

ACM Reference Format:

Federico Aschieri and Francesco A. Genco. 2020. Par Means Parallel: Multiplicative Linear Logic Proofs as Concurrent Functional Programs. *Proc. ACM Program. Lang.* 4, POPL, Article 18 (January 2020), 28 pages. <https://doi.org/10.1145/3371086>

1 INTRODUCTION

1.1 The Proofs-as-Processes Program

Introduced by Girard in 1987, linear logic was announced right off the bat as the logic of parallelism. According to [Girard 1987], the “connectives of linear logic have an obvious meaning in terms of parallel computation (...) In particular, the multiplicative fragment can be seen as a system of communication without problems of synchronization.” Stimulated by this remark and the further observation that “cut elimination is parallel communication between processes” ([Girard et al. 1989], pp. 155), Abramsky [Abramsky 1994] launched the “Proofs-as-Processes” program, whose goal was to support with computational evidence Girard’s claims. Namely, the goal was “to show how a process calculus, sufficiently expressive to allow a reasonable range of concurrent programming examples to be handled, could be exhibited as the computational correlate of a proof system” [Abramsky 1994]. A candidate process calculus was Milner’s π -calculus [Milner et al. 1992]; a possible proof system, the linear sequent calculus. The hope was that linear logic could provide a canonical and firm foundation to concurrent computation, serving as a tool for typing and reasoning about communicating processes.

*Funded by FWF grant P32080-N31.

†Funded by ANR JCJC project *Intuitions Bolzaniennes* and by FWF project P32080-N31.

Authors’ addresses: Federico Aschieri, TU Wien, Vienna, Austria; Francesco A. Genco, IHPST, Université Paris 1 Panthéon-Sorbonne, Paris, France.



This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License.

© 2020 Copyright held by the owner/author(s).

2475-1421/2020/1-ART18

<https://doi.org/10.1145/3371086>

1.2 Early Successes and Limitations

After early works of [Bellin and Scott 1994] and [Beffara 2006], the turning point was the discovery by Caires and Pfenning [Caires and Pfenning 2010] of a tight correspondence between intuitionistic linear logic and a session typed π -calculus, the motto being: linear propositions as session types, proofs as processes, cut-elimination as communication. This was yet another instance of the famous Curry-Howard correspondence [Wadler 2015], linking proofs to programs, propositions to types, proof normalization to computation. Soon after, Wadler [Wadler 2012] introduced the session typed π -calculus CP, shown to tightly correspond to classical linear logic.

This important research notwithstanding, it appears that there is still ground to cover toward canonical and firm foundations for concurrent computation. First of all, asynchronous communication does not appear to rest on solid foundations. Yet this communication style is easier to implement and more practical than the purely synchronous paradigm, so it is widespread and asynchronous typed process calculi have been already investigated (see [Gay and Vasconcelos 2010]). Unfortunately, linear logic has not so far provided via Curry-Howard a logical account of asynchronous communication. The reason is that there is a glaring discrepancy between *full* cut-elimination in sequent calculus and π -calculus reduction which has not so far been addressed. On one hand, as we shall see, linear logic does support asynchronous communication, but only through the *full* process of cut-elimination, which indeed makes essential use of asynchronous communication. On the other hand, the π -calculus of [Caires and Pfenning 2010] only mimics a *partial* cut-elimination process that only eliminates top-level cuts. Indeed, by comparison with its type system, the π -calculus lacks some necessary computational reduction rules. Some of the missing reductions, corresponding to commuting conversions, were provided in Wadler's CP. The congruence rules that allow the extra reductions to mirror full cut-elimination, however, were rejected: in Wadler's [Wadler 2012] own words, “*such rules do not correspond well to our notion of computation on processes, so we omit them*”. A set of reductions similar to that rejected by Wadler is regarded in [Pérez et al. 2012] as sound relatively to a notion of “typed context bisimilarity”. The notion, however, only ensures that two “bisimilar” processes have the same input/output behaviour along their main communication channel; the internal synchronization among the parallel components of the two processes is not captured and may differ significantly. Thus, the extensional flavor of the bisimilarity notion prevents it from detecting the intensionally different behaviour of the related processes, that is, how differently they communicate and compute.

Attempts to model asynchronous communication by mirroring full cut-elimination with Milner's π -calculus are indeed bound to fail. An example will clarify the matter. In the process $vz(x(y).z(w).P \mid z[a].Q)$, the second process $z[a].Q$ wishes to transmit the message a along the channel z to the first process. However, the first process is not ready to receive on z yet: it is designed to first receive a message on the channel x . In Milner's π -calculus, this is a deadlock, because input and output actions are rigidly ordered and *blocking*, for the sake of synchronization. Indeed, as the heir of CCS [Milner 1980], π -calculus was born as formalism for representing synchronization: a communication happens when two processes synchronize by consuming two dual constructs, as for instance $\bar{x}m$ and $x(y)$. However, when a process is seen as a proof in the logical system, the order between prefixes becomes less relevant, thus in CP there is an extra commuting conversion

$$vz(x(y).z(w).P \mid z[a].Q) \mapsto x(y).vz(z(w).P \mid z[a].Q)$$

In CP, there is no further reduction allowed, and rightly so: otherwise the blocking nature of the prefix $x(y)$ would be violated, making the very construct pointless for synchronization. On the other hand, for cut-elimination the now possible reduction

$$x(y).vz(z(w).P \mid z[a].Q) \mapsto x(y).vz(P[a/w] \mid Q)$$

is needed. In this case, however, as we have seen the very existence of a construct for input becomes misleading and it would be more coherent to drop it altogether, so that the first process would be directly re-written and reduced as follows

$$\nu z(P \mid w[a].Q) \mapsto \nu z(P[a/w] \mid Q)$$

This is indeed the approach of the present paper. Sensitive to similar concerns, [Kokke et al. 2019] introduced HCP, which features delayed input-output actions, modeling a form of asynchrony. However, the change in the transitions with respect to CP undermines the synchronous nature of Milner’s π -calculus, as we have seen, making HCP quite a different calculus and questioning its canonicity. Moreover, HCP comes at the cost of adding more structure, in the form of the hypersequent separator [Avron 1991], to the linear sequent calculus. In HCP this structure is logically redundant, whereas Avron employed it to *increase* the logical power of sequents. On the other hand, the added structure enables HCP to enjoy a nice isomorphism with proofs.

There are other limitations in the current linear logic foundations of concurrent computation. The linear sequent calculus fails to combine seamlessly functional and concurrent computation. Indeed, according to Abramsky et al. [Abramsky et al. 1996] it is a “*major open problem (...) to combine our understanding of the functional and concurrent paradigms, with their associated mathematical underpinnings, in a single unified theory*”. The sequent calculus falls short in several other respects: it prevents deadlocks, but by excessively restricting possible communication patterns, for instance by forbidding cyclic configurations; it does not permit in any simple way code mobility, that is, the ability to communicate code, like in higher-order π -calculus [Sangiorgi 1993]; it does not represent multi-party communication sessions; its syntax does not match exactly the syntax of processes. In order to address some of these pitfalls, much work has been done, by either considering non-standard proof systems for linear logic or by extending the logic itself; unfortunately for the canonical-foundation enterprise and practical usability, these systems tends to be proof-theoretically artificial. A system adding a functional language on top of linear-logic-based typed π -calculus was studied in [Toninho et al. 2013]. A solution for allowing, also in the context of the session typed π -calculus, cyclic communication patterns, was given in [Dardha and Gay 2018] by coming up with a new variant of linear logic; this makes it possible, for example, to implement Milner’s scheduler process. Multi-party session types have been treated for example in [Carbone et al. 2017]. A hypersequent proof system for linear logic whose syntax perfectly matches the syntax of π -calculus has been introduced in [Kokke et al. 2019]. A logic for typing asynchronous communication in a non-standard process calculus has been proposed in [Pruiksmas and Pfenning 2019].

1.3 $\lambda_{\mathcal{N}}$: A Concurrent Extension of λ -Calculus Arising from Linear Logic

We shall illustrate a new approach in the study of linear logic as typing system for concurrent programs. Our framework does not suffer any of the mentioned limitations of linear sequent calculus. Inspired by [Eades III and de Paiva 2016; Parigot 1991], we move away from sequent calculus and adopt a multiple-conclusion natural deduction for multiplicative linear logic, which we show to be isomorphic to a concurrent extension of λ -calculus, called $\lambda_{\mathcal{N}}$. As a result, $\lambda_{\mathcal{N}}$ is typed by linear multiple-conclusion natural deduction; it is naturally asynchronous, yet it can model synchronous communication as well; it allows arbitrary communication patterns, like cyclic ones, but prevents deadlocks; it allows higher-order communication and code mobility; it mirrors perfectly a full normalization procedure resulting in analytic proofs; its types can be read as program specifications in the traditional way, although they do not describe communication protocols as session types do.

The typing system of $\lambda_{\mathcal{N}}$, in its non-linear variant, has been very well studied: computationally, it was famously interpreted by Parigot [Parigot 1992] as $\lambda\mu$ -calculus; proof-theoretically, it was

thoroughly investigated by [Cellucci 1992]. Although proof-nets have sometimes been dubbed “*the natural deduction of linear logic*”, our typing system is closer to a natural deduction. It is not exactly *natural* deduction [Prawitz 1971], since it is multiple-conclusion, hence it is natural for building and typing parallel programs, but not much so for modeling *human* deduction. However, our system is based on natural-deduction normalization rather than sequent-calculus cut-elimination and linear implication \multimap is a primitive connective, with the standard introduction and elimination rules. Like proof-nets (see [Girard 1987], [Honda and Laurent 2010]), $\lambda_{\mathfrak{N}}$ abstracts away from those inessential permutations in the order of rules that plague multiple-conclusion logical systems. As a result, $\lambda_{\mathfrak{N}}$ avoids commuting conversions, which have never been convincing from the computational point of view. Actually, $\lambda_{\mathfrak{N}}$ is not only a concurrent λ -calculus, it also looks like a natural deduction version of proof nets. Finally, it enjoys all the good properties that a well-behaved functional programming language should have: subject reduction, progress, strong normalization, and confluence. It is a step forward in the direction of that elusive concurrent λ -calculus which Milner attempted to find, before creating CCS out of the failure¹.

1.4 The Insight

The main insight behind the typing system of $\lambda_{\mathfrak{N}}$ is to break into two rules the linear implication introduction. One rule is intuitionistic and yields functional abstraction, the second is classical and yields communication. Namely, the first rule, taken from [Eades III and de Paiva 2016], introduces the λ -operator, the second a communication channel with continuation:

$$\frac{\Gamma, x : A \Rightarrow t : B, \Delta}{\Gamma \Rightarrow \lambda x t : A \multimap B, \Delta} \multimap I \text{ (if } x \text{ occurs in } t) \quad \frac{\Gamma, x : A \Rightarrow t : B, \Delta}{\Gamma \Rightarrow \bar{x}. t : A \multimap B, \Delta} \multimap I \text{ (if } x \text{ occurs in } \Delta)$$

Indeed, a communication channel $\bar{x}. t$ is supposed to take as argument a term $u : A$, transmit it along the channel x and then continue with $t : B$. We shall indeed denote the application of $\bar{x}. t$ to u as $\bar{x}u.t$, as in π -calculus: a promising sign!

The parallel operator $|$ is introduced by the \mathfrak{N} introduction rule:

$$\frac{\Gamma \Rightarrow s : A, t : B, \Delta}{\Gamma \Rightarrow s | t : A \mathfrak{N} B, \Delta} \mathfrak{N}I$$

finally doing justice to the \mathfrak{N} connective, called *par* since the beginning.

2 MULTIPLICATIVE LINEAR LOGIC AS A CONCURRENT FUNCTIONAL CALCULUS

In this section, we present the syntax, the typing system NMLL (Natural deduction for Multiplicative Linear Logic) and the reduction rules of the concurrent functional calculus $\lambda_{\mathfrak{N}}$.

¹According to Milner [Milner 1984], “CCS is an attempt to provide a paradigm for concurrent computation. It arose after several unsuccessful attempts by the author to find a satisfactory generalization of the lambda calculus, to admit concurrent computation.”

2.1 The Calculus $\lambda_{\mathfrak{N}}$

Definition 2.1 (Terms of $\lambda_{\mathfrak{N}}$). The *untyped terms* of $\lambda_{\mathfrak{N}}$ are defined by the following grammar:

$u, v ::= \circ$	(terminated process)
$\mid \bar{x}. u$	(output channel \bar{x} with continuation)
$\mid \bar{x} \bar{y} u$	(output channels \bar{x}, \bar{y} without continuation)
$\mid u \mid v$	(parallel composition)
$\mid \text{close}(u)$	(process termination)
$\mid x$	(variable)
$\mid u v$	(function application)
syntactic sugar	
$\mid \lambda x u := \bar{x}. u$	(function definition (provided x occurs in u))
$\mid \bar{x}v.u := (\bar{x}. u)v$	

The calculus $\lambda_{\mathfrak{N}}$ is nothing but a linear lambda calculus extended with communication primitives. When x occurs in u , the operator $\bar{x}. u$ is denoted and behaves exactly as $\lambda x u$. When x does not occur in u , the term $\bar{x}. u$ behaves as an output channel \bar{x} followed by a continuation u . Channels in $\lambda_{\mathfrak{N}}$ are thus first-class citizens that can be transmitted, shared, moved and applied to messages just like functions to arguments. When the channel $\bar{x}. u$ is applied to an argument v , it will be denoted as $\bar{x}v.u$, because it behaves exactly as its π -calculus counterpart: it transmits the message v and continues with u . The term $\bar{x} \bar{y} u$ transmits the two messages contained in u along the channels \bar{x} and \bar{y} and then terminates. Indeed, as it will be clear from the type system, the term u inside $\bar{x} \bar{y} u$ must be of type $A \mathfrak{N} B$ and must contain two messages: the term of type A and the term of type B .

If we omit parentheses, each term of $\lambda_{\mathfrak{N}}$ can be rewritten, not uniquely, as a parallel composition of several terms: $t_1 \mid t_2 \mid \dots \mid t_n$. We will adopt this notation throughout the paper since it is irrelevant how the parallel operator \mid associates.

