Formulas as Processes, Deadlock-Freedom as Choreographies

Matteo Acclavio[1], Giulia Manara[2,3(✉)], and Fabrizio Montesi[3]

[1] University of Sussex, Brighton, UK
[2] Université Paris Cité, Paris, France
manara@imada.sdu.dk
[3] Universitá Roma Tre, Roma, Italy
[4] University of Southern Denmark, Odense, Denmark

Abstract. We introduce a novel approach to studying properties of processes in the π-calculus based on a processes-as-formulas interpretation, by establishing a correspondence between specific sequent calculus derivations and computation trees in the reduction semantics of the recursion-free π-calculus. Our method provides a simple logical characterisation of deadlock-freedom for the recursion- and race-free fragment of the π-calculus, supporting key features such as cyclic dependencies and an independence of the name restriction and parallel operators. Based on this technique, we establish a strong completeness result for a nontrivial choreographic language: all deadlock-free and race-free finite π-calculus processes composed in parallel at the top level can be faithfully represented by a choreography.
With these results, we show how the computation-as-derivation paradigm extends the reach of logical methods for the study of concurrency, by bridging gaps between logic, the expressiveness of the π-calculus, and the expressiveness of choreographic languages.

1 Introduction

The Curry-Howard isomorphism is a remarkable example of the synergy between logic and programming languages, which establishes a *formulas-as-types* and *proofs-as-programs* (and *computation-as-reduction*) correspondences for functional programs [67,90]. In view of this success, an analogous *proofs-as-processes* correspondence has been included in the agenda of the study of concurrent programming languages [1,17]. The main idea in this research line is that, as types provide a high-level specification of the input/output data types for a function, propositions in linear logic correspond to *session types* [56] that specify the communication actions performed by processes (as in process calculi) [20]. However, while the Curry-Howard correspondence fits functional programming naturally, it does not come without issues when applied to concurrency (we discuss the details in related work, section 5). This is because functional programming deals with the 'sequential' aspect of computation (given an input, return a specific output), while most of the interesting aspects of concurrent computation are

V. Vafeiadis (Ed.): ESOP 2025, LNCS 15694, pp. 23–55, 2025.
https://doi.org/10.1007/978-3-031-91118-7_2

about the communication patterns used during the computation itself. Therefore, as suggested in numerous works (e.g., [69,72,11,11,13]), the Curry-Howard correspondence may not be the right lens for the study of concurrent programs.

In this paper we investigate an alternative proof-theoretical approach to the study of processes, which is based on logical operators that faithfully model the fundamental operators of process calculi (like prefixing, parallel, and restriction). Our approach is close to the *computation-as-deduction* paradigm, where program executions are modelled as proof searches in a given sequent calculus; executions are therefore grounded in a logic by construction, as originally proposed by Miller in [69]. Specifically, we interpret formulas as processes, inference rules as rules of an operational semantics, and (possibly partial) derivations as execution trees (snapshots of computations up to a certain point).

The approach we follow is not to be confused with the intent of using logic as an auxiliary language to enunciate statements about computations, that is, viewing *computation-as-model* as done in Hennessy-Milner logic [51,54], modal μ-calculus [65], Hoare logic [52], or dynamic logics [48,10]. We are instead interested in directly reasoning on programs and their execution using the language of the programs itself. This allows for an immediate transfer of properties of proofs to properties of programs, without needing intermediate structures (e.g., models) or languages (e.g., types).

1.1 Contributions of the paper

We consider the recursion-free fragment of the π-calculus – as presented in [22,42] – and embed it in the language of the system PiL from [8]. PiL extends Girard's first order multiplicative and additive linear logic [43] with a non-commutative and non-associative sequentiality connective ($\blacktriangleleft$), and nominal quantifiers (И and its dual Я) for variable scoping.

Using this embedding, we prove the following main results.

1. We show that the operational semantics of the π-calculus is captured by the linear implication ($\multimap$) in PiL: if we denote by $[\![P]\!]$ the formula encoding the process P, then
   - if P is a process reducing to P' by performing a communication or an external choice, then $\vdash_{\mathsf{PiL}} [\![P']\!] \multimap [\![P]\!]$; while
   - if P may reduce to $P_{\ell_1}, \ldots, P_{\ell_n}$ by performing an internal choice, then $\vdash_{\mathsf{PiL}} \&_{i=1}^{n} ([\![P_{\ell_i}]\!]) \multimap [\![P]\!]$.

   Crucial to prove this result is our proof that the system PiL supports a substitution principle, which allows us to simulate reductions within a context.
2. We establish a computation-as-deduction correspondence, which we use to characterise two key safety properties studied for race-free processes in terms of derivability in PiL: *deadlock-freedom*, i.e., the property that a process can always keep executing until it eventually terminates [96]; and *progress*, i.e., the property that if a process gets stuck, it is always because of a missing interaction with an action that can be provided by the environment [29].[4]

[4] For progress, in this paper we restrict our attention to processes that do not send restricted names.

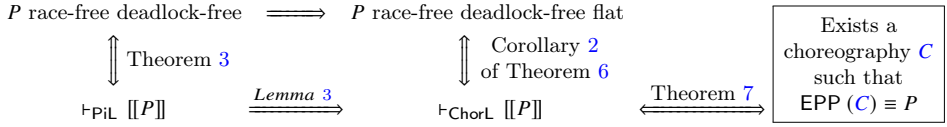


Fig. 1. Road map of the main technical results in this work.

In particular, thanks to the structure of PiL and its operators, we can successfully detect safe processes that were previously problematic in the logical setting due to cyclic dependencies, like the process in Equation (1) below.

$$(\nu x)(\nu y)\,(x!\langle a\rangle.y \triangleleft \{\ell : y!\langle b\rangle.\mathsf{Nil}\} \mid x?(a).y \triangleright \{\ell : y?(b).\mathsf{Nil}, \ell' : z!\langle c\rangle.\mathsf{Nil}\}) \quad (1)$$

3. We show that our approach provides an adequate logical foundation for choreographic programming, a paradigm where programs are *choreographies* (coordination plans) that express the communications that a network of processes should enact [77].[5] Specifically, we establish a *choreographies-as-proofs* correspondence by using a sequent system, called ChorL, that consists of rules derivable in PiL.
   Our choreographies-as-proofs correspondence has an important consequence. An open question in theory of choreographic programming [78] is about how expressive choreographies can be:

   What are the processes that can be captured by choreographies? (2)

   To date, there are no answers to this question for the setting of processes with unrestricted name mobility and cyclic dependencies. Our correspondence implies a strong completeness result: in our setting, all and only race- and deadlock-free networks can be expressed as choreographies. This is the first such completeness result in the case of recursion-free networks.

1.2 Structure of the paper

In Section 2 we report PiL and prove the additional technical results required for our development. In Section 3 we recall the syntax and semantics of the π-calculus, introduce an alternative reduction semantics with the same expressiveness with respect to the property of deadlock-freedom, show that the reduction semantics of the π-calculus is captured by linear implication in PiL, and that each step of the reduction semantics of the π-calculus can be seen as blocks of rules in this system. In Section 4 we define our choreographic language and provide our completeness result for choreographies. We discuss related work in Section 5. We conclude in Section 6, where we discuss research directions opened by this work. Due to space constraints, details of certain proofs are provided in the extended version of this paper [9].

[5] Networks are parallel compositions of sequential processes assigned to distinct locations.