Remark 2.1. An alternative choice for the syntax of $\lambda_{\mathfrak{N}}$ may have been:

$$u, v ::= x \mid uv \mid \lambda x u \mid \bar{x}v.u \mid u \mid v \mid \bar{x} \bar{y} u \mid \circ \mid \text{close}(u)$$

Then the operator $\bar{x}v.u$ would have been typed by a cut-rule and the construct $\bar{x}. u$ would have been defined as: $\lambda y \bar{x}y.u$. This approach would have raised some technical complications: when transmitting a message v , one would have to deal with the free variables of v which are bound by λ operators occurring outside v . A similar issue is solved in [Aschieri et al. 2018], and a simpler solution could be adopted here. Nevertheless, we adopt the present approach, as it is the smoother.

2.2 The Typing System NMLL

Definition 2.2 (Types). The **types** of NMLL are built in the usual way from propositional variables, \perp and the connectives \multimap and \mathfrak{N} .

Since the typing system of $\lambda_{\mathfrak{N}}$ is classical linear logic, the \otimes connective can be defined in terms of the others. Indeed, its computational interpretation seems less primitive, as it defines no computational construct that is not already captured by \multimap and \mathfrak{N} . Hence, we leave it out of the system.

Definition 2.3 (Sequents). The judgments of the typing system NMLL for $\lambda_{\mathfrak{N}}$ are **sequents** $\Gamma \Rightarrow \Delta$, where:

1. $\Gamma = x_1 : A_1, \dots, x_n : A_n$ is a sequence of distinct variable declarations, each x_i being a variable, A_i a type and $x_i \neq x_j$ for $i \neq j$.
2. $\Delta = t_1 : B_1, \dots, t_n : B_n, t_{n+1}, \dots, t_m$, each t_i being a term of $\lambda_{\mathfrak{N}}$ and B_i a type.

The typing system of $\lambda_{\mathfrak{N}}$ is a linear version of the multiple-conclusion classical natural deduction of [Cellucci 1992]. It is presented in Table 1 and discussed in detail below. As usual, we interpret a derivable sequent $\Gamma \Rightarrow \Delta$ as a judgement typing a sequence of terms Δ , provided their variables are used with types declared in Γ . The terms in Δ that have no type are processes that return no value, and always have the form $\text{close}(u)$, with $u : \perp$. These terms cannot be ignored, as they may be involved in some communication; thus they are evaluated and then terminated. The $\text{close}()$ construct is introduced by the $\perp\text{E}$ rule and is essential to keep a tight correspondence between logic and calculus, as explained at the beginning of Section 5.

2.3 Linearity of Variables

The only condition that the typing rules must satisfy in order to be applied is that their premises share no variable. This amounts to requiring the linearity of variables and to making sure that there is no clash among the names of the output channels. As we shall see in Proposition 7.1, the result of this implicit renaming is that a typed term cannot contain two distinct occurrences of the same output channel \bar{x} nor of the same input channel x . This property has two consequences. First, in $\lambda_{\mathfrak{N}}$ we do not need, and actually do not use, any renaming during reductions. Secondly, communication channels cannot be used twice: once a message is transmitted along a channel, the channel is consumed and disappears forever, both from the sender and from the receiver of the message. These two processes can of course keep communicating with each other or with other processes, but they must use different channels for further communications.

Table 1. The type assignment rules of NMLL.

$x : A \Rightarrow x : A$		
$\frac{\Gamma, x : A \Rightarrow t : B, \Delta}{\Gamma \Rightarrow \lambda x t : A \multimap B, \Delta} \multimap\text{I (if } x \text{ occurs in } t)$	$\frac{\Gamma, x : A \Rightarrow t : B, \Delta}{\Gamma \Rightarrow \bar{x}. t : A \multimap B, \Delta} \multimap\text{I (if } x \text{ occurs in } \Delta)$	
$\frac{\Gamma \Rightarrow s : A \multimap B, \Delta \quad \Sigma \Rightarrow t : A, \Theta}{\Gamma, \Sigma \Rightarrow st : B, \Delta, \Theta} \multimap\text{E}$	$\frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \circ : \perp} \perp\text{I}$	$\frac{\Gamma \Rightarrow t : \perp, \Delta}{\Gamma \Rightarrow \text{close}(t), \Delta} \perp\text{E}$
$\frac{\Gamma \Rightarrow s : A, t : B, \Delta}{\Gamma \Rightarrow s \mid t : A \wp B, \Delta} \wp\text{I}$	$\frac{\Gamma \Rightarrow s : A \wp B, \Delta \quad \Sigma_1, x : A \Rightarrow \Theta_1 \quad \Sigma_2, y : B \Rightarrow \Theta_2}{\Gamma, \Sigma_1, \Sigma_2 \Rightarrow \bar{x} \bar{y} s : \perp, \Delta, \Theta_1, \Theta_2} \wp\text{E}$	
Each rule in this table can only be applied if its premises share no variable		

2.4 \wp as a Parallel Operator

The typing system of $\lambda_{\mathfrak{N}}$ is tailored to parallel computation. It is based on the view that the different processes of a parallel program are not independent entities, but make sense only as components of a larger and more complex system: they share resources through communication and they exploit the computational power of one another. Hence, one cannot build and type one process, without building at the same time all the interconnected processes. Coherently, a derivation of a sequent $\Gamma \Rightarrow t_1 : A_1, \dots, t_n : A_n, t_{n+1}, \dots, t_m$ builds in parallel the terms t_1, \dots, t_m , following the logical structure of the intended communication pattern. The commas in the conclusion of a sequent, indeed, mean exactly parallel composition. Since the comma, in the final analysis, is the \wp

connective, we obtain the following typing rule:

$$\frac{\Gamma \Rightarrow s : A, t : B, \Delta}{\Gamma \Rightarrow s \mid t : A \wp B, \Delta} \wp I$$

As the \wp is commutative and associative, so is the comma. Our reduction rules, indeed, treat parallel composition as commutative and associative, since they bypass the order and association of processes. Technically, however, the term $s \mid t : A \wp B$ is different from the term $t \mid s : B \wp A$, since $A \wp B$ and $B \wp A$ are logically equivalent, but not the same type. It would be possible to add explicit commutation and association congruences on the outermost parallel operators \mid , provided Subject Reduction is restated modulo logical equivalence. Since there is no computational gain in adding such congruences, we do not consider them.

2.5 Linear λ -Calculus

λ_{\wp} contains the linear λ -calculus as a subsystem. On one hand, function application and variable declaration are respectively given by the rule $\multimap E$ and the axiom:

$$\frac{\Gamma \Rightarrow s : A \multimap B, \Delta \quad \Sigma \Rightarrow t : A, \Theta}{\Gamma, \Sigma \Rightarrow st : B, \Delta, \Theta} \multimap E \qquad x : A \Rightarrow x : A$$

as usual. On the other hand, the treatment of linear implication is the main novelty of the type system of λ_{\wp} . Namely, the linear implication introduction rule is broken in two rules. The first is intuitionistic and yields functional abstraction, the second is classical and yields communication. Functional abstraction λ is obtained by the rule:

$$\frac{\Gamma, x : A \Rightarrow t : B, \Delta}{\Gamma \Rightarrow \lambda x t : A \multimap B, \Delta} \multimap I \text{ (if } x \text{ occurs in } t)$$

Such a rule has been studied by [Eades III and de Paiva 2016] in the context of multiple-conclusion intuitionistic linear logic. Indeed, when x occurs in t , the term $\lambda x t$ is a standard linear λ -term. The application of a function to an argument generates in λ_{\wp} the standard λ -calculus reduction

$$(\lambda x u)t \mapsto u[t/x]$$

where $u[t/x]$ is the term obtained from u by replacing the occurrence of x with t .

2.6 Communication

The other rule for linear implication introduction is:

$$\frac{\Gamma, x : A \Rightarrow t : B, \Delta}{\Gamma \Rightarrow \bar{x}.t : A \multimap B, \Delta} \multimap I \text{ (if } x \text{ occurs in } \Delta)$$

In this case, the variable x does not occur in t , therefore the term $\bar{x}.t$ is no more a λ -term and behaves like an output channel with continuation t . Namely, when $\bar{x}.t$ is applied to an argument u , a communication is triggered: the term u is transmitted along the channel x to another process inside Δ and will replace x inside that process. We thus have in λ_{\wp} the reduction rules:

$$C[\bar{x}u.s] \mid \mathcal{D}[x] \mapsto C[s] \mid \mathcal{D}[u] \qquad C[x] \mid \mathcal{D}[\bar{x}u.s] \mapsto C[u] \mid \mathcal{D}[s]$$

where $C[\]$ and $\mathcal{D}[\]$ are arbitrary contexts, as usual defined as follows.

Definition 2.4 (Contexts). A **context** $C[\]$ is a term containing a variable $[\]$ occurring exactly once. For any term t we denote by $C[t]$ the result of replacing $[\]$ by t in $C[\]$. $C[\]$ is **simple** if it does not contain subterms of the form $\mathcal{D}[\] \mid u$ or $u \mid \mathcal{D}[\]$.

In the reduction above, by definition of context, only one occurrence of x is replaced by u . Therefore, if the context $\mathcal{D}[\]$ is not simple, then the term $\mathcal{D}[x]$ is a parallel composition of several processes and only one of them will actually receive u . For instance, we have

$$\bar{x}u.\circ \mid (\lambda y y \mid zx) \mapsto \circ \mid (\lambda y y \mid zu).$$

Unlike in pure π -calculus, the output operator $\bar{x}u.s$ may transmit its message even if it is surrounded by a potentially non-empty context. This is necessary for fully normalizing the proofs: the alternative is to give up on a full correspondence with the normalization/cut-elimination process, as done in Wadler's typed π -calculus CP [Wadler 2012]. In our case, it might happen that some variable y of the message u lies under the scope of an operator λy occurring in the context. There is no problem, though: the operator λy is just a notation which automatically becomes \bar{y} , if the need arises – namely, if y changes location. For instance, let us consider the natural reduction strategy that avoids to reduce under λ :

$$(\lambda y \bar{x}(y w).\circ)v \mid x \mapsto \bar{x}(v w).\circ \mid x \mapsto \circ \mid v w$$

If we instead first fire the communication reduction along the channel x , we obtain the very same result, but by a different mechanism:

$$(\lambda y \bar{x}(y w).\circ)v \mid x = (\bar{y}.\bar{x}(y w).\circ)v \mid x \mapsto (\bar{y}.\circ)v \mid y w = \bar{y}v.\circ \mid y w \mapsto \circ \mid v w$$

What happens in this second reduction is that the value of y , still not known during the first communication, is automatically redirected and communicated to the new location of y by the second communication. Therefore, the code mobility issues created by closures, solved in [Aschieri et al. 2018] in a more complicated way, literally disappear in $\lambda_{\mathfrak{N}}$.

Similarly, the “capture” of variables during communications by output operators \bar{x} creates no issue. For instance, the result of the reduction above communicating first along x , then along y :

$$\bar{x}v.y \mid \bar{y}x.\circ \mapsto y \mid \bar{y}v.\circ \mapsto v \mid \circ$$

can also be obtained by communicating first along y , then along x , thus allowing the variable x to be *captured* by the \bar{x} operator, which then automatically becomes λx :

$$\bar{x}v.y \mid \bar{y}x.\circ \mapsto \bar{x}v.x \mid \circ = (\lambda x x)v \mid \circ \mapsto v \mid \circ$$

2.7 Output Channels without Continuation

The elimination rule for the connective \mathfrak{N} types binary output channels with no associated continuation, a kind of operator also found in the asynchronous π -calculus [Boudol 1992]:

$$\frac{\Gamma \Rightarrow s : A \mathfrak{N} B, \Delta \quad \Sigma_1, x : A \Rightarrow \Theta_1 \quad \Sigma_2, y : B \Rightarrow \Theta_2}{\Gamma, \Sigma_1, \Sigma_2 \Rightarrow \bar{x} \bar{y} s : \perp, \Delta, \Theta_1, \Theta_2} \mathfrak{N}E$$

When s is a parallel composition $u \mid v$, the term $\bar{x} \bar{y} s$ transmits u and v respectively along the channel x and y and then terminates with no value, as reflected by its type \perp . As a result, exactly one occurrence of x will be replaced by u and exactly one occurrence of y will be replaced by v , in two different processes or in the same. The correspondent reduction rules of $\lambda_{\mathfrak{N}}$ are:

$$\begin{aligned} C[\bar{x} \bar{y} (s \mid t)] \mid \mathcal{D}[x][y] &\mapsto C[\circ] \mid \mathcal{D}[s][t] \\ \mathcal{D}[x][y] \mid C[\bar{x} \bar{y} (s \mid t)] &\mapsto \mathcal{D}[s][t] \mid C[\circ] \\ \mathcal{D}[x] \mid C[\bar{x} \bar{y} (s \mid t)] \mid \mathcal{E}[y] &\mapsto \mathcal{D}[s] \mid C[\circ] \mid \mathcal{E}[t] \\ \mathcal{D}[y] \mid C[\bar{x} \bar{y} (s \mid t)] \mid \mathcal{E}[x] &\mapsto \mathcal{D}[t] \mid C[\circ] \mid \mathcal{E}[s] \end{aligned}$$

The first two reduction rules concern the case in which the variables x and y occur in the same context. The last two address the case in which the variables x and y occur in two different contexts.

2.8 Process Termination

Classical logic is obtained by a combination of the classical linear implication introduction rule and the \perp -introduction rule, which introduces the terminated process \circ , which, in turn, does nothing:

$$\frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \circ : \perp} \perp I$$

We can derive the linear excluded middle as shown below on the left. Since terms of type \perp return no value, we have the typing rule below on the right.

$$\frac{\frac{x : A \Rightarrow x : A}{x : A \Rightarrow x : A, \circ : \perp}}{\Rightarrow x : A, \bar{x}. \circ : A \multimap \perp} \quad \frac{\Gamma \Rightarrow t : \perp, \Delta}{\Gamma \Rightarrow \text{close}(t), \Delta} \perp E$$

which introduces the construct $\text{close}(t)$, whose intended meaning is to execute t and then terminate the computation with no return value, hence it has no type. One could also add the reduction rule $\text{close}(\circ) \mapsto$.

2.9 The Reduction Rules of $\lambda_{\mathcal{N}}$

The complete list of reduction rules of $\lambda_{\mathcal{N}}$ is presented in Table 2 for the reader's convenience.

Table 2. Reduction rules for NMLL terms.