Formulas		De Morgan Laws			α-equivalence
$A, B :=$	$\circ$ unit (atom)				
$\mid$	$\langle x!y\rangle$ atom				$a = a$
$\mid$	$(x?y)$ atom	$\circ^{\perp}$	$=$	$\circ$	if $a \in \{\circ, \langle x!y\rangle, (x?y)\}$
$\mid$	$A \parr B$ par	$(A^{\perp})^{\perp}$	$=$	A	
$\mid$	$A \otimes B$ tensor	$\langle x!y\rangle^{\perp}$	$=$	$(x?y)$	$A_1 \odot A_2 = B_1 \odot B_2$
$\mid$	$A \blacktriangleleft B$ prec	$(A \parr B)^{\perp}$	$=$	$A^{\perp} \otimes B^{\perp}$	if $A_i = B_i$
$\mid$	$A \oplus B$ oplus	$(A \blacktriangleleft B)^{\perp}$	$=$	$A^{\perp} \blacktriangleleft B^{\perp}$	and $\odot \in \{\parr, \blacktriangleleft, \otimes, \oplus, \&\}$
$\mid$	$A \,\&\, B$ with	$(A \oplus B)^{\perp}$	$=$	$A^{\perp} \,\&\, B^{\perp}$	
$\mid$	$\forall x.A$ for all	$(\forall x.A)^{\perp}$	$=$	$\exists x.A^{\perp}$	$\eth x.A = \eth y.A\,[y/x]$
$\mid$	$\exists x.A$ exists	$(И\mathcal{X}.A)^{\perp}$	$=$	$Я\mathcal{X}.A^{\perp}$	y fresh for A and $\eth \in \{И, Я, \forall, \exists\}$
$\mid$	$Иx.A$ new				
$\mid$	$Яx.A$ ya				

Fig. 2. Formulas (with $x, y \in \mathcal{V}$), and their syntactic equivalences.

2 Non-commutative logic

In this section we recall PiL and some of its established properties [8]. We then prove additional results required for our development.

2.1 The system PiL

The language of PiL is a first-order language containing the following.

- Atoms generated by (i) a countable set of variables $\mathcal{V}$, (ii) a binary predicate on symbols $\langle -!-\rangle$, and (iii) its dual binary predicate $(-?-)$;
- The multiplicative connectives for disjunction ($\parr$) and conjunction ($\otimes$) and the additive connectives for disjunction ($\oplus$) and conjunction ($\&$) form *multiplicative additive linear logic* [43]. We generalize the standard binary $\oplus$ and $\&$, allowing them to have any positive arity (including 1) in order to avoid dealing with associativity and commutativity when modelling choices;
- A binary non-commutative and non-associative self-dual multiplicative connective ***precede*** ($\blacktriangleleft$), whose properties reflect the ones of the prefix operator used in standard process calculi (see CCS [74] and π-calculus [75,42]);
- A unit ($\circ$). We observe that its properties reflect the ones of the terminated process Nil. Notably, it is the neutral element of the connectives that we will use to represent parallelism ($\parr$) and sequentiality ($\blacktriangleleft$), and it is derivable from no assumption (thus also neutral element of $\otimes$);
- The standard first order existential ($\exists$) and universal ($\forall$) quantifiers.
- A *nominal quantifier* ***new*** ($И$), and its dual ***ya*** ($Я$). We observe that these restrict variable scope in formulas in the same way that the ν constructor in process calculi restricts the scope of names.

More precisely, we consider ***formulas*** generated by grammar in Figure 2 modulo the standard ***α-equivalence*** from the same figure. From now on, we assume formulas and sequents to be ***clean***, that is, such that each variable $x \in \mathcal{V}$ occurring in them can be bound by at most a unique universal quantifier or at

most a pair of dual nominal quantifiers, and, if bound, it cannot occur free .[6] The ***(linear) implication*** $A \multimap B$ (resp. the ***logical equivalence*** $A \multimap\!\!\circ\, B$) is defined as $A^\perp \parr B$ (resp. as $(A \multimap B) \otimes (B \multimap A)$), where the ***negation*** $(\cdot^\perp)$ is defined by the ***de Morgan duality*** in Figure 2.

The set $\mathsf{free}(A)$ (resp. $\mathsf{free}(\Gamma)$) of ***free variables*** of a formula A (resp. of a sequent $\Gamma = A_1, \ldots, A_n$) is the set of atoms occurring in A which are not bound by any quantifier (resp. the set $\bigcap_{i=1}^{n} \mathsf{free}(A_i)$). A ***context*** is a formula containing a single occurrence of a propositional variable $\bullet$ (called ***hole***). We denote by $C[A] := C\,[A/\bullet]$. A ***И⅄-context*** is a context $\mathcal{K}[\bullet]$ of the form $\mathcal{K}[\bullet] = Иx_1 \ldots Иx_n.([\bullet] \parr A)$ for a $n \in \mathbb{N}$.

In this work we assume the reader to be familiar with the syntax of sequent calculus (see, e.g., [93]), but we recall here the main definitions.

Notation 1. *A **sequent** is a set of occurrences of formulas.*[7] *A **sequent rule*** r *with **premise** sequents* $\Gamma_1, \ldots, \Gamma_n$ *and **conclusion*** Γ *is an expression of the form* $\mathsf{r}\,\dfrac{\Gamma_1 \quad \cdots \quad \Gamma_n}{\Gamma}$. *A formula occurring in the conclusion (resp. in a premise) of a rule but in none of its premises (resp. not in its conclusion) is said **principal** (resp. **active**). Given a set of rules* X*, a **derivation** in* X *is a non-empty tree* $\mathcal{D}$ *of sequents, whose root is called **conclusion**, such that every sequent occurring in* $\mathcal{D}$ *is the conclusion of a rule in* X*, whose children are (all and only) the premises of the rule. An **open derivation** is a derivation whose leaves may be conclusions of no rules, in which case are called **open premises**. We may denote a derivation (resp. an open derivation with a single open premise* Δ*) with conclusion* Γ *by* $\begin{matrix}\mathcal{D}\,\|\\ \vdash \Gamma\end{matrix}$ $\left(\text{resp. } \begin{matrix}\vdash \Delta\\ \mathcal{D}\,\|\\ \vdash \Gamma\end{matrix}\right)$. *Finally we may write* $\mathsf{r}\,\dfrac{\vdash \Gamma_1 \quad \cdots \quad \vdash \Gamma_n}{\vdash \Gamma}$ *if there is an open derivation with premises* $\Gamma_1, \ldots, \Gamma_n$ *and conclusion* Γ *made only of rules* r.

A ***nominal variable*** is an element of the form x^∇ with $x \in \mathcal{X}$ and $\nabla \in \{И, Я\}$. If $\mathcal{S}$ is a set of nominal variables, we say that x ***occurs*** in $\mathcal{S}$ if $x^И$ or $x^Я$ is an element of $\mathcal{S}$. A ***(nominal) store*** $\mathcal{S}$ is a set of nominal variables such that each variable occurs at most once in $\mathcal{S}$. A ***judgement*** $\mathcal{S} \vdash \Gamma$ is a pair consisting of a clean sequent Γ and a store $\mathcal{S}$. We write judgements $\mathcal{S} \vdash \Gamma$ with $\mathcal{S} = \varnothing$ (resp. $\mathcal{S} = \{x_1^{\nabla_1}, \ldots, x_n^{\nabla_n}\}$) simply as $\vdash \Gamma$ (resp. $x_1^{\nabla_1}, \ldots, x_n^{\nabla_n} \vdash \Gamma$, i.e. omitting parenthesis). We write $\mathcal{S}_1, \mathcal{S}_2$ to denote the union of two stores such that a same variable does not occur in both $\mathcal{S}_1$ and $\mathcal{S}_2$ – i.e., a disjoint union.

[6] This can be considered as a variation of *Barendregt's convention*. It allows us to avoid variable renaming for universal and nominal quantifier rules in derivations, by assuming the bound variable to be the eigenvariable of the quantifier or the shared fresh name in the case of a pair of dual nominal quantifiers.