$(\lambda x u)t$	\mapsto	$u[t/x]$
$C[\bar{x}u.s] \mid \mathcal{D}[x]$	\mapsto	$C[s] \mid \mathcal{D}[u]$
$C[x] \mid \mathcal{D}[\bar{x}u.s]$	\mapsto	$C[u] \mid \mathcal{D}[s]$
$C[\bar{x}\bar{y}(s \mid t)] \mid \mathcal{D}[x][y]$	\mapsto	$C[\circ] \mid \mathcal{D}[s][t]$
$\mathcal{D}[x][y] \mid C[\bar{x}\bar{y}(s \mid t)]$	\mapsto	$\mathcal{D}[s][t] \mid C[\circ]$
$\mathcal{D}[x] \mid C[\bar{x}\bar{y}(s \mid t)] \mid \mathcal{E}[y]$	\mapsto	$\mathcal{D}[s] \mid C[\circ] \mid \mathcal{E}[t]$
$\mathcal{D}[y] \mid C[\bar{x}\bar{y}(s \mid t)] \mid \mathcal{E}[x]$	\mapsto	$\mathcal{D}[t] \mid C[\circ] \mid \mathcal{E}[s]$

As usual, for any context $C[\]$, we adopt the reduction scheme $C[t] \mapsto C[u]$ whenever $t \mapsto u$. We denote by \mapsto^* the reflexive and transitive closure of the one-step reduction \mapsto . As usual, we shall employ the λ -calculus concepts of normal form and strong normalization.

Definition 2.5 (Normal Forms and Strongly Normalizable Terms).

- A **redex** is a term u such that $u \mapsto v$ for some v and reduction in Table 2. A term t is called a **normal form** or, simply, **normal**, if there is no u such that $t \mapsto u$.
- A finite or infinite sequence of terms $u_1, u_2, \dots, u_n, \dots$ is said to be a **reduction** of t , if $t = u_1$, and for all i , $u_i \mapsto u_{i+1}$. A term u of $\lambda_{\mathcal{N}}$ is **normalizable** if there is a finite reduction of u whose last term is normal and is **strong normalizable** if every reduction of u is finite.

3 PROGRAMMING WITH $\lambda_{\mathcal{N}}$

In order to acquire familiarity with the main features of λ , we discuss now some programming examples. Namely, we shall see how to implement client-server communication, synchronization, cyclic communication patterns and channel sharing.

Example 3.1 (Client-Server: Request/Answer). In this example, we discuss how to represent in $\lambda_{\mathcal{N}}$ a simple communication protocol, consisting in a request/answer message exchange. A server hosts an online catalogue, mapping product names to prices. A client transmits a string $\text{prod} : \text{String}$

to the server, representing a product name whose price the client wishes to know. The server answers the client request by sending back the price of the product *prod*, computed by the function $\text{cost} : \text{String} \multimap \mathbb{N}$. A naive way to implement client and server would be

$$\overbrace{\bar{x}\text{prod}.y}^{\text{CLIENT}} \mid \overbrace{\bar{y}(\text{cost } x).\circ}^{\text{SERVER}}$$

Although, as expected

$$\bar{x}\text{prod}.y \mid \bar{y}(\text{cost } x).\circ \mapsto y \mid \bar{y}(\text{cost } \text{prod}).\circ \mapsto y \mid \bar{y}(\text{price}).\circ \mapsto \text{price} \mid \circ$$

we observe that, this way, synchronization is implemented poorly. As we see below, the server may send to the client the message $(\text{cost } x)$ before it receives any request whatsoever!

$$\bar{x}\text{prod}.y \mid \bar{y}(\text{cost } x).\circ \mapsto \bar{x}\text{prod}.\text{cost } x \mid \circ = (\lambda x \text{ cost } x)\text{prod} \mid \circ \mapsto^* \text{price} \mid \circ$$

What we expect, instead, is that only the request of the client can trigger computation and transmission of the answer by the server. In order to force that, the trick is to encapsulate the message *prod* into the λ -term $\lambda a \lambda b a (b \text{ prod})$, whose task is to take as input a server channel *a*, a continuation *b* and apply the channel *a* to *b prod*. We thus implement client and server as shown below on the left. Since the server output channel $\bar{y}.\circ$ is not applied to any argument, the server has no choice but to wait for the client request, as shown below on the right.

$$\begin{array}{l} \overbrace{\bar{x}(\lambda a \lambda b a (b \text{ prod})).y}^{\text{CLIENT}} \mid \overbrace{x(\bar{y}.\circ) \text{ cost}}^{\text{SERVER}} \\ \mapsto y \mid (\lambda a \lambda b a (b \text{ prod}))(\bar{y}.\circ) \text{ cost} \\ \mapsto^* y \mid (\bar{y}(\text{cost } \text{prod})).\circ \\ \mapsto^* y \mid \bar{y}\text{price}.\circ \\ \mapsto \text{price} \mid \circ \end{array}$$

By defining $X := (\mathbb{N} \multimap \perp) \multimap (\text{String} \multimap \mathbb{N}) \multimap \perp$, we can type the process above as follows:

$$\begin{array}{c} \frac{x : X \Rightarrow x : X \quad \Rightarrow y : \mathbb{N}, \bar{y}.\circ : \mathbb{N} \multimap \perp}{x : X \Rightarrow y : \mathbb{N}, x(\bar{y}.\circ) : (\text{String} \multimap \mathbb{N}) \multimap \perp} \quad \Rightarrow \text{cost} : \text{String} \multimap \mathbb{N} \\ \frac{x : X \Rightarrow y : \mathbb{N}, x(\bar{y}.\circ) \text{ cost} : \perp}{\Rightarrow \bar{x}.y : X \multimap \mathbb{N}, x(\bar{y}.\circ) \text{ cost} : \perp} \quad \Rightarrow (\lambda a \lambda b a (b \text{ prod})) : X \\ \frac{\Rightarrow \bar{x}(\lambda a \lambda b a (b \text{ prod})).y : \mathbb{N}, x(\bar{y}.\circ) \text{ cost} : \perp}{\Rightarrow \bar{x}(\lambda a \lambda b a (b \text{ prod})).y \mid x(\bar{y}.\circ) \text{ cost} : \mathbb{N} \wp \perp} \end{array}$$

Example 3.2 (Client-Server: Dialogue). We present now a continuation of the previous example which models a longer interaction. We want to represent the following transaction of an online sale. As before, a buyer transmits to a seller a product name *prod* : *String*, the seller computes by the function *cost* the monetary cost *price* : \mathbb{N} of *prod* and communicates it to the buyer. Then the buyer applies a function *pay* : $\mathbb{N} \rightarrow \text{String}$ to *price*, that will transmit to the server the credit card number *card* : *String* if the client wishes to buy the product, the empty string otherwise. By comparison with the previous example, we must add a new communication channel \bar{z} for the third communication, because in λ_{\wp} each communication requires a fresh channel. As before, in order to implement synchronization, messages *M* will be always transmitted as $(\lambda a \lambda b a (b M))$:

$$\overbrace{\bar{x}(\lambda a \lambda b a (b \text{ prod})).y(\bar{z}.\circ) \text{ pay}}^{\text{CLIENT}} \mid \overbrace{x(\bar{y}.z)(\lambda c \lambda a' \lambda b' a' (b' \text{ cost } c))}^{\text{SERVER}}$$

We can observe the following interaction:

$$\begin{aligned}
& \bar{x}(\lambda a \lambda b a (b \text{ prod})).y(\bar{z}. \circ) \text{ pay} \mid x(\bar{y}. z) (\lambda c \lambda a' \lambda b' a' (b' \text{ cost } c)) \\
& \mapsto y(\bar{z}. \circ) \text{ pay} \mid (\lambda a \lambda b a (b \text{ prod}))(\bar{y}. z) (\lambda c \lambda a' \lambda b' a' (b' \text{ cost } c)) \\
& \mapsto^* y(\bar{z}. \circ) \text{ pay} \mid \bar{y}((\lambda c \lambda a' \lambda b' a' (b' \text{ cost } c)) \text{ prod}).z \\
& \mapsto^* y(\bar{z}. \circ) \text{ pay} \mid \bar{y}(\lambda a' \lambda b' a' (b' \text{ price})).z \\
& \mapsto (\lambda a' \lambda b' a' (b' \text{ price}))(\bar{z}. \circ) \text{ pay} \mid z \\
& \mapsto^* \bar{z}(\text{pay price}).\circ \mid z \\
& \mapsto \bar{z} \text{ card}.\circ \mid z \\
& \mapsto \circ \mid \text{card}
\end{aligned}$$

By defining $X := (Y \multimap \text{String}) \multimap (\text{String} \multimap Y) \multimap \text{String}$, $Y := (\text{String} \multimap \perp) \multimap (\mathbb{N} \multimap \text{String}) \multimap \perp$, we can type the process above as follows:

$$\begin{array}{c}
\frac{\frac{\frac{\Rightarrow z : \text{String}, \bar{z}. \circ : \text{String} \multimap \perp \quad y : Y \Rightarrow y : Y}{y : Y \Rightarrow y(\bar{z}. \circ) : (\mathbb{N} \multimap \text{String}) \multimap \perp, z : \text{String}}}{\Rightarrow y(\bar{z}. \circ) : (\mathbb{N} \multimap \text{String}) \multimap \perp, \bar{y}. z : Y \multimap \text{String}} \Rightarrow \text{pay} : \mathbb{N} \multimap \text{String}} \\
\frac{x : X \Rightarrow x : X \quad \Rightarrow y(\bar{z}. \circ) \text{ pay} : \perp, \bar{y}. z : Y \multimap \text{String}}{x : X \Rightarrow y(\bar{z}. \circ) \text{ pay} : \perp, x(\bar{y}. z) : (\text{String} \multimap Y) \multimap \text{String}} \Rightarrow \lambda c \lambda a' \lambda b' a' (b' \text{ cost } c) : \text{String} \multimap Y \\
\frac{x : X \Rightarrow y(\bar{z}. \circ) \text{ pay} : \perp, x(\bar{y}. z)(\lambda c \lambda a' \lambda b' a' (b' \text{ cost } c)) : \text{String}}{\Rightarrow \bar{x}. y(\bar{z}. \circ) \text{ pay} : X \multimap \perp, x(\bar{y}. z)(\lambda c \lambda a' \lambda b' a' (b' \text{ cost } c)) : \text{String}} \Rightarrow \lambda a \lambda b a (b \text{ prod}) : X \\
\frac{\Rightarrow \bar{x}(\lambda a \lambda b a (b \text{ prod})).y(\bar{z}. \circ) \text{ pay} : \perp, x(\bar{y}. z)(\lambda c \lambda a' \lambda b' a' (b' \text{ cost } c)) : \text{String}}{\Rightarrow \bar{x}(\lambda a \lambda b a (b \text{ prod})).y(\bar{z}. \circ) \text{ pay} \mid x(\bar{y}. z)(\lambda c \lambda a' \lambda b' a' (b' \text{ cost } c)) : \perp \wp \text{String}}
\end{array}$$

Example 3.3 (Cyclic Communication). Unlike in the session typed π -calculi of [Caires and Pfenning 2010] and [Wadler 2012], in λ_{\wp} one can type in a natural way cyclic communication patterns, as shown in the following example. A client has some secret information $M : \text{String}$, which it wishes to encrypt. For added security, the client desires two encryptions of the message. A couple of servers, by joining forces, offer this kind of double-encryption service. Therefore, the client sends M to the first server, which encrypts M by applying the function $\text{enc}_1 : \text{String} \multimap \text{String}$ and then transmits the result to the second server, which in turn encrypts it by applying the function $\text{enc}_2 : \text{String} \multimap \text{String}$ and finally transmits the result to the client. The implementation is shown below on the left, and we observe the interaction shown below on the right.

$$\begin{array}{c}
\text{CLIENT} \quad \text{SERVER 1} \quad \text{SERVER 2} \\
\overbrace{\bar{x}M.z} \quad \overbrace{\bar{y}(\text{enc}_1 x).\circ} \quad \overbrace{\bar{z}(\text{enc}_2 y).\circ}
\end{array}$$

$$\begin{aligned}
& \bar{x}M.z \mid \bar{y}(\text{enc}_1 x).\circ \mid \bar{z}(\text{enc}_2 y).\circ \\
& \mapsto z \mid \bar{y}(\text{enc}_1 M).\circ \mid \bar{z}(\text{enc}_2 y).\circ \\
& \mapsto z \mid \bar{y}M'.\circ \mid \bar{z}(\text{enc}_2 y).\circ \\
& \mapsto z \mid \circ \mid \bar{z}(\text{enc}_2 M').\circ \\
& \mapsto z \mid \circ \mid \bar{z}M''.\circ \\
& \mapsto M'' \mid \circ \mid \circ
\end{aligned}$$

We can type the process above as follows:

$$\begin{array}{c}
\frac{\Rightarrow z : \text{String}, \bar{z}. \circ : \text{String} \multimap \perp \quad y : \text{String} \Rightarrow \circ : \perp, \text{enc}_2 y : \text{String}}{\frac{y : \text{String} \Rightarrow z : \text{String}, \circ : \perp, \bar{z}(\text{enc}_2 y). \circ : \perp}{\Rightarrow z : \text{String}, \bar{y}. \circ : \text{String} \multimap \perp, \bar{z}(\text{enc}_2 y). \circ : \perp} \quad x : \text{String} \Rightarrow \text{enc}_1 x : \text{String}}{\frac{x : \text{String} \Rightarrow z : \text{String}, \bar{y}(\text{enc}_1 x). \circ : \perp, \bar{z}(\text{enc}_2 y). \circ : \perp}{\Rightarrow \bar{x}. z : \text{String} \multimap \text{String}, \bar{y}(\text{enc}_1 x). \circ : \perp, \bar{z}(\text{enc}_2 y). \circ : \perp} \Rightarrow M : \text{String}} \\
\frac{\Rightarrow \bar{x}M.z : \text{String}, \bar{y}(\text{enc}_1 x). \circ : \perp, \bar{z}(\text{enc}_2 y). \circ : \perp}{\Rightarrow \bar{x}M.z \mid \bar{y}(\text{enc}_1 x). \circ : \text{String} \wp \perp, \bar{z}(\text{enc}_2 y). \circ : \perp} \\
\Rightarrow \bar{x}M.z \mid \bar{y}(\text{enc}_1 x). \circ \mid \bar{z}(\text{enc}_2 y). \circ : (\text{String} \wp \perp) \wp \perp
\end{array}$$

Example 3.4 (Channel Transmission). Just like π -calculus, λ_{\wp} supports communication of channel names and thus dynamic communication patterns, as we see in the following example. A server offers a printing service, which is hosted on another machine, connected to the server. In order to exploit the service an access code is required. A consumer, which wants to print the string M , sends its access code $\text{access} : \text{String}$ to the server, which checks it by the function check . Upon success, the server transmits to the consumer the channel \bar{z} along which the printer offers its services, so that finally the consumer can send M to the printer. We model printer, server and client as follows:

$$\overbrace{z \text{ print}}^{\text{PRINTER}} \mid \overbrace{(\text{check } x)(\bar{y}. \circ)(\bar{z}. \circ)}^{\text{SERVER}} \mid \overbrace{\bar{x}\text{access}.y (\lambda a a M)}^{\text{CONSUMER}}$$

We observe the following interaction:

$$\begin{aligned}
& z \text{ print} \mid (\text{check } x)(\bar{y}. \circ)(\bar{z}. \circ) \mid \bar{x}\text{access}.y (\lambda a a M) \\
& \mapsto z \text{ print} \mid (\text{check access})(\bar{y}. \circ)(\bar{z}. \circ) \mid y (\lambda a a M) \\
& \mapsto z \text{ print} \mid \bar{y}(\bar{z}. \circ). \circ \mid y (\lambda a a M) \\
& \mapsto z \text{ print} \mid \circ \mid \bar{z}(\lambda a a M). \circ \\
& \mapsto (\lambda a a M) \text{ print} \mid \circ \mid \circ \\
& \mapsto \text{print } M \mid \circ \mid \circ
\end{aligned}$$

By defining $C := ((\text{String} \multimap \perp) \multimap \perp) \multimap \perp$, we can type the above process as follows:

$$\begin{array}{c}
\frac{y : C \Rightarrow \circ : \perp, y (\lambda a a M) : \perp}{x : \text{String} \Rightarrow \text{check } x : C \multimap C \multimap \perp \quad \Rightarrow \bar{y}. \circ : C, y (\lambda a a M) : \perp} \\
\frac{x : \text{String} \Rightarrow (\text{check } x)(\bar{y}. \circ) : C \multimap \perp, y (\lambda a a M) : \perp}{\Rightarrow (\text{check } x)(\bar{y}. \circ) : C \multimap \perp, \bar{x}. y (\lambda a a M) : \text{String} \multimap \perp} \Rightarrow \text{access} : \text{String} \\
\frac{\Rightarrow (\text{check } x)(\bar{y}. \circ) : C \multimap \perp, \bar{x}\text{access}.y (\lambda a a M) : \perp \quad \Rightarrow z \text{ print} : \perp, \bar{z}. \circ : C}{\Rightarrow z \text{ print} : \perp, (\text{check } x)(\bar{y}. \circ)(\bar{z}. \circ) : \perp, \bar{x}\text{access}.y (\lambda a a M) : \perp} \\
\frac{\Rightarrow z \text{ print} \mid (\text{check } x)(\bar{y}. \circ)(\bar{z}. \circ) : \perp \wp \perp, \bar{x}\text{access}.y (\lambda a a M) : \perp}{\Rightarrow z \text{ print} \mid (\text{check } x)(\bar{y}. \circ)(\bar{z}. \circ) \mid \bar{x}\text{access}.y (\lambda a a M) : (\perp \wp \perp) \wp \perp}
\end{array}$$

4 INTERMEZZO: SYNCHRONOUS COMMUNICATION IN λ_{\wp}

In π -calculus the actions of both sending and receiving messages are *synchronous* and *blocking*. They are synchronous, because they require that the sender and the receiver synchronize; they are *blocking*, because the execution of both the sender and the receiver is blocked until the message is actually transmitted. Therefore, on one hand, if there is no receiver listening to the channel, a process can neither transmit its message along the channel nor proceed with its execution. On the other hand, a process which does listen cannot proceed with its execution until it receives the message it is waiting for. In π -calculus, synchronicity is implemented by a construct for sending $\bar{x}m. _$ and a construct for receiving $x(y). _$ which need to be outermost in the connected processes in order to activate the reduction: $\bar{x}m.P \mid x(y).Q \mapsto P \mid Q[m/y]$. The blocking nature of the actions is implemented by forbidding reductions inside P and Q before the actions are executed, that is, by forcing a reduction strategy.

Non-blocking actions have been advocated by various authors (e.g. [Merro and Sangiorgi 2004], [Kokke et al. 2019]) and are necessary for a full correspondence between cut-elimination and process execution. In $\lambda_{\mathcal{N}}$, the fact that sending and receiving are non-blocking is implemented by just removing the input construct $x(y)$. Hence, provided that x occurs in Q , in $\lambda_{\mathcal{N}}$ we have:

$$\bar{x}m.P \mid Q \mapsto P \mid Q[m/x]$$

4.1 The Call-by-Value $\lambda_{\mathcal{N}}$

Although $\lambda_{\mathcal{N}}$ is naturally asynchronous, we can model synchronous and blocking actions also in $\lambda_{\mathcal{N}}$, by defining reduction to automatically follow a call-by-value discipline.

Definition 4.1 (Value). A *value* is any term of one of the following forms: $\bar{x}.t \quad s \mid t$

Definition 4.2 (Call-by-value Contexts). We define a **call-by-value evaluation context** by the following grammar:

$$E ::= [\] \mid Eu \mid VE \mid \bar{x}\bar{y}E \mid \text{close}(E)$$

where V is any value.

To obtain a call-by-value version of $\lambda_{\mathcal{N}}$, we rewrite the main reduction rules of $\lambda_{\mathcal{N}}$ as follows:

$$\begin{aligned} \dots \mid E[(\lambda x u)V] \mid \dots &\mapsto^{cbv} \dots \mid E[u[V/x]] \mid \dots \\ \dots \mid E[\bar{x}V.s] \mid \dots \mid E'[x] \mid \dots &\mapsto^{cbv} \dots \mid E[s] \mid \dots \mid E'[V] \mid \dots \\ \dots \mid E'[x] \mid \dots \mid E[\bar{x}V.s] \mid \dots &\mapsto^{cbv} \dots \mid E'[V] \mid \dots \mid E[s] \mid \dots \end{aligned}$$

where V is a value and E, E' call-by-value evaluation contexts. The reduction for $\bar{x}\bar{y}V$ are analogous. The idea is that call-by-value contexts bring back a notion of order among operations, which can be used to represent the sequential facet of computation. Thus a term $E'[x]$ represents a process whose next operation is a read operation on the channel x , followed by the operations contained in $E'[\]$. Similarly to Wadler's principal-cut-elimination strategy for CP, \mapsto^{cbv} only allows communication on the top level parallel operators.

We can now define in $\lambda_{\mathcal{N}}$ the synchronous input-construct as follows:

$$x(y).u := (\lambda y u) x$$

Indeed, the term $x(y).u$ is always stuck: since u is located under a λ , no reduction can be performed inside u ; since x is not a value, the redex $(\lambda y u) x$ cannot be contracted either. For similar reasons, the term $\bar{x}V.u = (\bar{x}.u)V$ is stuck as well: since V is a value, no reduction can be performed inside V ; since u is located under an output operator, no reduction can be performed inside u either. Thus, the only reduction that could be fired is the transmission of V along the channel x , which however requires a suitable receiver, namely a process whose call-by-value evaluation cannot further proceed: the process requires an input value and waits for it. Indeed, in the following configuration, we can reduce

$$\bar{x}V.u \mid x(y).w = \bar{x}V.u \mid (\lambda y w)x \mapsto^{cbv} u \mid (\lambda y w)V \mapsto^{cbv} u \mid w[V/y]$$

In the configuration

$$\bar{x}V.u \mid (\lambda z z x(y).w)(\lambda k k)$$

however, the process $\bar{x}V.u$ cannot transmit its message V until $(\lambda z z x(y).w)(\lambda k k)$ is ready to receive, which will happen as soon as its head redex will be contracted and the value of $x(y).w$ will be really needed. Thus, after one reduction step, we have

$$\bar{x}V.u \mid (\lambda k k) x(y).w \mapsto^{cbv} u \mid (\lambda k k)((\lambda y w) V) \mapsto^{cbv} u \mid (\lambda k k) w[V/y] \mapsto^{cbv} u \mid w[V/y]$$

5 SOUNDNESS AND COMPLETENESS

We show now that the typing system NMLL of $\lambda_{\mathcal{N}}$ captures exactly classical multiplicative linear logic. Namely, we show that NMLL is equivalent to the sequent calculus MLL for classical multiplicative linear logic. The cut rule and the logical rules of MLL – namely, the rules introducing the logical connectives of linear logic on the left or on the right – are the standard double-sided sequent rules for classical linear logic. The initial sequent $\perp \Rightarrow$ corresponds to the initial sequent $\Rightarrow 1$. We use the former since we do not use an explicit duality operator. The only non-standard rule of NMLL is $\perp E$. It is clearly sound since it captures the neutrality of \perp with respect to \mathcal{N} , and hence to the comma on the right-hand side of sequents.

Table 3. The sequent calculus MLL.

$\frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \perp} \perp$	$\perp \Rightarrow$	$\frac{\Gamma \Rightarrow A, \Delta \quad \Sigma, B \Rightarrow \Theta}{\Gamma, \Sigma, A \multimap B \Rightarrow \Delta, \Theta} \multimap l$	$\frac{\Gamma, A \Rightarrow B, \Delta}{\Gamma \Rightarrow A \multimap B, \Delta} \multimap r$
$\frac{\Gamma, A \Rightarrow \Delta \quad \Sigma, B \Rightarrow \Theta}{\Gamma, \Sigma, A \mathcal{N} B \Rightarrow \Delta, \Theta} \mathcal{N}l$	$\frac{\Gamma \Rightarrow A, B, \Delta}{\Gamma \Rightarrow A \mathcal{N} B, \Delta} \mathcal{N}r$	$\frac{\Gamma \Rightarrow A, \Delta \quad \Sigma, A \Rightarrow \Theta}{\Gamma, \Sigma \Rightarrow \Delta, \Theta} \text{cut}$	

PROPOSITION 5.1 (SOUNDNESS). *If $\Gamma \Rightarrow \Delta$ is derivable in NMLL, then $\Gamma \Rightarrow \Delta$ is derivable in MLL.*

PROOF. By induction on the number of rule applications in the NMLL derivation of $S = \Gamma \Rightarrow \Delta$. \square

The natural deduction calculus NMLL corresponds very neatly to the sequent calculus MLL. Hence, a completeness proof of NMLL with respect to MLL is quite straightforward. The proof deviates from standard reasoning only insofar as we need to handle the occurrences of \perp introduced by $\mathcal{N}E$. Indeed, whenever we apply $\mathcal{N}E$, we introduce an occurrence of \perp to type a term of the form $\bar{x} \bar{y} m$ containing output channels \bar{x} and \bar{y} and a message m . But, in order to obtain a derivation of a specific target sequent $\Gamma \Rightarrow \Delta$, we might need to remove those extra occurrences of \perp . To do so, we use the $\perp E$ rule.

PROPOSITION 5.2 (COMPLETENESS). *If $\Gamma \Rightarrow \Delta$ is derivable in MLL, then $\Gamma \Rightarrow \Delta$ is derivable in NMLL.*

PROOF. By induction on the number of rule applications in the MLL derivation of $S = \Gamma \Rightarrow \Delta$. \square

6 SUBJECT REDUCTION

We are now going to prove that the reductions of $\lambda_{\mathcal{N}}$ preserve the typing of terms, a property well-known under the name of Subject Reduction. We first need the concept of substitution, which in $\lambda_{\mathcal{N}}$ is just a replacement of some variables, with no renaming involved.

Definition 6.1 (Substitution). For any multiset of typed terms Δ and terms v_1, \dots, v_n , we denote by $\Delta[v_1/x_1 \dots v_n/x_n]$ the simultaneous replacement, for any $i \in \{1, \dots, n\}$, of all occurrences of x_i in all terms of Δ by v_i . Given a substitution $[t/x]$, we refer to t as *the substituting term* and to x as *the substituted variable*.

In order to prove Subject Reduction, we first prove that if we have a variable $x : A$ in the context of a derivable sequent $\Gamma, x : A \Rightarrow \Delta$ and we can type a term $t : A$ by deriving a sequent $\Sigma \Rightarrow t : A, \Theta$, then we can derive the sequent $\Gamma, \Sigma \Rightarrow \Delta[t/x], \Theta$ effectively replacing the variable $x : A$ with the

term $t : A$. The proof-theoretical intuition is the following: if A is among the assumptions of a proof, and we have a derivation t of A , then we can use t to derive A directly inside the proof.

LEMMA 6.1 (SUBSTITUTION). *If $\Sigma \Rightarrow t : A, \Theta$ and $\Gamma, x : A \Rightarrow \Delta$ are derivable in NMLL and share no variable, then $\Gamma, \Sigma \Rightarrow \Delta[t/x], \Theta$ is derivable in NMLL as well.*

PROOF. We proceed by induction on the number of rule applications in the derivation of $\Gamma, x : A \Rightarrow \Delta$.

If no rule is applied in the derivation, then $\Gamma, x : A \Rightarrow \Delta$ is of the form $x : A \Rightarrow x : A$. Therefore, $\Gamma, \Sigma \Rightarrow \Delta[t/x], \Theta$ is just $\Sigma \Rightarrow t : A, \Theta$ and the claim trivially holds.

Suppose now that the statement holds for any sequent which is derivable with n or less rule applications, we show the result for any sequent $\Gamma, x : A \Rightarrow \Delta$ which is derivable using $n + 1$ rule applications. We reason by cases on the last rule applied in this derivation of $\Gamma, x : A \Rightarrow \Delta$.

- $\frac{\Gamma, x : A \Rightarrow \Delta'}{\Gamma, x : A \Rightarrow \Delta', \circ : \perp} \perp I$ where $\Delta = \Delta', \circ : \perp$. By inductive hypothesis $\Gamma, \Sigma \Rightarrow \Delta'[t/x], \Theta$ is derivable. By applying

$$\frac{\Gamma, \Sigma \Rightarrow \Delta'[t/x], \Theta}{\Gamma, \Sigma \Rightarrow \Delta'[t/x], \Theta, \circ : \perp} \perp I$$

we obtain a derivation of $\Gamma, \Sigma \Rightarrow \Delta[t/x], \Theta$, which verifies the claim. The rule $\perp E$ is similar.