[7] In a set of occurrences of formulas, it is assumed that each formula has a unique identifier, differently from a multiset of formulas where each formula has a multiplicity. The former definition simplifies the process of tracing occurrences of formulas in a derivation, as we need in Section 4.

$$\mathsf{ax}\,\frac{}{\mathcal{S} \vdash \langle x!y\rangle, (x?y)} \qquad \parr\,\frac{\mathcal{S} \vdash \Gamma, A, B}{\mathcal{S} \vdash \Gamma, A \parr B} \qquad \otimes\,\frac{\mathcal{S}_1 \vdash \Gamma, A \quad \mathcal{S}_2 \vdash B, \Delta}{\mathcal{S}_1, \mathcal{S}_2 \vdash \Gamma, A \otimes B, \Delta} \qquad \circ\,\frac{}{\mathcal{S} \vdash \circ} \qquad \mathsf{mix}\,\frac{\mathcal{S}_1 \vdash \Gamma \quad \mathcal{S}_2 \vdash \Delta}{\mathcal{S}_1, \mathcal{S}_2 \vdash \Gamma, \Delta}$$

$$\oplus\,\frac{\mathcal{S} \vdash \Gamma, A_k}{\mathcal{S} \vdash \Gamma, \bigoplus_{i=1}^{n} A_i} \text{ for a } k \in \{1, \dots, n\} \qquad \&\,\frac{\mathcal{S} \vdash \Gamma, A_1 \quad \cdots \quad \mathcal{S} \vdash \Gamma, A_n}{\mathcal{S} \vdash \Gamma, \&_{i=1}^{n} A_i} \qquad \forall\,\frac{\mathcal{S} \vdash \Gamma, A}{\mathcal{S} \vdash \Gamma, \forall x.A}\,\dagger \qquad \exists\,\frac{\mathcal{S} \vdash \Gamma, A\,[y/x]}{\mathcal{S} \vdash \Gamma, \exists x.A}$$

$$\blacktriangleleft\,\frac{\mathcal{S}_1 \vdash \Gamma, A, C \quad \mathcal{S}_2 \vdash \Delta, B, D}{\mathcal{S}_1, \mathcal{S}_2 \vdash \Gamma, \Delta, A \blacktriangleleft B, C \blacktriangleleft D} \qquad \blacktriangleleft^{\circ}\,\frac{\mathcal{S}_1 \vdash \Gamma, A \quad \mathcal{S}_2 \vdash \Delta, B}{\mathcal{S}_1, \mathcal{S}_2 \vdash \Gamma, \Delta, A \blacktriangleleft B} \qquad \boxed{\mathsf{cut}\,\frac{\mathcal{S}_1 \vdash \Gamma, A \quad \mathcal{S}_2 \vdash A^{\perp}, \Delta}{\mathcal{S}_1, \mathcal{S}_2 \vdash \Gamma, \Delta}}$$

$$И_{\circ}\,\frac{\mathcal{S} \vdash \Gamma, A}{\mathcal{S} \vdash \Gamma, Иx.A}\,\dagger \qquad И_{\mathsf{load}}\,\frac{\mathcal{S}, x^{И} \vdash \Gamma, A}{\mathcal{S} \vdash \Gamma, Иx.A}\,\dagger \qquad И_{\mathsf{pop}}\,\frac{\mathcal{S} \vdash \Gamma, A\,[y/x]}{\mathcal{S}, y^{И} \vdash \Gamma, Яx.A}$$

$$Я_{\circ}\,\frac{\mathcal{S} \vdash \Gamma, A}{\mathcal{S} \vdash \Gamma, Яx.A}\,\dagger \qquad Я_{\mathsf{load}}\,\frac{\mathcal{S}, x^{Я} \vdash \Gamma, A}{\mathcal{S} \vdash \Gamma, Яx.A}\,\dagger \qquad Я_{\mathsf{pop}}\,\frac{\mathcal{S} \vdash \Gamma, A\,[y/x]}{\mathcal{S}, y^{Я} \vdash \Gamma, Иx.A}$$

Fig. 3. Sequent calculus rules, with $\dagger := x \notin \mathsf{free}(\Gamma)$.

The system PiL is defined by the all rules in Figure 3 except the rule cut. We write $\vdash_{\mathsf{PiL}} \Gamma$ to denote that the judgement $\varnothing \vdash \Gamma$ is derivable in PiL.

Remark 1. The system $\mathsf{MLL}^1 = \{\mathsf{ax}, \parr, \otimes, \exists, \forall\}$ is the standard one for first order multiplicative linear logic [43]. The rules $\oplus$ and & are generalisations of the standard ones in additive linear logic for the n-ary generalized connectives we consider here; thus, in proof search, the rule $\oplus$ keeps only one A_k among all A_i occurring in $\bigoplus_{i=1}^{n} A_i$, and the rule & branches the proof search in n premises. The rule mix and $\circ$ are standard for multiplicative linear logic with mix in presence of units[8] [40,28], and the rule $\blacktriangleleft^{\circ}$ ensures that the unit $\circ$ is not only the unit for the connectives $\parr$ and $\otimes$, but also for $\blacktriangleleft$. The rule $\blacktriangleleft$ is required to capture the self-duality of the connective $\blacktriangleleft$; it should be read as introducing at the same time the connective $\blacktriangleleft$ and its dual (which in this case is $\blacktriangleleft$ itself) – as a general underlying pattern for multiplicative connectives, see [2, Remark 5].

The store is used to guarantee that each И is linked to at most a unique Я (or vice versa) in any branch of a derivation. If a rule $И_{\circ}$ (resp. $Я_{\circ}$) is applied, then the nominal quantifier is not linked, reason why the rule reminds the standard universal quantifier rule. Otherwise, either the rule $И_{\mathsf{load}}$ (resp. $Я_{\mathsf{load}}$) loads a nominal variable in the store, or a rule $И_{\mathsf{pop}}$ (resp. $Я_{\mathsf{pop}}$) uses a nominal variable (of dual type) occurring in the store as a witness variable. Note that in a derivation with the conclusion a judgement with empty store any $И_{\mathsf{pop}}$ (resp. $Я_{\mathsf{pop}}$) is uniquely linked to a $И_{\mathsf{load}}$ (resp. $Я_{\mathsf{load}}$) below it.

2.2 Proof Theoretical Properties of PiL

We now recall some basic proof-theoretical properties of the system PiL and then prove additional results (Theorem 2) that will be important for the main technical results in this paper.

[8] In presence of mix the two multiplicative units collapse.

In PiL we can prove that atomic axioms are sufficient to prove that the implication $A \multimap A$ holds for any formula A. Moreover, the cut-rule is admissible in these systems, allowing us to conclude the transitivity of the linear implication, as well as the sub-formula property for all the rules of the systems.

Theorem 1 ([8]). *Let Γ be a non-empty sequent in* PiL*. Then*

1. $\vdash_{\mathsf{PiL}} A^{\perp}, A$ *for any formula* A*;*
2. *if* $\vdash_{\mathsf{PiL}\cup\{\mathsf{cut}\}} \Gamma$*, then* $\vdash_{\mathsf{PiL}} \Gamma$*;*
3. *if* $\vdash_{\mathsf{PiL}} A \multimap B$ *and* $\vdash_{\mathsf{PiL}} B \multimap C$*, then* $\vdash_{\mathsf{PiL}} A \multimap C$*.*

Proposition 1 ([8]). *The following logical equivalences are derivable in* PiL *(for any σ permutation over $\{1,\dots,n\}$).*

$$\begin{array}{c} (A \parr \circ) \multimap A \\ (A \blacktriangleleft \circ) \multimap A \\ (A \parr B) \parr C \multimap A \parr (B \parr C) \\ A \parr B \multimap B \parr A \\ \left(\bigoplus_{i=1}^{n} A_i\right) \multimap \left(\bigoplus_{i=1}^{n} A_{\sigma(i)}\right) \\ \left(\&_{i=1}^{n} A_i\right) \multimap \left(\&_{i=1}^{n} A_{\sigma(i)}\right) \end{array} \quad \left| \quad \begin{array}{c} (\Pi x.\Pi y.A) \multimap (\Pi y.\Pi x.A) \\ \left(\&_{i=1}^{n} \Pi x.A_i\right) \multimap \left(\Pi x.\&_{i=1}^{n} A_i\right) \\ \left(\bigoplus_{i=1}^{n} \Pi A_{.i}\right) \multimap \left(\Pi x.\bigoplus_{i=1}^{n} A_i\right) \\ \hline \Pi x.(A \parr D) \multimap (\Pi x.A) \parr D \\ \Pi x.D \multimap D \\ \text{if } x \notin \mathsf{free}(D) \end{array} \right. \tag{3}$$

Moreover, $\vdash_{\mathsf{PiL}} \left(\&_{i=1}^{n}(A_i \parr B)\right) \multimap \left(\left(\&_{i=1}^{n} A_i\right) \parr B\right)$.