- $\frac{\Gamma_1 \Rightarrow v : B \multimap C, \Delta_1 \quad \Gamma_2 \Rightarrow w : B, \Delta_2}{\Gamma_1, \Gamma_2 \Rightarrow vw : C, \Delta_1, \Delta_2} \multimap E$ where $\Delta = vw : C, \Delta_1, \Delta_2$. Suppose that $x : A$ is contained in Γ_1 and hence that $\Gamma_1 = \Gamma'_1, x : A$ and that $\Gamma = \Gamma'_1, \Gamma_2$. The case in which $x : A$ is contained in Γ_2 is analogous. By inductive hypothesis $\Gamma'_1, \Sigma \Rightarrow v[t/x] : B \multimap C, \Delta_1[t/x], \Theta$ is derivable. Now, due to the type assignment rules of NMLL and since $x : A$ is contained in Γ_1 , we have that x cannot occur in w . By hypothesis, t and Θ share no variable with Γ_2, w, Δ_2 . Hence, by applying

$$\frac{\Gamma_1, \Sigma \Rightarrow v[t/x] : B \multimap C, \Delta_1[t/x], \Theta \quad \Gamma_2 \Rightarrow w : B, \Delta_2}{\Gamma_1, \Sigma, \Gamma_2 \Rightarrow (v[t/x])w : C, \Delta_1[t/x], \Theta, \Delta_2} \multimap E$$

we obtain a derivation of $\Gamma, \Sigma \Rightarrow (vw)[t/x] : C, \Delta[t/x], \Theta$, which verifies the claim.

- $\frac{\Gamma, y : B \Rightarrow s : C, \Delta'}{\Gamma \Rightarrow \bar{y}. s : B \multimap C, \Delta'} \multimap I$ where $\Delta = \bar{y}. s : B \multimap C, \Delta'$. Since $x : A$ must occur in Γ in the conclusion of the rule application, we know that $x \neq y$. By inductive hypothesis, $\Gamma, y : B, \Sigma \Rightarrow s[t/x] : C, \Delta'[t/x], \Theta$ is derivable. Hence, since by hypothesis y does not occur in t , by applying

$$\frac{\Gamma, y : B, \Sigma \Rightarrow s[t/x] : C, \Delta'[t/x], \Theta}{\Gamma, \Sigma \Rightarrow \bar{y}. (s[t/x]) : B \multimap C, \Delta'[t/x], \Theta} \multimap I$$

we obtain a derivation of $\Gamma, \Sigma \Rightarrow (\bar{y}. s)[t/x] : B \multimap C, \Delta'[t/x], \Theta$, which verifies the claim.

- The cases $\wp E$ and $\wp I$ are similar.

□

The proof of the Subject Reduction is now quite standard.

THEOREM 6.2 (SUBJECT REDUCTION). *Assume there is a NMLL derivation of $\Pi \Rightarrow t_1 : F_1, \dots, t_n : F_n, t_{n+1}, \dots, t_m$. If $t_1 \mid \dots \mid t_m \mapsto t'_1 \mid \dots \mid t'_m$, then there is a derivation of $\Pi \Rightarrow t'_1 : F_1, \dots, t'_n : F_n, t'_{n+1}, \dots, t'_m$.*

PROOF. We prove the statement by induction on the number of rule applications in the derivation of $\Pi \Rightarrow t_1 : F_1, \dots, t_n : F_n, t_{n+1}, \dots, t_m$.

If no rule is applied in the derivation, then $\Pi \Rightarrow t_1 : F_1, \dots, t_n : F_n, t_{n+1}, \dots, t_m$ is of the form $x : A \Rightarrow x : A$. Since no reduction applies to any term in the right-hand side of the sequents, the claim trivially holds.

Suppose now that the statement holds for any NMLL derivation containing m or less rule applications, we show the result for a generic NMLL derivation of $\Pi \Rightarrow t_1 : F_1, \dots, t_n : F_n, t_{n+1}, \dots, t_m$ containing $m + 1$ rule applications. We reason by cases on the last rule applied in this NMLL derivation.

- $\frac{\Gamma \Rightarrow \bar{x}. v : A \multimap B, \Delta \quad \Sigma \Rightarrow w : A, \Theta}{\Gamma, \Sigma \Rightarrow \bar{x}w.v : B, \Delta, \Theta} \multimap E$ where

$$t_1 \mid \dots \mid t_m = t_1 \mid \dots \mid \bar{x}w.v \mid \dots \mid t_m$$

and

$$t'_1 \mid \dots \mid t'_m = (t_1 \mid \dots \mid v \mid \dots \mid t_m)[w/x]$$

since x occurs only once.

- If the last rule applied above $\Gamma \Rightarrow \bar{x}. v : A \multimap B, \Delta$ is

$$\frac{\Gamma, x : A \Rightarrow v : B, \Delta}{\Gamma \Rightarrow \bar{x}. v : A \multimap B, \Delta} \multimap I$$

then, by the restriction on NMLL rule applications, x occurs either in v or in Δ , and by Lemma 6.1 applied to the sequents $\Sigma \Rightarrow w : A, \Theta$ and $\Gamma, x : A \Rightarrow v : B, \Delta$ we obtain that $\Gamma, \Sigma \Rightarrow v[w/x] : B, \Delta[w/x], \Theta$ is derivable and hence that we have a derivation of $\Pi \Rightarrow t'_1 : F_1, \dots, t'_n : F_n, t'_{n+1}, \dots, t'_m$.

- Otherwise, we have

$$\frac{\Phi_1 \Rightarrow \bar{x}. v : A \multimap B, \Psi_1 \quad \dots \quad \Phi_p \Rightarrow \Psi_p}{\Gamma \Rightarrow \bar{x}. v : A \multimap B, \Delta} \rho \quad \frac{\Sigma \Rightarrow w : A, \Theta}{\Gamma, \Sigma \Rightarrow \bar{x}w.v : B, \Delta, \Theta} \multimap E$$

where $\bar{x}. v : A \multimap B$ only occurs in one of the premises of ρ – we display it in the first premise without loss of generality.

If we construct the following derivation

$$\frac{\frac{\Phi_1 \Rightarrow \bar{x}. v : A \multimap B, \Psi_1 \quad \Sigma \Rightarrow w : A, \Theta}{\Phi_1, \Sigma \Rightarrow \bar{x}w.v : B, \Psi_1, \Theta} \multimap E \quad \dots \quad \Phi_p \Rightarrow \Psi_p}{\Gamma, \Sigma \Rightarrow \bar{x}w.v : B, \Delta, \Theta} \rho$$

we know without loss of generality that

$$\bar{x}w.v : B, \Psi_1, \Theta = t_1 : F_1, \dots, t_j : F_j, t_{n+1}, \dots, t_i$$

and that

$$t_1 \mid \dots \mid t_j \mid t_{n+1} \mid \dots \mid t_i \mapsto t'_1 \mid \dots \mid t'_j \mid t'_{n+1} \mid \dots \mid t'_i$$

because by Proposition 7.1, x occurs either in v or in Ψ_1 . By inductive hypothesis, there is a derivation of

$$\begin{aligned} \Phi_1, \Sigma \Rightarrow t'_1 : F_1, \dots, t'_j : F_j, t'_{n+1}, \dots, t'_i \\ = \Phi_1, \Sigma \Rightarrow v[w/x] : B, \Psi_1[w/x], \Theta \end{aligned}$$

By assumption we have

$$\Gamma, \Sigma \Rightarrow \bar{x}w.v : B, \Delta, \Theta = \Pi \Rightarrow t_1 : F_1, \dots, t_n : F_n, t_{n+1}, \dots, t_m$$

Therefore, by inspection of the rules of NMLL, it is easy to see that by the rule application

$$\frac{\Phi_1, \Sigma \Rightarrow v[w/x] : B, \Psi_1[w/x], \Theta \quad \dots \quad \Phi_p \Rightarrow \Psi_p}{\Gamma, \Sigma \Rightarrow v[w/x] : B, \Delta[w/x], \Theta} \rho$$

we obtain a derivation of the sequent

$$\begin{aligned} \Gamma, \Sigma \Rightarrow v[w/x] : B, \Delta[w/x], \Theta &= \Gamma, \Sigma \Rightarrow t'_1 : F_1, \dots, t'_n : F_n, t'_{n+1}, \dots, t'_m \\ \bullet \frac{\Gamma \Rightarrow v \mid w : A \wp B, \Delta \quad \Sigma_1, x : A \Rightarrow \Theta_1 \quad \Sigma_2, y : B \Rightarrow \Theta_2}{\Gamma, \Sigma_1, \Sigma_2 \Rightarrow \bar{x} \bar{y} (v \mid w) : \perp, \Delta, \Theta_1, \Theta_2} \wp E \text{ where} \\ t_1 \mid \dots \mid t_m &= t_1 \mid \dots \mid \bar{x} \bar{y} (v \mid w) \mid \dots \mid t_m \end{aligned}$$

and

$$t'_1 \mid \dots \mid t'_m = (t_1 \mid \dots \mid \circ \mid \dots \mid t_m)[v/x \ w/y]$$

– If the last rule applied above $\Gamma \Rightarrow v \mid w : A \wp B, \Delta$ is

$$\frac{\Gamma \Rightarrow v : A, w : B, \Delta}{\Gamma \Rightarrow v \mid w : A \wp B, \Delta} \wp I$$

By the application of Lemma 6.1 to the sequents $\Gamma \Rightarrow v : A, w : B, \Delta$ and $\Sigma_1, x : A \Rightarrow \Theta_1$ we obtain that $\Gamma, \Sigma_1 \Rightarrow w : B, \Delta, \Theta_1[v/x]$ is derivable. Furthermore, by applying Lemma 6.1 to the sequents $\Gamma, \Sigma_1 \Rightarrow w : B, \Delta, \Theta_1[v/x]$ and $\Sigma_2, y : B \Rightarrow \Theta_2$ we obtain that $\Gamma, \Sigma_1, \Sigma_2 \Rightarrow \Delta, \Theta_1[v/x], \Theta_2[w/y]$. Since x occurs in Θ_1 and y occurs in Θ_2 , we can construct a derivation of $\Pi \Rightarrow t'_1 : F_1, \dots, t'_n : F_n, t'_{n+1}, \dots, t'_m$ by applying $\perp I$.

– Otherwise, we have

$$\frac{\frac{\Phi_1 \Rightarrow v \mid w : A \wp B, \Psi_1 \quad \dots \quad \Phi_p \Rightarrow \Psi_p}{\Gamma \Rightarrow v \mid w : A \wp B, \Delta} \rho \quad \Sigma_1, x : A \Rightarrow \Theta_1 \quad \Sigma_2, y : B \Rightarrow \Theta_2}{\Gamma, \Sigma \Rightarrow \bar{x} \bar{y} (v \mid w) : \perp, \Delta, \Theta_1, \Theta_2} \wp E$$

where $v \mid w : A \wp B$ only occurs in one of the premises of ρ – we display it in the first premise without loss of generality.

If we construct the following derivation

$$\frac{\frac{\Phi_1 \Rightarrow v \mid w : A \wp B, \Psi_1 \quad \Sigma_1, x : A \Rightarrow \Theta_1 \quad \Sigma_2, y : B \Rightarrow \Theta_2}{\Phi_1, \Sigma \Rightarrow \bar{x} \bar{y} (v \mid w) : \perp, \Psi_1, \Theta_1, \Theta_2} \wp E \quad \dots \quad \Phi_p \Rightarrow \Psi_p}{\Gamma, \Sigma \Rightarrow \bar{x} \bar{y} (v \mid w) : \perp, \Delta, \Theta_1, \Theta_2} \rho$$

we know without loss of generality that

$$\bar{x} \bar{y} (v \mid w) : \perp, \Psi_1, \Theta_1, \Theta_2 = t_1 : F_1, \dots, t_j : F_j, t_{n+1}, \dots, t_i$$

and that

$$t_1 \mid \dots \mid t_j \mid t_{n+1} \mid \dots \mid t_i \mapsto t'_1 \mid \dots \mid t'_j \mid t'_{n+1} \mid \dots \mid t'_i$$

because x and y occur in Θ_1 and Θ_2 respectively. By inductive hypothesis, there is a derivation of

$$\Phi_1, \Sigma \Rightarrow t'_1 : F_1, \dots, t'_j : F_j, t'_{n+1}, \dots, t'_i = \Phi_1, \Sigma \Rightarrow \circ : \perp, \Psi_1, \Theta_1[v/x], \Theta_2[w/y]$$

By assumption

$$\Gamma, \Sigma \Rightarrow \bar{x} \bar{y} (v \mid w) : \perp, \Delta, \Theta_1, \Theta_2 = \Pi \Rightarrow t_1 : F_1, \dots, t_n : F_n, t_{n+1}, \dots, t_m$$

Therefore, by the rule application

$$\frac{\Phi_1, \Sigma \Rightarrow \circ : \perp, \Psi_1, \Theta_1[v/x], \Theta_2[w/y] \quad \dots \quad \Phi_p \Rightarrow \Psi_p}{\Gamma, \Sigma \Rightarrow \circ : \perp, \Delta, \Theta_1[v/x], \Theta_2[w/y]} \rho$$

we have a derivation of the sequent

$$\Gamma, \Sigma \Rightarrow \circ : \perp, \Delta, \Theta_1[v/x], \Theta_2[w/y] = \Gamma, \Sigma \Rightarrow t'_1 : F_1, \dots, t'_n : F_n, t'_{n+1}, \dots, t'_m$$

- In all other cases, the term $\bar{x}w.v$ (or $\bar{x}\bar{y}v$) that triggered the reduction $t_1 \mid \dots \mid t_m \mapsto t'_1 \mid \dots \mid t'_m$ already occurs in the premises of the last rule ρ applied in the NMLL derivation of $\Pi \Rightarrow t_1 : F_1, \dots, t_n : F_n, t_{n+1}, \dots, t_m$.