In addition to these properties, our development requires some new ones showing that implication is preserved in different contexts. This is necessary because some rules in the operational semantics of the π-calculus enables rewriting of deeply nested subterms. For this reason, we are required to establish properties similar to those for proving *subject reduction* in λ-calculus. That is, we prove that in PiL we can still reproduce the application of inference rules inside contexts preserving soundness and completeness. These necessary properties are collected in the next theorem.

Theorem 2. *For any context $C[\bullet]$ and $\Pi\parr$-context $\mathcal{K}[\bullet]$ we have:*

1. *if* $\vdash_{\mathsf{PiL}} A \multimap B$*, then* $\vdash_{\mathsf{PiL}} C[A] \multimap C[B]$*;*
2. *if* $\vdash_{\mathsf{PiL}} A_i \multimap B$ *for* $i \in \{1,\dots,n\}$*, then* $\vdash_{\mathsf{PiL}} \&_{i=1}^{n} \mathcal{K}[A_i] \multimap \mathcal{K}[B]$*;*
3. *if* $\vdash_{\mathsf{PiL}} A \multimap A'$ *and* $\vdash_{\mathsf{PiL}} C[A'] \multimap B$*, then* $\vdash_{\mathsf{PiL}} C[A] \multimap B$*.*

Proof. Item 1 and Item 2 are proven by induction on the structure of the contexts. To prove Item 3 we use Theorem 1.3 since if $\vdash_{\mathsf{PiL}} A \multimap A'$, then by 1 also $\vdash_{\mathsf{PiL}} C[A] \multimap C[A']$. Details of the proof are available in [9].

Remark 2. In the proofs-as-processes interpretation, cut is the linchpin that triggers the rewriting simulating the reduction semantics (cut elimination). In our work, as in general in the study of processes-as-formulas, the cut-rule is freed from being the keystone of the system. Instead, the admissibility of the cut-rule in the computation-as-deduction approach guarantees the existence of canonical models [71,70,53]. In particular, we use this property to tame the syntactic bureaucracy of the reduction semantics due to rules Par, Res, and $\mathsf{Struc}^{\Rrightarrow}$, as well as to ensure the transitivity of logical implication (required to compose reduction steps).

Processes		Free names	bound names
$P, Q, R \coloneqq \mathsf{Nil}$	nil	$\varnothing$	$\varnothing$
$\mid x!\langle y\rangle.P$	send (y on x)	$\mathcal{F}_P \setminus \{x, y\}$	$\mathcal{B}_P$
$\mid x?(y).P$	receive (y on x)	$\mathcal{F}_P \setminus \{x\}$	$\mathcal{B}_P \cup \{y\}$
$\mid P \mid Q$	parallel	$\mathcal{F}_P \cup \mathcal{F}_Q$	$\mathcal{B}_P \cup \mathcal{B}_Q$
$\mid (\nu x)P$	nu	$\mathcal{F}_P \setminus \{x\}$	$\mathcal{B}_P \cup \{x\}$
$\mid x \triangleleft \{\ell : P_\ell\}_{\ell \in L}$	label send (on x)	$\bigcup_{\ell \in L} \mathcal{F}_{P_\ell}$	$\bigcup_{\ell \in L} \mathcal{B}_{P_\ell}$
$\mid x \triangleright \{\ell : P_\ell\}_{\ell \in L}$	label receive (on x)	$\bigcup_{\ell \in L} \mathcal{F}_{P_\ell}$	$\bigcup_{\ell \in L} \mathcal{B}_{P_\ell}$

Fig. 4. Syntax for π-calculus processes with $x, y \in \mathcal{N}$ and $L \subset \mathcal{L}$, and their sets of free and bound names.

α-equivalence

$$\mathsf{Nil} \equiv_\alpha \mathsf{Nil}$$
$$x?(y).P \equiv_\alpha x?(z).P\,[z/y] \quad z \text{ fresh for } P$$
$$x!\langle y\rangle.P \equiv_\alpha x!\langle y\rangle.Q \quad \text{if } P \equiv_\alpha Q$$
$$P \mid Q \equiv_\alpha R \mid S \quad \text{if } P \equiv_\alpha R \text{ and } Q \equiv_\alpha S$$
$$(\nu x)P \equiv_\alpha (\nu u)P\,[u/x] \quad u \text{ fresh for } P$$
$$x \triangleleft \{\ell : P_\ell\}_{\ell \in L} \equiv_\alpha x \triangleleft \{\ell : Q_\ell\}_{\ell \in L} \quad \text{if } P_\ell \equiv_\alpha Q_\ell \text{ for all } \ell \in L$$
$$x \triangleright \{\ell : P_\ell\}_{\ell \in L} \equiv_\alpha x \triangleright \{\ell : Q_\ell\}_{\ell \in L} \quad \text{if } P_\ell \equiv_\alpha Q_\ell \text{ for all } \ell \in L$$

Structural equivalence generators

$$P \mid Q \Leftrightarrows Q \mid P$$
$$(P \mid Q) \mid R \Leftrightarrows P \mid (Q \mid R)$$
$$(\nu x)(\nu y)P \Leftrightarrows (\nu y)(\nu x)P$$
$$P \mid \mathsf{Nil} \Rrightarrow P$$
$$(\nu x)S \Rrightarrow S$$
$$(\nu x)P \mid S \Rrightarrow (\nu x)(P \mid S) \quad \text{with } x \notin \mathsf{free}(S)$$

Fig. 5. The standard α-equivalence, and relations generating of the structural equivalence ($\equiv$) π-calculus processes, where $A \Leftrightarrows B$ stands for $A \Rrightarrow B$ and $B \Rrightarrow A$.

3 Embedding the π-calculus in PiL

In this section we provide an interpretation of π-calculus processes as formulas in PiL, showing also that each successful execution of a process corresponds to a branch in a correct derivation in PiL.

We start by recalling the definition of the π-calculus and its operational semantics. Our presentation has explicit primitives for communicating choices, as usual in the literature of session types [95,96,64]. We then present an alternative semantics in which we use structural precongruence instead of structural equivalence (this is a standard simplification [33,62], which does not affect reasoning about deadlock-freedom, progress, or races). We then provide a translation of processes P into formula $[\![P]\!]$ in PiL and characterise deadlock-freedom for P in terms of provability of $[\![P]\!]$ in PiL.

3.1 The π-calculus and its reduction semantics

The set of π-calculus ***processes*** is generated by the grammar in Figure 4, which uses a fixed countable set of ***(channel) names*** $\mathcal{N} = \{x, y, \ldots\}$ and a finite set of ***labels*** $\mathcal{L}$. We may denote by $(\nu x_1 \ldots x_k)$ a generic sequence $(\nu x_1) \cdots (\nu x_n)$ of ν-constructors of length $n > 0$, and we may simply write $x \triangleleft \{\ell : P_\ell\}$ (resp. $x \triangleright \{\ell : P_\ell\}$) as a shortcut for $x \triangleleft \{\ell : P_\ell\}_{\ell \in L}$ (resp. $x \triangleright \{\ell : P_\ell\}_{\ell \in L}$) whenever $L = \{\ell\}$. A process is ***sequential*** if it contains no parallel ($\mid$) or restrictions (ν), it is ***flat***[9] if of the form $P = (\nu x_1 \ldots x_k)\,(P_1 \mid \cdots \mid P_n)$ for some sequential processes

[9] Sometimes referred to as *non hierarchical* in the literature.

$$\begin{array}{rl|l}
\textsf{Com:} & x!\langle a\rangle.P \mid x?(b).Q \to P \mid Q\,[a/b] & \textsf{Res}: (\nu x)P \to (\nu x)P' \quad \text{if} \quad P \to P' \\
\textsf{Choice:} & x \triangleleft \{\ell : P_\ell\}_{\ell\in L} \to x \triangleleft \{\ell_k : P_{\ell_k}\} \ \text{if}\ \ell_k \in L & \textsf{Par}: \ P \mid Q \to P' \mid Q \quad \text{if} \quad P \to P' \\
\textsf{Label:} & x \triangleleft \{\ell_k : P_{\ell_k}\} \mid x \triangleright \{\ell : Q_\ell\}_{\ell\in L} \to P_{\ell_k} \mid Q_{\ell_k} \ \text{if}\ \ell_k \in L &
\end{array}$$

$$\textsf{Struc}: \quad P \to Q \quad \text{if} \quad P \equiv P' \to Q' \equiv Q$$

Fig. 6. Reduction semantics for the π-calculus.