According to the typing rules, no variable can occur in different premises of the same rule application. Therefore, both the term $\bar{x}w.v$ (respectively $\bar{y}\bar{z}v$) and the variable x (respectively y and z) occur inside the same premise of the last rule application

$$\frac{\Gamma \Rightarrow u_1 : G_1, \dots, u_p : G_p \quad \dots \quad \Phi \Rightarrow \Psi}{\Sigma \Rightarrow \Delta, C_1[u_1] : F_j, \dots, C_p[u_p] : F_n} \rho$$

in the derivation of

$$\Sigma \Rightarrow \Delta, C_1[u_1] : F_j, \dots, C_p[u_p] : F_n = \Pi \Rightarrow t_1 : F_1, \dots, t_n : F_n, t_{n+1}, \dots, t_m$$

Without loss of generality, we assume that x (or y and z) occurs in the premise

$$\Gamma \Rightarrow u_1 : G_1, \dots, u_p : G_p$$

By reducing the redex triggered by $\bar{x}w.v$ or $\bar{x}\bar{y}v$ in the term $u_1 \mid \dots \mid u_p$ we have $u_1 \mid \dots \mid u_p \mapsto u'_1 \mid \dots \mid u'_p$. By inductive hypothesis, there is a derivation of $\Gamma \Rightarrow u'_1 : G_1, \dots, u'_p : G_p$. Now, by inspection of NMLL typing rules, we can easily see that $t_1 \mid \dots \mid t_m$ is of the form $\mathcal{D}[C_1[u_1] \mid \dots \mid C_p[u_p]]$ and by assumption

$$t'_1 \mid \dots \mid t'_m = \mathcal{D}[C_1[u'_1] \mid \dots \mid C_p[u'_p]]$$

Therefore, by inspection of the rules of NMLL, it is easy to see that by the application of ρ

$$\frac{\Gamma \Rightarrow u'_1 : G_1, \dots, u'_p : G_p \quad \dots \quad \Phi \Rightarrow \Psi}{\Sigma \Rightarrow \Delta, C_1[u'_1] : F_j, \dots, C_p[u'_p] : F_n} \rho$$

to the root of the derivation of $\Gamma \Rightarrow u'_1 : G_1, \dots, u'_p : G_p$, we obtain a derivation of

$$\Sigma \Rightarrow \Delta, C_1[u'_1] : F_j, \dots, C_p[u'_p] : F_n = \Pi \Rightarrow t'_1 : F_1, \dots, t'_n : F_n, t'_{n+1}, \dots, t'_m$$

□

7 PROGRESS

We show now that $\lambda_{\mathcal{N}}$ is deadlock-free: if a process contains a potential communication, then the term is not normal and the communication will be carried out. In $\lambda_{\mathcal{N}}$, potential communications are represented by subterms of the form $\bar{x}v.w$ or $\bar{x}\bar{y}v$. Indeed, the presence of those subterms in a process means that the process can use a communication channel x – or a pair of channels x and y – to transmit a message v . Thus we need to show that if such a subterm occurs in a term, there is also a receiver and the term is not normal. We start by showing some properties of the distribution of variables inside NMLL sequents and inside typed $\lambda_{\mathcal{N}}$ -terms. The crucial property is that an output communication channel \bar{x} can only occur if there is a corresponding input channel x .

PROPOSITION 7.1 (LINEARITY OF CHANNELS). *Assume $\Gamma \Rightarrow \Delta$ is derivable in NMLL and let x be any variable. Then:*

- if x occurs in Γ , x occurs exactly once in Δ and \bar{x} does not occur in Δ ;
- if \bar{x} does not occur in Γ but occurs in Δ , x occurs exactly once as x and exactly once as \bar{x} in Δ .

PROOF. By straightforward induction on the length of the derivation of $\Gamma \Rightarrow \Delta$. □

We recall that by Definition 2.4 a simple context $C[]$ is a process which is not a direct parallel composition of simpler processes. A crucial property of simple contexts that we are going to obtain now is the following: if $C[\bar{z}u.v]$ is typable and $C[]$ is simple, then the variable z cannot occur in $C[]$. If such a configuration were possible, we might type $\lambda_{\mathcal{N}}$ -terms like $z(\bar{z}u.v)$ and we would have to choose between allowing components of the same process reduction to communicate $z(\bar{z}u.v) \mapsto uv$ or tolerating a deadlock. Fortunately, the next proposition rules out this scenery.

PROPOSITION 7.2. *Suppose $\Phi \Rightarrow C[u] : F, \Pi$ or $\Phi \Rightarrow C[u], \Pi$ is derivable in NMLL. Then, if $C[]$ is simple and the variable \bar{z} occurs in u , then z does not occur in $C[]$.*

PROOF. We prove the statement by induction on the number of rule applications in the derivation of $\Phi \Rightarrow C[u] : F, \Pi$ or $\Phi \Rightarrow C[u], \Pi$.

If $C[]$ is empty, we are done.

If no rule is applied in the derivation, then $\Phi \Rightarrow C[u] : F, \Pi$ or $\Phi \Rightarrow C[u], \Pi$ is of the form $x : A \Rightarrow x : A$, thus $u = x$ and $C[]$ is empty, therefore the claim trivially holds.

Suppose now that the statement holds for any NMLL derivation containing m or less rule applications, we show the result for a generic NMLL derivation containing $m + 1$ rule applications. We reason by cases on the last rule applied in this NMLL derivation. We may assume that the term $C[u]$ does not occur in any premise of this last rule, otherwise we just apply the inductive hypothesis to any premise containing $C[u]$ and obtain the thesis.

- $\frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, \circ : \perp} \perp I$. Since $C[u]$ occurs in Δ , this case is ruled out by our assumption that $C[u]$ does not occur in $\Gamma \Rightarrow \Delta$.
- $\frac{\Gamma \Rightarrow \Delta, t : \perp}{\Gamma \Rightarrow \Delta, \text{close}(t)} \perp E$. By our assumption, it must be the case that $C[u] = \text{close}(t)$. If $C[] = \text{close}([])$, surely z does not occur in $C[]$. If $C[] = \text{close}(\mathcal{D}[])$, with $t = \mathcal{D}[u]$, then by inductive hypothesis, z does not occur in $\mathcal{D}[]$, thus it does not occur in $C[] = \text{close}(\mathcal{D}[])$.
- $\frac{\Gamma \Rightarrow t : A \multimap B, \Delta \quad \Sigma \Rightarrow s : A, \Theta}{\Gamma, \Sigma \Rightarrow ts : B, \Delta, \Theta} \multimap E$. By our assumption, it must be the case that $C[u] = ts$. Therefore, $t = \mathcal{D}[u]$ or $s = \mathcal{D}[u]$, with respectively $C[] = \mathcal{D}[]$ and $C[] = t \mathcal{D}[]$. By inductive hypothesis, z does not occur in $\mathcal{D}[]$. Moreover, since the premises of the typing rule share no variable, z cannot occur respectively in s and t . Therefore, z does not occur in $C[]$.
- $\frac{\Gamma, x : A \Rightarrow t : B, \Delta}{\Gamma \Rightarrow \bar{x}. t : A \multimap B, \Delta} \multimap I$. By our assumption, it must be the case that $C[u] = \bar{x}. t$. Therefore, $t = \mathcal{D}[u]$ and $C[] = \bar{x}. \mathcal{D}[]$. By inductive hypothesis, z does not occur in $\mathcal{D}[]$. Moreover, $z \neq x$, otherwise \bar{z} would occur twice in $\bar{x}. t$, contradicting Proposition 7.1. Therefore, z does not occur in $C[] = \bar{x}. \mathcal{D}[]$.
- $\frac{\Gamma \Rightarrow t : A \wp B, \Delta \quad \Sigma_1, x : A \Rightarrow \Theta_1 \quad \Sigma_2, y : B \Rightarrow \Theta_2}{\Gamma, \Sigma_1, \Sigma_2 \Rightarrow \bar{x} \bar{y} t : \perp, \Delta, \Theta_1, \Theta_2} \wp E$. By our assumption, it must be the case that $C[u] = \bar{x} \bar{y} t$. Therefore, $t = \mathcal{D}[u]$ and $C[] = \bar{x} \bar{y} \mathcal{D}[]$. By inductive hypothesis, z does not occur in $\mathcal{D}[]$. Moreover, $z \neq x, y$, otherwise \bar{z} would occur twice in $\bar{x} \bar{y} t$, contradicting Proposition 7.1. Therefore z does not occur in $C[] = \bar{x} \bar{y} \mathcal{D}[]$.
- $\frac{\Gamma \Rightarrow t : A, s : B, \Delta}{\Gamma \Rightarrow t \mid s : A \wp B, \Delta} \wp I$. We claim that this case is not possible. Indeed, if it were, by our assumption we would have $C[u] = t \mid s$. Therefore $t = \mathcal{D}[u]$ or $s = \mathcal{D}[u]$, with respectively $C[] = \mathcal{D}[] \mid s$ and $C[] = t \mid \mathcal{D}[]$. Since by Definition 2.4 the context $C[]$ would not be simple, we would have a contradiction.

□

It is now easy to show that if a communication is possible, it will be carried out: if a typable $\lambda_{\mathcal{N}}$ -term contains output channels ready to transmit messages, then it can always perform the communication, because there is always a suitable receiver.

THEOREM 7.3 (PROGRESS).

- (1) *Suppose $\Gamma \Rightarrow C[\bar{x}v.u] : F, t_1 : F_1, \dots, t_n : F_n, t_{n+1}, \dots, t_m$ is derivable in NMLL. Then the term $C[\bar{x}v.u] \mid t_1 \mid \dots \mid t_m$ is not normal.*
- (2) *Suppose $\Gamma \Rightarrow C[\bar{x}\bar{y}u] : F, t_1 : F_1, \dots, t_n : F_n, t_{n+1}, \dots, t_m$ is derivable in NMLL. Then the term $C[\bar{x}\bar{y}u] \mid t_1 \mid \dots \mid t_m$ is not normal.*

PROOF. 1. By Proposition 7.1, \bar{x} and x must occur exactly once in the term $C[\bar{x}v.u] \mid t_1 \mid \dots \mid t_m$. If x occurs in some t_i , then a reduction is possible and we are done. We assume therefore that x occurs in $C[\bar{x}v.u]$. We also assume $C[\]$ is empty, otherwise x occurs in u , hence $\bar{x}v.u = (\lambda x u)v$, which is not normal and we are done.

We claim now that $C[\bar{x}v.u]$ has a subterm of the form $\mathcal{E}[\bar{x}v.u] \mid \mathcal{D}[x]$ or $\mathcal{D}[x] \mid \mathcal{E}[\bar{x}v.u]$, which implies a communication reduction is possible, allowing us to conclude the proof. We prove our claim by induction on $C[\bar{x}v.u]$.

If $C[\bar{x}v.u] = \bar{z}.C'[\bar{x}v.u]$ or $C[\bar{x}v.u] = \bar{y}\bar{z}.C'[\bar{x}v.u]$ or $C[\bar{x}v.u] = \text{close}(C'[\bar{x}v.u])$, then we apply the induction hypothesis on $C'[\bar{x}v.u]$, and we are done.

If $C[\bar{x}v.u] = w.C'[\bar{x}v.u]$ or $C[\bar{x}v.u] = C'[\bar{x}v.u]w$, then by Proposition 7.2, x must occur in $C'[\bar{x}v.u]$, hence we apply the induction hypothesis on $C'[\bar{x}v.u]$, and we are done.

If $C[\bar{x}v.u] = C'[\bar{x}v.u] \mid w$ or $C[\bar{x}v.u] = w \mid C'[\bar{x}v.u]$, then if x occurs in w , we are done. Otherwise, we apply the induction hypothesis on $C'[\bar{x}v.u]$, and concluding the proof of the claim.

2. By Proposition 7.1, \bar{x} and x must occur both exactly once in the term $C[\bar{x}\bar{y}u] \mid t_1 \mid \dots \mid t_m$ and so do \bar{y} and y . By inspection of the inference rules of NMLL it is easy to see that x, y cannot occur in u , therefore they occur in some t_i . Thus, a reduction is possible. \square

8 THE SUBFORMULA PROPERTY

We show in this section that each normal $\lambda_{\mathcal{N}}$ -term corresponds to an analytic derivation: the type of each subterm of any normal $\lambda_{\mathcal{N}}$ -term t is either a subformula of the type of t or a subformula of the types of the variables of t . This property guarantees that, proof-theoretically, the reduction rules of $\lambda_{\mathcal{N}}$ give rise to a complete detour removal procedure, hence we can conclude they are also satisfactory from a logical point of view. Indeed, without this property it is not possible to relate normalization and cut-elimination, as a normal proof without the subformula property cannot be translated in to a cut-free proof. NMLL can then be considered as a well-behaved alternative to sequent calculus and proof-nets for multiplicative linear logic.

We start by defining what it means for a derivation to be in normal form.

Definition 8.1 (Normal form). An NMLL derivation of a sequent $\Gamma \Rightarrow t_1 : T_1, \dots, t_n : T_n, t_{n+1}, \dots, t_m$ is in normal form if the term $t_1 \mid \dots \mid t_m$ is in normal form.

We then recall the notion of stack [Krivine 2009]. A stack represents, from a logical perspective, a series of elimination rules, from a computational perspective, a series of tasks to be carried out.

Definition 8.2 (Stack). A *stack* is a possibly empty sequence $\sigma = \xi_1\xi_2 \dots \xi_n$ such that for every $1 \leq i \leq n$, $\xi_i = t$, with t term of NMLL. If t is a proof term, $t\sigma$ denotes as usual the term $((t\xi_1)\xi_2) \dots \xi_n$.

Before proving the Subformula Property, we need to establish two auxiliary results concerning the shape of terms. The first of these two results guarantees that the expected connection between the shape of a term with its type holds.

PROPOSITION 8.1 (TYPE COHERENCE). *If $\Gamma \Rightarrow t : F, \Delta$ is derivable in NMLL and t is a value, then either $t = \bar{x}.t$ and $F = A \multimap B$ or $t = s \mid t$ and $F = A \wp B$, for some types A and B .*

PROOF. By induction on the number of rule applications in the derivation of $\Gamma \Rightarrow t : F, \Delta$. \square

The second auxiliary result establishes that any normal λ_{\wp} -term – the type of which is not \perp – is either a value or a sequence of operations that cannot be carried out.

LEMMA 8.2 (THE SHAPE OF NORMAL TERMS). *If there is a derivation in NMLL of $\Gamma \Rightarrow u : F, \Pi$, where u is in normal form, is not a value and $F \neq \perp$, then $u = y\sigma$.*

PROOF. By induction on the number of rule applications in the derivation of $\Gamma \Rightarrow u : F, \Pi$. \square

We can now show that normal λ_{\wp} -terms satisfy the Subformula Property. The proof is by induction on the size of the NMLL derivation and, as usual, the difficult case is the one involving implication elimination. Even though we handle this case by a standard argument on the type of sequences of eliminations, represented here by stacks, the multiple-conclusion setting forces us to integrate this argument in the main induction. We do so by using a stronger inductive statement that carries along the induction the required statement about the type of stacks.