$P_1, \ldots, P_n$ (also called sequential components of P). We use the common notation $P\,[x/y]$ for substitution (see the Appendix of [9]).

The set $\mathcal{F}_P$ of ***free names*** and the set $\mathcal{B}_P$ of ***bound names*** in a process P are defined in Figure 4. The set of ***names*** in P is denoted $\mathcal{N}_P$ and a name x is ***fresh*** in P if $x \notin \mathcal{N}_P$. A ***context*** is a process $\mathcal{P}[\bullet]$ containing a single occurrence of a special free name $\bullet$ called ***hole*** such that $\mathcal{P}[P] \coloneqq \mathcal{P}\,[P/\bullet]$ is a process. A ***network context*** is a context of the form $\mathcal{N}[\bullet] = (\nu x_1 \ldots x_k)\,([\bullet] \mid P_1 \mid \cdots \mid P_n)$.

The ***α-equivalence*** ($\equiv_\alpha$) is recalled in Figure 5. To improve the presentation of the technical results, we assume processes written in an ***unambiguous*** form, that is, in such a way each bound variable $x \in \mathcal{B}_P$ is bound by a unique ν-constructor and do not occur free in P. In the same figure we provide the relation $\Rrightarrow$, whose reflexive and transitive closure is denoted $\Rrightarrow\!\!\!\!\Rrightarrow$, and we define the standard ***(structural) equivalence*** ($\equiv$) as the equivalence relation generated by the union of $\Rrightarrow$ and $\equiv_\alpha$.

The ***reduction semantics*** for processes is defined by the relation $\to$ over processes induced by the rules in Figure 6. As standard, we denote by $\twoheadrightarrow$ the reflexive and transitive closure of $\to$. As in [22], to allow for nondeterminism, the syntax of processes contains a construct $x \triangleleft \{\ell : P_\ell\}_{\ell\in L}$ allowing for different options rather than the typical $x \triangleleft \{P : \ell\}$. Thus, the corresponding rule Choice for choosing among the available options induces a branching in the computation tree of the process. We say that a process P is ***stuck*** if $P \not\equiv \mathsf{Nil}$ and there is no P' such that $P \to P'$. A process P is called ***deadlock-free*** if there is no stuck process P' such that $P \twoheadrightarrow P'$. Also, a process P has ***progress***[10] if it is deadlock-free or $P \mid Q$ is deadlock-free for a stuck process Q.

Remark 3. Intuitively, deadlock-freedom means that there is always a part of a process that can reduce [62,78]. Progress for processes, instead, was introduced in [29] to characterise processes that get stuck merely because they lack a communicating partner that could be provided by the environment.

For example, the process $P = (\nu x)(x!\langle a\rangle.\mathsf{Nil} \mid x?(b).\mathsf{Nil} \mid y!\langle c\rangle.\mathsf{Nil})$ is not deadlock-free because it reduces (via Com) to the stuck process $(\nu x)(\mathsf{Nil} \mid y!\langle c\rangle.\mathsf{Nil}) \equiv y?(d).\mathsf{Nil}$, but this later has progress since $y!\langle d\rangle.\mathsf{Nil} \mid y?(d).\mathsf{Nil}$ is deadlock-free.

A process P has a ***race condition*** if there is a network context $\mathcal{N}[\bullet]$ such that P is structurally equivalent to a term of the following shape.

$$\begin{array}{cc}
\mathcal{N}[x!\langle y\rangle.R \mid x!\langle z\rangle.Q] & \mathcal{N}[x?(y).R \mid x?(z).Q] \\
\mathcal{N}[x \triangleleft \{\ell : P_\ell\}_{\ell\in L} \mid x \triangleleft \{\ell : P_\ell\}_{\ell\in L'}] & \mathcal{N}[x \triangleright \{\ell : P_\ell\}_{\ell\in L} \mid x \triangleright \{\ell : P_\ell\}_{\ell\in L'}]
\end{array}$$

[10] See Section 1.1 for the precise intended meaning of the term *progress* in this paper.

A process P is ***race-free*** if there is no P' with a race condition such that $P \twoheadrightarrow P'$.

Remark 4. Race conditions identify in a syntactic way the semantic property of a process *potentially* having nondeterministic executions because of concurrent actions on a same channel. For example, $P = x!\langle a\rangle.\mathsf{Nil} \mid x?(b).\mathsf{Nil} \mid x?(c).\mathsf{Nil}$ has a race condition, and it can reduce either to $P_b = x?(b).\mathsf{Nil}$ or to $P_c = x?(c).\mathsf{Nil}$ according to the way the reduction rule Com is applied. We specify 'potentially' because, for example, the process $Q = (\nu x)\,(x?(b).\mathsf{Nil} \mid x?(c).\mathsf{Nil})$ has a race but cannot reduce. In fact, in the execution of a race-free process, rules Com and Label are applied deterministically. That is, the same send (resp. selection) is synchronised via a Com (resp. a Label) with the same receive (resp. branching), and vice versa, in any possible (branch of an) execution.

3.2 A Simpler Equivalent Presentation of the Reduction Semantics

To simplify the presentation of the new methodologies we use in our new framework, we replace the structural equivalence $\equiv$ with the *precongruence* $\Rrightarrow$ (as in [62,33]). In particular, such a precongruence orients the direction of scope extrusion (by extending the scope of the binder as much as possible), but also rules out those rewritings that may add superfluous information such as $P \Rrightarrow (P \mid \mathsf{Nil})$ or $P \Rrightarrow ((\nu x)P)$ for a $x \notin \mathcal{N}_P$. Thus in the reduction semantics we consider in this paper we employ the following rule instead of the standard Struc (see Figure 6):

$$\mathsf{Struc}^{\Rrightarrow} : P \to Q \quad \text{if} \quad P \Rrightarrow P' \;=\; \mathcal{P}[S] \to \mathcal{P}[S'] = Q \quad \text{with } P \neq P' \text{ and } S \to S' \text{ not via } \mathsf{Struc}^{\Rrightarrow} \tag{4}$$

Remark 5. The reduction semantics using the rule $\mathsf{Struc}^{\Rrightarrow}$ instead of Struc is weaker because the set of processes reachable via a step of $\mathsf{Struc}^{\Rrightarrow}$ is strictly contained in the set of processes reachable via Struc. By means of example, consider the process $x!\langle y\rangle.\mathsf{Nil} \mid x?(z).\mathsf{Nil}$ which reduces to both $\mathsf{Nil} \mid \mathsf{Nil}$ and Nil using Struc, but can only reduce to $\mathsf{Nil} \mid \mathsf{Nil}$ using $\mathsf{Struc}^{\Rrightarrow}$.

However, it is immediate to show that if $P \to P'$ via Struc, then there is a $Q \equiv P'$ such that $P \to Q$ via $\mathsf{Struc}^{\Rrightarrow}$. Therefore, the standard reduction semantics (containing the rule Struc) is as informative as the one we consider here (where we use the rule $\mathsf{Struc}^{\Rrightarrow}$ instead) for the study of deadlock-freedom and for the definition of the race condition.

In the definition of the rules of the reduction semantics, the rules Com, Choice and Label are, in some sense, performing 'meaningful' transformation on processes, while rules Res, Par and $\mathsf{Struc}^{\Rrightarrow}$ deal with the syntactic bureaucracy of rewriting modulo the structural equivalence. In the proofs in the next sections we need to be able to identify in each reduction step $P \to P'$ the sub-process $\mathsf{rdx}_{(P,P')}$ of P (called *core-redex*) which is irreversibly transformed to the process $\mathsf{rdt}_{(P,P')}$ (called *core-reductum*), as well as to measure the amount of syntactical manipulations we need to 'reach' such a sub-process to apply a reduction step (which we call *entropy*). We make these concepts precise in the next definitions and exemplify them in Figure 7.

S	S'	$\mathsf{rdx}_{(S,S')}$	$\mathsf{rdt}_{(S,S')}$	$\mathsf{Ent}_{(S,S')}$
$(x!\langle a\rangle.P \mid x?(y).Q) \mid R$	$(P \mid Q\,[a/y]) \mid R$	$x!\langle a\rangle.P \mid x?(y).Q$	$P \mid Q\,[a/y]$	2
$(\nu a)\,(b!\langle a\rangle.P) \mid b?(c).R$	$(\nu a)\,(P \mid R\,[a/c])$	$b!\langle a\rangle.P \mid b?(c).R$	$b!\langle a\rangle.P \mid b?(c).R$	6

Fig. 7. Examples of processes S and S' such that $S \to S'$, and the core-redex, core-reductum, and entropy of the rewriting step.

Definition 1. *Let P and P' processes such that $P \to P'$. The* ***core*** $\mathsf{Core}_{(P,P')} = (\mathsf{rdx}_{(P,P')}, \mathsf{rdt}_{(P,P')})$ *and the* ***entropy*** $\mathsf{Ent}_{(P,P')} \in \mathbb{N}$ *of $P \to P'$ are defined as:*

- *if $P \to P'$ via* Com, Label *or* Choice, *then* $\mathsf{Ent}_{(P,P')} = 1$ *and* $\mathsf{Core}_{(P,P')} = (P, P')$;
- *if $P \to P'$ via* Par *(resp.* Res*), then there are processes Q and Q' such that $Q \to Q'$ and a context $\mathcal{P}[\bullet]$ of the form $\bullet \mid R$ (resp. of the form $(\nu x)(\bullet)$) such that $P = \mathcal{P}[Q]$ and $P' = [Q']$ by definition of the reduction step. Then* $\mathsf{Ent}_{(P,P')} = 2\mathsf{Ent}_{(Q,Q')}$ *and* $\mathsf{Core}_{(P,P')} = \mathsf{Core}_{(Q,Q')}$;
- *if $P \to P'$ via* $\mathsf{Struc}^{\Rrightarrow}$, *then there are processes Q and Q' such that $P \Rrightarrow Q \to Q' \Rrightarrow P'$ with $P \Rrightarrow Q$ and $Q' \Rrightarrow P'$. Then* $\mathsf{Ent}_{(P,P')} = 3\mathsf{Ent}_{(Q,Q')}$ *and* $\mathsf{Core}_{(P,P')} = \mathsf{Core}_{(Q,Q')}$.

The ***core-reduction*** *of $P \to P'$ is the rule used to reduce* $\mathsf{rdx}_{(P,P')}$ *to* $\mathsf{rdt}_{(P,P')}$.

Definition 2. *A* ***execution tree*** *of a process P is a tree of processes* $\mathsf{Ctree}(P)$ *with root P, such that a process Q' is a child of Q if $Q \to Q'$, and such that:*

- *if the core-reduction of $Q \to Q'$ is a* Com *or a* Label, *then Q' is the unique child of Q;*
- *if the core-reduction of $Q \to Q'$ is a* Choice, *then the set $\{Q_1, \ldots, Q_n\} \ni Q'$ of children of Q is such that the core-reduction of $Q \to Q_i$ is a* Choice *and* $\mathsf{rdx}_{(Q,Q_i)} = \mathsf{rdx}_{(Q,Q')}$ *for all $i \in \{1, \ldots, n\}$.*

It is ***maximal*** *if each leaf of the tree is a process $R \equiv \mathsf{Nil}$ or is stuck.*

We conclude this subsection with this result, which, together with Remark 5, allows us to consider each maximal execution tree as a witness of deadlock-freedom for race-free processes.

Lemma 1. *Let P be a process. If P is deadlock-free, then each execution tree with root P can be extended to a maximal execution tree whose leaves are processes structurally equivalent to* Nil.

3.3 Translating Processes into Formulas

We define a translation of π-calculus processes into PiL formulas.

Definition 3 (Processes as Formulas). *We associate to each π-calculus process P a formula $[\![P]\!]$ inductively defined as follows.*

$$
\begin{array}{cccc}
[\![\mathsf{Nil}]\!] = \circ & [\![P \mid Q]\!] = [\![P]\!] \parr [\![Q]\!] & [\![(\nu x)(P)]\!] = ?x.\,[\![P]\!] & \\
[\![\bullet]\!] = \bullet & [\![x!\langle y\rangle.P]\!] = \langle x!y\rangle \blacktriangleleft [\![P]\!] & [\![x?(y).P]\!] = \exists y.((x?y) \blacktriangleleft [\![P]\!]) & \\
[\![x \triangleleft \{\ell : P_\ell\}_{\ell\in L}]\!] = \mathop{\&}_{\ell\in L} (\langle x!\ell\rangle \blacktriangleleft [\![P_\ell]\!]) & & [\![x \triangleright \{\ell : P_\ell\}_{\ell\in L}]\!] = \bigoplus_{\ell\in L} ((x?\ell) \blacktriangleleft [\![P_\ell]\!]) &
\end{array}
\tag{5}
$$

Note that assuming P unambiguous, the translation is a clean formula.

Remark 6. The reader familiar with session types could be curious about the choice of representing by a &-formula a process of the form $x \triangleleft \{\ell : P_\ell\}_{\ell \in L}$ (whose session type is a $\oplus$-type) and, dually, by a $\oplus$-formula a process $x \triangleright \{\ell : P_\ell\}_{\ell \in L}$ (whose session type is a &-type). This is only an apparent contradiction because *our formulas are not types*. Rather, they encode processes whose executions are then derivations in the PiL system. Under this new interpretation, during proof search the rule for & gives exactly the expected branching of possible executions of terms like $x \triangleleft \{\ell : P_\ell\}_{\ell \in L}$, corresponding to rule Choice in the reduction semantics. Rule Label can then be applied 'afterwards' (above in the derivation) to select the appropriate branch at the receiver, discarding all the others. Thus, in the formulas-as-processes, receiving a label corresponds to $\oplus$.

For the same reason, parallel composition is represented by $⅋$ (as in [71,14]), while in most works using propositions as session types it is represented by cut and $\otimes$. We will come back to this aspect in Section 5.

Proposition 2. *Let P_1 and P_2 processes. If $P_1 \Rrightarrow P_2$ then $[\![P_2]\!] \multimap [\![P_1]\!]$.*

Proof. $[\![P \mid Q]\!] \multimap\!\!\circ [\![Q \mid P]\!]$ and $[\![(P \mid Q) \mid R]\!] \multimap\!\!\circ [\![P \mid (Q \mid R)]\!]$ derive from commutativity and associativity of $⅋$ (see Proposition 1). The logical equivalences $[\![P \mid \mathsf{Nil}]\!] \multimap\!\!\circ [\![P]\!]$ and $[\![(\nu x)\mathsf{Nil}]\!] \multimap\!\!\circ [\![\mathsf{Nil}]\!]$ are direct consequence of the ones in Figure 2. The implication $[\![(\nu x)(P \mid Q)]\!] \multimap [\![(\nu x)P \mid Q]\!]$ for $x \notin \mathsf{free}(Q)$ is shown in Proposition 1. Finally, $[\![(\nu x)(\nu y)P]\!] \multimap\!\!\circ [\![(\nu y)(\nu x)P]\!]$ derives from the quantifier shifts $Их.Иу.P = Иу.Их.P$ (Figure 2).

3.4 Deadlock-Freedom as Provability in PiL

We can now establish a correspondence between process reductions and linear implication in PiL, as well as a correspondence between each computation tree with root a process P and a proof search strategy in PiL for the formula $[\![P]\!]$. Combining these two results, we obtain a purely logical characterisation of deadlock-free processes as pre-images via $[\![\cdot]\!]$ of formulas derivable in PiL.