THEOREM 8.3 (SUBFORMULA PROPERTY). *Consider any normal NMLL derivation \mathcal{P} of the sequent $x_1 : X_1, \dots, x_m : X_m \Rightarrow t_1 : T_1, \dots, t_n : T_n, \Pi$, where Π does not contain any type. Then every type S occurring in \mathcal{P} is a subformula of some type T_1, \dots, T_n or X_1, \dots, X_m or \perp .*

PROOF. We prove a stronger statement:

Consider any normal NMLL derivation \mathcal{P} of the sequent $x_1 : X_1, \dots, x_m : X_m \Rightarrow t_1 : T_1, \dots, t_n : T_n, \Pi$, where Π does not contain any type. Then every type S occurring in \mathcal{P} is a subformula of some type T_1, \dots, T_n or X_1, \dots, X_m or \perp . Moreover, if a term t_i with $i \in \{1, \dots, n\}$ is of the form $y\sigma$ where σ is a stack, then T_i is a subformula of some type $T_1, \dots, T_{i-1}, T_{i+1}, \dots, T_n$ or X_1, \dots, X_m or \perp .

The proof is by induction on the number of rule applications in \mathcal{P} . \square

9 STRONG NORMALIZATION

It is quite immediate to see that all terms of the calculus strongly normalize. Indeed, each reduction step strictly decreases the size of the term. This is due to the linear nature of terms (Proposition 7.1): each communication or λ -reduction moves one or two terms from one location to another. Therefore, no duplication is involved in the reductions. Since, moreover, each reduction removes one binder from the term, we have that the size of the term strictly decreases. We formally show this in the following theorem.

Definition 9.1 (Communication-size of a term). For any term t , its *communication size* $cs(t)$ is the number of occurrences of \bar{x} . and $\bar{x}\bar{y}$ in t .

THEOREM 9.1 (STRONG NORMALIZATION). *For any term t such that $\Gamma \Rightarrow t : F$ is derivable in NMLL, all sequences of terms t_1, t_2, \dots such that $t_1 = t$ and $t_i \mapsto t_{i+1}$ are finite and contain exactly $cs(t)$ terms.*

PROOF. The proof is by induction on the communication-size $cs(t)$ of t . \square

10 CONFLUENCE

We show now that each λ_{\wp} -term has a unique normal form. Since all λ_{\wp} -terms have a normal form, this implies that the calculus λ_{\wp} is confluent: for any term t , if $t \mapsto t'$ and $t \mapsto t''$, then there is a term t^* such that both $t' \mapsto t^*$ and $t'' \mapsto t^*$.

In order to simplify the proof of confluence, we define the concept of activator. Intuitively, the activator of a reduction is the term which is responsible for the reduction. For instance, to trigger a communication we only need a subterm which is ready to send a message, such as $\bar{x}w.v$. If we do trigger this communication of w through x , the term $\bar{x}w.v$ is going to be its activator.

Definition 10.1 (Activator). Given any reduction $s \mapsto t$, we say that the *activator* of $s \mapsto t$ is the subterm of s of the form $(\lambda x v)w, \bar{x}w.v, \bar{y}z.v$, which is displayed in the corresponding reduction.

THEOREM 10.1 (UNIQUENESS OF THE NORMAL FORM). *For any term t such that the sequent $\Gamma \Rightarrow t : F$ is derivable in NMLL, t has only one normal form.*

PROOF. The proof is by induction on the length of the normalization of t .

If $\text{cs}(t) = 0$, the claim trivially holds. We show the claim for a generic term t such that $\text{cs}(t) = m+1$, under the assumption that the claim holds for all terms with communication-size m or less.

Suppose that the term t reduces to two different terms t' and t'' if we reduce two different redexes in t . We show that both t' and t'' can reduce to the same term t^* . Since, by inductive hypothesis, t' has a unique normal form and t'' has a unique normal form as well, this is enough to show that t has a unique normal form. Since the argument for the $\bar{\lambda}$ -reductions is everywhere close to identical to the argument for \rightarrow -reductions, we only present the latter. Let us denote then the reduction $t \mapsto t'$ as

$$u_1 \mid \dots \mid C[\bar{x}w.v] \mid \dots \mid u_n \mapsto (u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[w/x]$$

There are two main cases.

1. If one of the activators of $t \mapsto t'$ and $t \mapsto t''$ is a subterm of the other, we consider the outermost activator. Without loss of generality, we assume that the outermost is the activator of $t \mapsto t'$. Now, the activator $\bar{y}r.s$ of $t \mapsto t''$ is either a subterm of v or of w .

- The activator of $t \mapsto t''$ is a subterm of v . The reduction $t \mapsto t''$ is then of the form

$$u_1 \mid \dots \mid C[\bar{x}w.v] \mid \dots \mid u_n \mapsto (u_1 \mid \dots \mid C[\bar{x}w.v'] \mid \dots \mid u_n)[r/y]$$

where v' is obtained from v by replacing $\bar{y}r.s$ with s . Since, by Proposition 7.2, y does not occur in w , we have $t'' \mapsto t^*$, with

$$t^* := ((u_1 \mid \dots \mid C[v'] \mid \dots \mid u_n)[r/y])[w/x]$$

We just need to show that $t' \mapsto t^*$, that is

$$(u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[w/x] \mapsto ((u_1 \mid \dots \mid C[v'] \mid \dots \mid u_n)[r/y])[w/x]$$

By Proposition 7.1 we have that $y \neq x$. Now, if r contains x ,

$$(u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[w/x] = (u_1 \mid \dots \mid C[v[w/x]] \mid \dots \mid u_n) \mapsto (u_1 \mid \dots \mid C[v'] \mid \dots \mid u_n)[r[w/x]/y] = t^*$$

If r does not contain x and since y does not occur in w , we have

$$(u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[w/x] \mapsto (u_1 \mid \dots \mid C[v'] \mid \dots \mid u_n)[w/x][r/y] = t^*$$

- The activator of $t \mapsto t''$ is a subterm of w . The reduction $t \mapsto t''$ is then of the form

$$u_1 \mid \dots \mid C[\bar{x}w.v] \mid \dots \mid u_n \mapsto (u_1 \mid \dots \mid C[(\bar{x}w'.v)] \mid \dots \mid u_n)[r/y]$$

where w' is obtained from w by replacing $\bar{y}r.s$ with s . We have $t'' \mapsto t^*$, with

$$t^* := ((u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[r/y])[w'[r/y]/x]$$

We just need to show that $t' \mapsto t^*$, that is

$$(u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[w/x] \mapsto ((u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[r/y])[w'[r/y]/x]$$

We have the following reduction:

$$(u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[w/x] \mapsto ((u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[w'/x])[r/y]$$

Now, r does not contain x , by Proposition 7.2 and because the activator of $t \mapsto t''$ occurs in w . Since, moreover, $y \neq x$, we also have

$$((u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[w'/x])[r/y] = ((u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[r/y])[w'[r/y]/x]$$

and therefore that $t' \mapsto t^*$.

2. If neither of the activators of $t \mapsto t'$ and $t \mapsto t''$ is a subterm of the other, the reduction $t \mapsto t''$ is of the form

$$u_1 \mid \dots \mid C[\bar{x}w.v] \mid \dots \mid u_n \mapsto (u'_1 \mid \dots \mid C'[\bar{x}w.v] \mid \dots \mid u'_n)[r/y]$$

where u'_i and $C'[\]$ are either equal to, respectively, u_i and $C[\]$ or obtained from u_i and $C[\]$, respectively, by replacing the activator $\bar{y}r.s$ of $t \mapsto t''$ with s . We have $t'' \mapsto t^*$, with

$$t^* := ((u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n)[r/y])[w[r/y]/x]$$

We just need to show that $t' \mapsto t^*$, that is

$$(u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[w/x] \mapsto ((u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n)[r/y])[w[r/y]/x]$$

Now, if s is the activator of a reduction, also $s[w/x]$ has the suitable shape to be one; and the relative reductions substitute the same variable. By Definition 2.3, moreover, $y \neq x$ and hence y occurs in the term $(u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[w/x]$, and we can reduce it as follows:

$$(u_1 \mid \dots \mid C[v] \mid \dots \mid u_n)[w/x] \mapsto ((u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n)[w/x])[r[w/x]/y]$$

We must now consider several cases, depending upon x and y occur in the messages w and r .

- If x occurs in r , but y does not occur in w , then, by Proposition 7.1, x does not occur in $u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n$ and we obtain indeed

$$\begin{aligned} & ((u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n)[w/x])[r[w/x]/y] \\ &= (u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n)[r[w/x]/y] = (u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n)[r/y][w/x] = t^* \end{aligned}$$

- If x does not occur in r , but y occurs in w , then, by Proposition 7.1, y does not occur in $u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n$ and we obtain indeed

$$\begin{aligned} & ((u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n)[w/x])[r[w/x]/y] \\ &= (u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n)[w/x][r/y] = (u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n)[w[r/y]/x] = t^* \end{aligned}$$

- If neither x occurs in r nor y occurs in w , then, by Proposition 7.1, y does not occur in $u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n$ and we obtain indeed

$$\begin{aligned} & ((u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n)[w/x])[r[w/x]/y] \\ &= (u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n)[w/x][r/y] = (u'_1 \mid \dots \mid C'[v] \mid \dots \mid u'_n)[r/y][w/x] = t^* \end{aligned}$$

- Finally, we show it cannot be that both x occurs in r and y occurs in w . Indeed, suppose, for the sake of contradiction, that they do. By letting the term $\bar{y}r.s$ transmit its message r , we would have

$$t = u_1 \mid \dots \mid C[\bar{x}w.v] \mid \dots \mid u_n \mapsto u'_1 \mid \dots \mid C'[\bar{x}w[r/y].v] \mid \dots \mid u'_n := t'''$$

By Thm. 6.2, $\Gamma \Rightarrow t''' : F$, which is in contradiction with Prop. 7.2, since x occurs in $w[r/y]$. \square

11 RELATED WORK AND CONCLUSIONS

$\lambda_{\mathcal{S}}$ and Linear Session Typed π -Calculi. An established way to interpret linear logic proofs into concurrent programs is by interpreting sequent calculi for linear logic into π -calculi with session types, see [Caires and Pfenning 2010; Kokke et al. 2019; Wadler 2012]. In those calculi, a session type is a logical expression containing information about the whole sequence of exchanges that occur between two processes along one channel. Session types are attached only to communication channels, therefore they only describe the channels' input/output behaviour, not the processes using them. In $\lambda_{\mathcal{S}}$, as in the tradition of the Curry-Howard correspondence, types are instead attached to processes, are read as process specifications and are employed to formally guarantee that processes will behave according to the specifications expressed by their type. In other words, while the goal of session types is to formally describe process interactions, hence the *dynamics* of communication, the types of $\lambda_{\mathcal{S}}$ formally describe the *result* of the computation. Session types focus on how computation proceeds, the types of $\lambda_{\mathcal{S}}$ focus on what computation accomplishes. Nevertheless, since the nature of π -calculus processes is determined by their channels' behaviour, specifying channels means also specifying processes, hence the difference with $\lambda_{\mathcal{S}}$ is not as significant as it may appear.

A channel typed by a session type will occur in general more than once and will support several transmissions. By contrast, in $\lambda_{\mathcal{S}}$ each output channel can only occur once and thus be used once. As our examples show, two processes of $\lambda_{\mathcal{S}}$ can nevertheless be connected by several different channels. Therefore, in $\lambda_{\mathcal{S}}$ there is no restriction on the number and direction of the communications between any two processes: information can be exchanged back and forth with no limitation.

Unlike in π -calculus, there is no need in $\lambda_{\mathcal{S}}$ of a construct νx for declaring a communication channel private. Since any given channel connects exactly two processes, it already is a private link between the processes, simply because there is no other process sharing the same channel.

The typing system of $\lambda_{\mathcal{S}}$ supports also cyclically interconnected processes, therefore $\lambda_{\mathcal{S}}$ cannot be faithfully translated in the linearly session typed π -calculi [Caires and Pfenning 2010; Kokke et al. 2019; Wadler 2012]. At the time of this writing, we do not know any translation of the multiplicative fragments of those calculi in $\lambda_{\mathcal{S}}$, but also we are aware of no reason a translation should not be possible. Indeed, a referee asked whether the equivalence between the natural deduction NMLL and the sequent calculus MLL induces a translation of CP into $\lambda_{\mathcal{S}}$. In order to derive in $\lambda_{\mathcal{S}}$ the CP cut-rule, based on the axiom $A \wp A^\perp$, where A^\perp is the involutive negation of A , one needs to prove $A \wp A^\perp$ in NMLL. The resulting proof requires several instances of the exclude middle $B \wp (B \multimap \perp)$, with different subformulas B of A , and thus introduces several different communication channels. This indicates that the behavior of a session type introduced in CP by the cut-rule might be simulated by several individual channels of $\lambda_{\mathcal{S}}$ – a technique known in the literature [Kobayashi et al. 1999] as *linearization*. A practical implementation of this idea is not trivial though, because in order to express the dual \bar{A} of A we need \otimes , which is not a primitive connective in $\lambda_{\mathcal{S}}$. It may be better to build a direct translation of CP into $\lambda_{\mathcal{S}}$, drawing inspiration from the translation of session types into linear types formally studied in [Dardha et al. 2012].

The work in [Paykin and Zdancewic 2016] may help out in this direction, because it presents a translation between a version of CP (CP^\pm) and a linear version of the calculus λ_μ typed by classical logic. A translation between CP^\pm and $\lambda_{\mathcal{S}}$ is not straightforward; obstacles like circular communication patterns generate non-trivial technical problems. Such a translation might be obtained if a linear version of λ_μ could simulate $\lambda_{\mathcal{S}}$. But even considering the result in [Paykin and Zdancewic 2016], it is not obvious how to translate $\lambda_{\mathcal{S}}$ into linear λ_μ .

To summarize, the main advantages of $\lambda_{\mathcal{S}}$ with respect to session typed π -calculi are: λ -calculus offers natively all the power of functional programming; process networks are more flexible, as processes need not always come in pairs; code mobility offers the possibility of transmitting directly

code, which is sometimes less expensive than transmitting a reference and letting the receiver and the referenced code exchange information back and forth. Some disadvantages: one single channel cannot represent a logically uniform sequence of message exchanges and types do not specify communication protocols, unlike session types; the sequentiality of input-output actions must be coded in CPS style, unlike in π -calculus, where it is represented by a more linear syntax.

$\lambda_{\mathcal{N}}$ and Typed Concurrent λ -Calculi. As consequence of adopting linear sequent calculus, the calculi of [Caires and Pfenning 2010; Kokke et al. 2019; Wadler 2012] do not directly support functional computation. Some approaches have been developed to overcome this intrinsic limitation.