Lemma 2. *Let P and P' processes.*

1. *If $P \Rrightarrow P'$, then $[\![P']\!] \multimap [\![P]\!]$.*
2. *If $P \to P'$, then either*
   (a) the core-reduction of $P \to P'$ is a Com *or a* Label*, and $\vdash_{\mathsf{PiL}} [\![P']\!] \multimap [\![P]\!]$;*
   (b) or the core-reduction of $P \to P'$ is a Choice *then there is a set $\{P_\ell \mid \ell \in L\} \ni P'$ such that $P \to P_\ell$ for all $\ell \in L$ and $\vdash_{\mathsf{PiL}} (\&_{\ell \in L} [\![P_\ell]\!]) \multimap [\![P]\!]$.*

Proof. Item 1 is proven using Proposition 1 and transitivity of $\multimap$ (see Theorem 1.3). To prove Item 2 we reason by induction on entropy:

– if $\mathsf{Ent}_{(P,P')} = 1$ then $P \to P'$ via Com, Label or Choice and we conclude using the derivations in Figure 8;

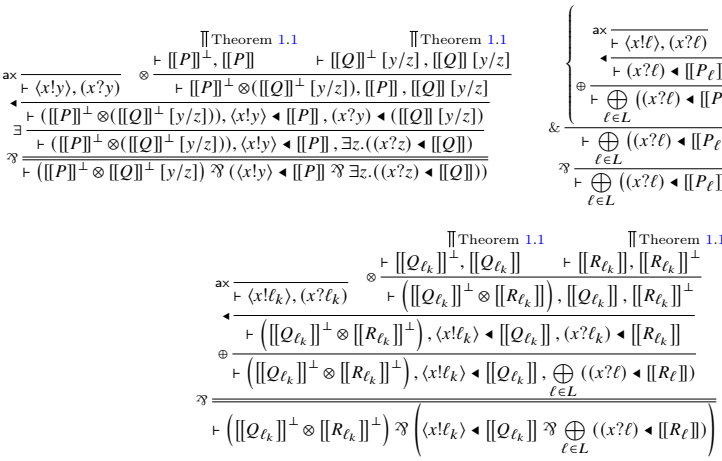


Fig. 8. Derivations in PiL corresponding to the rules Com, Choice and Label of the reduction semantics of the π-calculus.

- if $P \to P'$ via Par (resp. Res), then there is a context $\mathcal{P}[\bullet] = (\bullet \mid R)$ (resp. $\mathcal{P}[\bullet] = (\nu x)\bullet$) such that $P = \mathcal{P}[S]$ and $P' = \mathcal{P}[S']$. We conclude using Theorem 2.1 and Theorem 1.3;
- if $P \to P'$ via $\mathsf{Struc}^{\Rrightarrow}$, then there is S such that $P \Rrightarrow S$ and $S \to P'$ (via a rule different from $\mathsf{Struc}^{\Rrightarrow}$). We conclude by Theorem 1.3 using Item 1 and Theorem 22.

Using this lemma we can prove the correspondence between deadlock-freedom and derivability in PiL.

Theorem 3. *Let P be a race-free process. Then P is deadlock-free iff* $\vdash_{\mathsf{PiL}} [\![P]\!]$. *More precisely,* $\vdash_{\{\circ,\mathsf{ax},\parr,\mathsf{mix},\blacktriangleleft,\&,\oplus,\exists,И^\circ\}} [\![P]\!]$.

Proof. It suffices to establish a correspondence between maximal execution trees and derivations in PiL. Details are provided in the appendix of [9].

($\Rightarrow$) If P is deadlock-free, then, any maximal execution tree $\mathsf{Ctree}(P)$ with root P has leaves which are processes structurally equivalent to Nil by Lemma 1. By induction on the structure of $\mathsf{Ctree}(P)$, we can define a derivation in PiL$\cup\{\mathsf{cut}\}$ composing (using cut) the derivations allowing simulating the transitions of the reduction semantics (Lemma 2); thus we obtain a derivation in PiL by applying cut-elimination (Theorem 1.2). We conclude observing that the subformula property ensures that only rules in such cut-free derivation are in $\{\circ, \mathsf{ax}, \parr, \mathsf{mix}, \blacktriangleleft, \&, \oplus, \exists, И^\circ\}$. Note that all judgements in such a derivation are empty.

($\Leftarrow$) To prove the converse, we show that each derivation $\mathcal{D}_P$ of $[\![P]\!]$ can be transformed using the *rule permutations* in Figure 9 into a derivation $\widetilde{\mathcal{D}_P}$ made of blocks of rules consisting of sequences of $И$- and $\parr$-rules only, or blocks as the

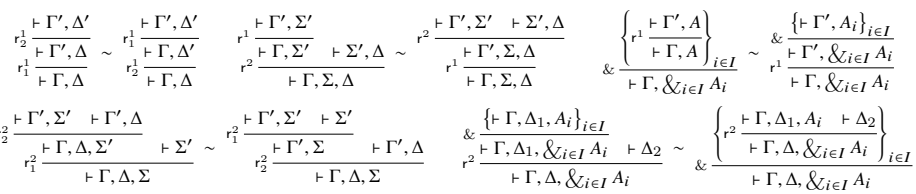
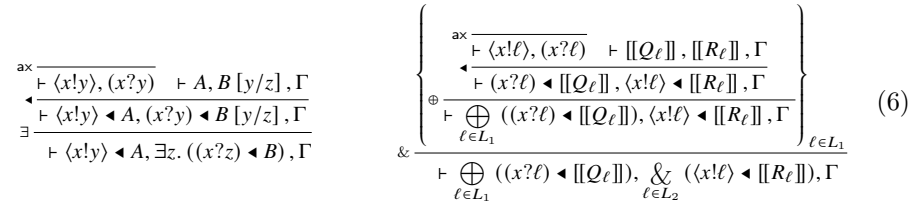


Fig. 9. Rule permutations with $\mathsf{r}^1, \mathsf{r}^1_1, \mathsf{r}^1_2 \in \{⅋, \exists, \oplus, И_\circ\}$ and $\mathsf{r}^2, \mathsf{r}^2_1, \mathsf{r}^2_2 \in \{◂, \otimes, \mathsf{mix}\}$.

ones shown in Equation (6) below.



(6)

We conclude by induction on the number of such blocks, since each block in the left (resp. right) of Equation (6) identifies an application of a Com (resp. a Bra followed by a Sel).

Note that since P is race-free, then it suffices to reason on a single execution tree and not to take into account all possible execution trees of P.

Corollary 1. *Let P be a race-free process. Then P has progress iff there is a $И⅋$-context $C[\bullet]$ such that $\vdash_{\mathsf{PiL}} C[P]$.*

We conclude this section by showing that progress for processes which never send 'private' channels can be easily captured in this new setting. Specifically, we say that a process P has ***private mobility*** if it is of the form $P = \mathcal{P}[a!\langle x\rangle]$ for an a bound by a ν in $\mathcal{P}$. We also denote by $\partial_{x_1,\ldots,x_k}\,[\![P]\!]$ the formula obtained by replacing with a unit ($\circ$) any atom in $[\![P]\!]$ of the form $\langle x!y\rangle$ or $(x?y)$ for any $x \in \{x_1, \ldots, x_k\}$.

Theorem 4. *Let P be a race-free process without private mobility. Then P has progress iff $\vdash_{\mathsf{PiL}} \partial_{\mathcal{F}_P}\,[\![P]\!]$.*

Proof. We prove a simulation result (as Lemma 2) for $\partial_{\mathcal{F}_P}\,[\![P]\!]$, and we conclude with the same argument used in the proof of Theorem 3. It P is deadlock-free, then we conclude as in Theorem 3. Otherwise, since P has progress there is Q such that $P \mid Q$ is deadlock-free. By definition, we must have that $\mathcal{N}_Q = \mathcal{F}_Q$ (otherwise either Q is not stuck or $P \mid Q$ is not deadlock-free) and that $\mathcal{F}_Q = \mathcal{F}_P = x_1, \ldots, x_k$. Thus $\partial_{x_1,\ldots,x_k}\,[\![P \mid Q]\!] \multimap \partial_{x_1,\ldots,x_k}\,[\![P]\!]$ by the fact that $\partial_{x_1,\ldots,x_k}\,[\![Q]\!]$ contains no atoms (i.e., only units) and $\circ \blacktriangleleft A \multimap A \multimap \circ ⅋ A$ (see Proposition 1).