The system in [Toninho et al. 2013] provides some combination of session typed processes and functional programs. However, the typing system for the functional part is just added on top of the linear logic system. Therefore, there is no seamless account of both functional and concurrent computation by a *single system* for linear logic, which is the main accomplishment of our work.

The linear concurrent functional calculus GV in [Wadler 2012] is more similar in spirit to $\lambda_{\mathcal{N}}$. However, when it comes to linear logic, it displays no connection between computation and the full normalization process leading to analytic proofs. No direct reduction rules for GV are provided either. [Lindley and Morris 2015] introduce a new version GV' of GV, endow it with an operational semantics and relate it to linear logic through CP. There are several differences between $\lambda_{\mathcal{N}}$ and GV'. First, GV' is not based on a direct isomorphism with a logical system: several constants, like send and receive, are not typed by logical tautologies, and parallel compositions have no type. Second, each process is just a parallel composition of simply typed λ -terms, while in $\lambda_{\mathcal{N}}$ any two processes can be composed. Hence, in GV' only terms of λ -calculus, and not complex processes, can be transmitted as messages, unlike in $\lambda_{\mathcal{N}}$. Nevertheless, the typed λ -calculus of GV' includes the fork operation, and thus it is possible to transmit terms that will eventually become parallel compositions of processes in the context of the receiver. Third, GV' is limited to a call-by-value strategy while $\lambda_{\mathcal{N}}$ reduction is not restricted to any particular strategy and reflects full cut-elimination.

The concurrent λ -calculi in [Aschieri et al. 2017, 2018] are typed by Gödel and classical logic. They are based on extensions of intuitionistic natural deduction, thus feature full typed λ -calculus. The reduction rules are significantly more complex than those of $\lambda_{\mathcal{N}}$, due to the treatment that code mobility requires; strong normalization and confluence do not hold. The concurrent λ -calculus of [Aschieri and Zorzi 2016] is typed by first-order classical logic. It resembles a session-typed calculus, but supports only data transmission. It is strongly normalizing, but non-confluent and does not enjoy any subformula property. The calculus Lollipop in [Mazurak and Zdancewic 2010] is a parallel interpretation of control operators, rather than a calculus for communication. The completeness of the typing system with respect to linear logic and the subformula property are not shown, which makes unclear if it actually is a legitimate proof system for the logic.

$\lambda_{\mathcal{N}}$ and Proof-Nets. Proof-nets for multiplicative linear logic [Girard 1987] can encode several programming languages, among them linear λ -calculus [Girard et al. 1989] and typed π -calculus [Honda and Laurent 2010]. We believe $\lambda_{\mathcal{N}}$ can be encoded in proof-nets as well and hence offers a new way of using proof-nets for representing functional concurrent computation. However, $\lambda_{\mathcal{N}}$ is based on the connectives \multimap , \wp and normalization, while proof-nets are based on the connectives \otimes , \wp and cut-elimination, which makes $\lambda_{\mathcal{N}}$ more suitable as syntax for functional concurrent programming. Indeed, $\lambda_{\mathcal{N}}$ is a full-blown programming language right off the bat, while proof-nets were created as an elegant and economical proof system for linear logic.

$\lambda_{\mathcal{N}}$ and Exponentials. As type system of $\lambda_{\mathcal{N}}$, multiplicative linear logic already features all the important characters of functional concurrent computation, and because of both its expressiveness and simplicity, it deserves to be treated as a type system on its own, as traditional in linear logic. One non-trivial problem is to extend the type system of $\lambda_{\mathcal{N}}$ with exponentials. We do not see any

real obstacles to doing that, but we leave the problem as future work. As a result, $\lambda_{\mathfrak{A}}$ may gain more duplication and replication abilities, as those of CP [Wadler 2012].

$\lambda\mu$ and $\lambda_{\mathfrak{A}}$: Sequentiality vs Parallelism. As far as the logical rules for implication are concerned, the type system of the original version of $\lambda\mu$ [Parigot 1991] is exactly the non-linear version of the type system of $\lambda_{\mathfrak{A}}$. However, $\lambda\mu$ and $\lambda_{\mathfrak{A}}$ strongly diverge if we consider their computational behaviour. The crucial difference between $\lambda_{\mathfrak{A}}$ and $\lambda\mu$ is that while in $\lambda\mu$ a whole list of formulas is used to type a single term: $t : A_1, \dots, A_n$, in $\lambda_{\mathfrak{A}}$ each element of the list of formulas types a different term: $t_1 : A_1, \dots, t_n : A_n$. Indeed, although the type system of $\lambda\mu$ actually builds several terms, it does it sequentially, so the result is a unique term; instead, the type system of $\lambda_{\mathfrak{A}}$ builds several terms in parallel and let them communicate. The reason is that in $\lambda\mu$ only one distinguished “active” formula of each list can be directly involved in an inference. Labels α, β, \dots – called μ -variables – are introduced to explicitly activate a formula by the rule $\frac{t : \Gamma \Rightarrow A^\beta, \Delta}{\mu\beta t : \Gamma \Rightarrow A, \Delta}$ or deactivate it by the rule $\frac{t : \Gamma \Rightarrow A, \Delta}{[\beta]t : \Gamma \Rightarrow A^\beta, \Delta}$, where Γ is a context containing declaration of λ -variables x, y, \dots and Δ is a multiset containing formulae labeled by μ -variables α, β, \dots . As we can see from the condition on the context Δ , at most one formula in each sequent can be without a label.

The history of activations and deactivations occurring in the derivation is recorded inside the corresponding $\lambda\mu$ -term, as we can see in the following example

$$\frac{\frac{\frac{w : A \rightarrow B, \Delta'}{[\beta]w : (A \rightarrow B)^\beta, \Delta'} \vdots u : (A \rightarrow B)^\beta, \Delta}{\mu\beta.u : A \rightarrow B, \Delta} \quad v : A, \Sigma}{(\mu\beta.u)v : B, \Delta, \Sigma}$$

Here the construction of w has been paused for a while and resumed with the last inference rule. Hence, v can be considered as the “real” argument of w , rather than of $(\mu\beta.u)v$. As consequence, there is the reduction $(\mu\beta.u)v \mapsto \mu\beta.u[[\beta](wv)/[\beta]w]$, where the argument v of $\mu\beta.u$ is transmitted as argument to all subterms of t with label $[\beta]$.

Intuitionistic Linear Logic with \mathfrak{A} . The λ -calculus in [Eades III and de Paiva 2016] provides a computational interpretation of an intuitionistic linear logic with \mathfrak{A} . The typing system is a linear sequent calculus, so the corresponding λ -calculus is rather obtained by translation than by a Curry-Howard isomorphism between proofs and λ -terms. To obtain such an isomorphism, our move to natural deduction was necessary. Also, in [Eades III and de Paiva 2016] cut-elimination is not interpreted as communication, so the system does not support concurrency. A radical change in the treatment of linear implication was needed to recover communication and an interpretation of \mathfrak{A} as parallel operator was needed to recover parallelism, which is what we did. It is also instructive to compare $\mathfrak{A}E$ rule to the corresponding rule in [Eades III and de Paiva 2016]:

$$\frac{\Gamma \Rightarrow s : A \mathfrak{A} B, \Delta \quad \Sigma_1, x : A \Rightarrow v : C, \Theta_1 \quad \Sigma_2, y : B \Rightarrow w : D, \Theta_2}{\Gamma, \Sigma_1, \Sigma_2 \Rightarrow \text{let } s \text{ be } (A \mathfrak{A} -) \text{ in } v : C, \text{let } s \text{ be } (- \mathfrak{A} B) \text{ in } w : D, \Delta, \Theta_1, \Theta_2} \mathfrak{A}E_2$$

This rule duplicates s , compromising linearity of variables. But computation linearity is safe if we consider that s virtually occurs twice, but it will only be used once: one half of it in v , the other half in w ; the two spare halves will be discarded. Since $\lambda_{\mathfrak{A}}$ is built around communication, we re-interpreted disjunction redexes as communications in order to avoid unnecessary duplication. Indeed, in the $\mathfrak{A}E$ rule with premises $\Gamma \Rightarrow s : A \mathfrak{A} B, \Delta, \quad \Sigma_1, x : A \Rightarrow v : C, \Theta_1 \quad \text{and} \quad \Sigma_2, y : B \Rightarrow w : D, \Theta_2$, we just let s be an autonomous term – living in parallel with all other terms in our multiple-conclusion – and establish a communication channel from s to both v and w . When s assumes the right form, we trigger a communication transmitting one half of s to v and the other half to w , as required.

REFERENCES

- Samson Abramsky. 1994. Proofs as Processes. *Theor. Comput. Sci.* 135, 1 (1994), 5–9.
- Samson Abramsky, Simon J. Gay, and Rajagopal Nagarajan. 1996. Interaction categories and the foundations of typed concurrent programming. In *Proceedings of the NATO Advanced Study Institute on Deductive Program Design, Marktoberdorf, Germany*. 35–113.
- Federico Aschieri, Agata Ciabattone, and Francesco A. Genco. 2017. Gödel logic: from natural deduction to parallel computation. In *LICS 2017*. 1–12.
- Federico Aschieri, Agata Ciabattone, and Francesco A. Genco. 2018. Classical proofs as parallel programs. In *GandALF 2018*. 43–57.
- Federico Aschieri and Margherita Zorzi. 2016. On Natural Deduction in Classical First-Order Logic: Curry–Howard Correspondence, Strong Normalization and Herbrand’s Theorem. *Theoretical Computer Science* 625 (2016), 125–146.
- Arnon Avron. 1991. Hypersequents, logical consequence and intermediate logics for concurrency. *Annals of Mathematics and Artificial Intelligence* 4, 3 (1991), 225–248.
- Emmanuel Beffara. 2006. A Concurrent Model for Linear Logic. *Electr. Notes Theor. Comput. Sci.* 155 (2006), 147–168.
- Gianluigi Bellin and Philip J. Scott. 1994. On the pi-Calculus and Linear Logic. *Theor. Comput. Sci.* 135, 1 (1994), 11–65.
- Gérard Boudol. 1992. *Asynchrony and the Pi-calculus*. Research Report RR-1702. INRIA. 15 pages.
- Luis Caires and Frank Pfenning. 2010. Session types as intuitionistic linear propositions. In *CONCUR 2010*. 222–236.
- Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. 2017. Multiparty session types as coherence proofs. *Acta Inf.* 54, 3 (2017), 243–269.
- Carlo Cellucci. 1992. Existential instantiation and normalization in sequent natural deduction. *Annals of Pure and Applied Logic* 58, 2 (1992), 111 – 148.
- Ornela Dardha and Simon J. Gay. 2018. A New Linear Logic for Deadlock-Free Session-Typed Processes. In *Foundations of Software Science and Computation Structures - 21st International Conference, FOSSACS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings*. 91–109.
- Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session types revisited. In *Principles and Practice of Declarative Programming, PPDP’12, Leuven, Belgium - September 19 - 21, 2012*. 139–150.
- Harley Eades III and Valeria de Paiva. 2016. Multiple Conclusion Linear Logic: Cut Elimination and More. In *Logical Foundations of Computer Science - International Symposium, LFCS 2016, Deerfield Beach, FL, USA, January 4-7, 2016, Proceedings*. 90–105.
- Simon J. Gay and Vasco Thudichum Vasconcelos. 2010. Linear type theory for asynchronous session types. *J. Funct. Program.* 20, 1 (2010), 19–50.
- Jean-Yves Girard. 1987. Linear logic. *Theoretical computer science* 50, 1 (1987), 1–101.
- Jean-Yves Girard, Yves Lafont, and Paul Taylor. 1989. *Proofs and Types*. Cambridge University Press.
- Kohei Honda and Olivier Laurent. 2010. An exact correspondence between a typed pi-calculus and polarised proof-nets. *Theor. Comput. Sci.* 411, 22-24 (2010), 2223–2238.
- Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. 1999. Linearity and the pi-calculus. *ACM Trans. Program. Lang. Syst.* 21, 5 (1999), 914–947.
- Wen Kokke, Fabrizio Montesi, and Marco Peressotti. 2019. Better late than never: a fully-abstract semantics for classical processes. In *POPL 2019*. 24:1–24:29.
- Jean-Louis Krivine. 2009. Classical Realizability. *Interactive models of computation and program behavior, Panoramas et synthèses* (2009), 197–229.
- Sam Lindley and J. Garrett Morris. 2015. A Semantics for Propositions as Sessions. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*. 560–584.
- Karl Mazurak and Steve Zdancewic. 2010. Lollipop: to concurrency from classical linear logic via curry-howard and control. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*. 39–50.
- Massimo Merro and Davide Sangiorgi. 2004. On asynchrony in name-passing calculi. *Mathematical Structures in Computer Science* 14, 5 (2004), 715–767.
- Robin Milner. 1980. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92. Springer.
- Robin Milner. 1984. Lectures on a calculus for Communicating systems. In *Seminar on Concurrency*. Springer, 197–219.
- Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, I. *Inf. Comput.* 100, 1 (1992), 1–40.
- Michel Parigot. 1991. Free Deduction: An Analysis of “Computations” in Classical Logic. In *Logic Programming*. 361–380.
- Michel Parigot. 1992. Lambda-Mu-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In *LPAR 1992*. 190–201.

- Jennifer Paykin and Steve Zdancewic. 2016. Linear $\lambda\mu$ is CP (more or less). In *A List of Successes That Can Change the World - Essays Dedicated to Philip Wadler on the Occasion of His 60th Birthday*. 273–291.
- Jorge A. Pérez, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Linear Logical Relations for Session-Based Concurrency. In *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*. 539–558.
- Dag Prawitz. 1971. Ideas and Results in Proof Theory. In *Proceedings of the Second Scandinavian Logic Symposium*.
- Klaas Pruiksma and Frank Pfenning. 2019. A Message-Passing Interpretation of Adjoint Logic. In *Proceedings Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES@ETAPS 2019, Prague, Czech Republic, 7th April 2019*. 60–79.
- Davide Sangiorgi. 1993. *Expressing mobility in process algebras: first-order and higher-order paradigms*. Ph.D. Dissertation. The University of Edinburgh.
- Bernardo Toninho, Luís Caires, and Frank Pfenning. 2013. Higher-Order Processes, Functions, and Sessions: A Monadic Integration. In *ESOP 2013*. 350–369.
- Philip Wadler. 2012. Propositions as Sessions. *ICFP 2012* 24 (2012), 384–418. Issue 2–3.
- Philip Wadler. 2015. Propositions as Types. *Commun. ACM* 58, 12 (2015), 75–84.