– If $P \mid Q$ is deadlock-free for Q stuck and $P \mid Q \to R$ via Par then, $R = P' \mid Q$ and $P \to P'$. If the core-reduction of $P \twoheadrightarrow P'$ is a Com or a Label, then

$\vdash_{\mathsf{PiL}} \partial_{x_1,\ldots,x_k}[\![P']\!] \multimap \partial_{x_1,\ldots,x_k}[\![P]\!]$; if the core-reduction is a Choice, there is a set of processes $\{P_\ell\}_{\ell\in L} \ni P'$ such that $\vdash_{\mathsf{PiL}} (\&_{\ell\in L}\, \partial_{x_1,\ldots,x_k}[\![P_\ell]\!]) \multimap \partial_{x_1,\ldots,x_k}[\![P]\!]$. This is proven by induction on the entropy as in Lemma 2.
- If $P' \mid Q \to P'$ not via Par and then the core-reduction is either a Com or a Sel. In this case can prove as that $\vdash_{\mathsf{PiL}} \partial_{x_1,\ldots,x_k}[\![P']\!] \multimap \partial_{x_1,\ldots,x_k}[\![P']\!]$ because $\partial_{x_1,\ldots,x_k}[\![P']\!] \multimap\!\!\circ \partial_{x_1,\ldots,x_k}[\![P' \mid Q]\!] \multimap\!\!\circ \partial_{x_1,\ldots,x_k}[\![P']\!]$.

Remark 7. To understand the requirement on private mobility in Theorem 4, consider the process $P = (\nu a)(b!\langle a\rangle.a!\langle c\rangle.\mathsf{Nil})$. This process has progress, because

$$(P \mid b?(x).x?(c).\mathsf{Nil}) \equiv (\nu a)(b!\langle a\rangle.a!\langle c\rangle.\mathsf{Nil} \mid b?(x).x?(c).\mathsf{Nil}) \twoheadrightarrow \mathsf{Nil}\ .$$

However, $\partial_{\{b\}}[\![P]\!] = И a.(\circ \blacktriangleleft \langle a!c\rangle \blacktriangleleft \circ)$ is not derivable in PiL. This makes our characterisation of progress as powerful as in previous work [22] (where the condition is not made explicit but clearly necessary, see the definition of 'co-process' therein).

4 Completeness of Choreographies

In this section we prove that any deadlock-free flat process can be expressed as a choreography, as intended in the paradigm of choreographic programming [77]. Key to this result is establishing a *proofs-as-choreographies* correspondence, whereby choreographies can be seen as derivations in the PiL system.

To this end, we first introduce the syntax and semantics of *choreographies*, the typical accompanying language for describing their implementations in terms of located processes (the *endpoint calculus*), and a notion of endpoint projection (EPP) from choreographies to processes. We then define the sequent calculus ChorL operating on sequents in which (occurrences of) formulas are labelled by process names, and we conclude by establishing the proofs-as-choreographies correspondence.

4.1 Choreographies

In a choreographic language, terms (called *choreographies*) are coordination plans that express the overall behaviour of a network of processes [78]. The ***choreographies*** that we consider in this paper are generated by a set of ***process names*** $\mathcal{P}$, a set of variables $\mathcal{V}$, and a set of ***selection labels*** $\mathcal{L}$ as shown in Figure 10. A choreography can be either:

- $\mathbf{0}$, the terminated choreography;
- $\mathsf{p}.x \twoheadrightarrow \mathsf{q}.y : k; C$, a communication from a process p to another q with a continuation C (y is bound in C and can appear only under q);
- $\mathsf{p}.L \twoheadrightarrow \mathsf{q}.L' : k \left\{ \begin{array}{l|l} \ell : C_\ell & \ell \in L \\ \ell : S_\ell & \ell \in L' \setminus L \end{array} \right\}$, a choice by a process p of a particular branch L offered by another process q[11]; or

[11] The set L' of labels the process q can accept contains the set L of labels p can send. For the continuation of labels in $L' \setminus L$ we only allow sequential processes because

Choreographies

$$C, C_\ell \coloneqq \mathbf{0} \mid \mathsf{p}.x \to \mathsf{q}.y : k; C \mid \mathsf{p}.L \to \mathsf{q}.L' : k \left\{ \begin{array}{l|l} \ell : C_\ell & \ell \in L \\ \ell : S_\ell & \ell \in L' \setminus L \end{array} \right\} \mid (\nu x) C^x \text{ (with } C^x \text{ containing no } (\nu x))$$

end | communication | choice | restriction

Reduction semantics for Choreographies

$$\textsf{Com}: \quad \mathsf{p}.x \to \mathsf{q}.y : k; C \xrightarrow{\mathsf{p} \to \mathsf{q} : k} C\,[x/y]$$

$$\textsf{Choice}: \quad \mathsf{p}.L \to \mathsf{q}.L' : k \left\{ \begin{array}{l|l} \ell : C_\ell & \ell \in L \\ \ell : S_\ell & \ell \in L' \setminus L \end{array} \right\} \xrightarrow{\mathsf{p} : k} \mathsf{p}.\{\ell_i\} \to \mathsf{q}.L' : k \left\{ \begin{array}{l|l} \ell : C_\ell & \ell \in L \\ \ell : S_\ell & \ell \in L' \setminus L \end{array} \right\} \text{ for a } \ell_i \in L$$

$$\textsf{Label}: \quad \mathsf{p}.\{\ell_i\} \to \mathsf{q}.L' : k \left\{ \begin{array}{l|l} \ell : C_\ell & \ell \in L \\ \ell : S_\ell & \ell \in L' \setminus L \end{array} \right\} \xrightarrow{\mathsf{p} \to \mathsf{q} : k} C_{\ell_i} \quad \text{if } \ell_i \in L'$$

$$\textsf{Rest}: \quad (\nu x) C \xrightarrow{\mu} (\nu x) C' \quad \text{if } C \xrightarrow{\mu} C'$$

$$\textsf{D-Com}: \quad \mathsf{p}.x \to \mathsf{q}.y : k; C \xrightarrow{\mu'} \mathsf{p}.x \to \mathsf{q}.y : k; C' \quad \text{if } C \xrightarrow{\mu'} C'$$

$$\textsf{D-Choice}: \quad \mathsf{p}.L \to \mathsf{q}.L' : k \left\{ \begin{array}{l|l} \ell : C_\ell & \ell \in L \\ \ell : S_\ell & \ell \in L' \setminus L \end{array} \right\} \xrightarrow{\mu'} \mathsf{p}.L \to \mathsf{q}.L' : k \left\{ \begin{array}{l|l} \ell : C'_\ell & \ell \in L \\ \ell : S_\ell & \ell \in L' \setminus L \end{array} \right\} \quad \text{if } C_\ell \xrightarrow{\mu'} C'_\ell \text{ for all } \ell \in L$$

with $\mathsf{pn}(\mu') \cap \{\mathsf{p}, \mathsf{q}, k\} = \varnothing$

Fig. 10. Syntax and semantics for choreographies, where p and q are distinct process names in $\mathcal{P}$, $x, y \in \mathcal{V}$, $L \subseteq L' \subseteq \mathcal{L}$, and S_ℓ are sequential processes (see Remark 8).

– $(\nu x)C^x$, which restricts x in a choreography C^x in which the variable x always occur free (i.e., no (νx) occurs in C^x).[12]

Note that we consider communication of process names or variables only (that is, $k \in \mathcal{P} \cup \mathcal{V}$). We say that a choreography is ***flat*** if it is of the form $(\nu x_1 \dots x_k)C^{\mathsf{rf}}$ for a ***restriction-free*** (i.e., containing no occurrences of ν) choreography C^{rf}. In the same figure we also provide the ***reduction semantics*** of our choreographic language, where each reduction step is labelled by a ***reduction label*** μ from the following set.

$$\{\mathsf{p} \to \mathsf{q} : k \ , \ \mathsf{p} : k \mid \mathsf{p}, \mathsf{q} \in \mathcal{P}, \ k \in \mathcal{V}\} \tag{7}$$

To each reduction label μ we associate the set $\mathsf{pn}(\mu)$ of processes names and variables occurring in it – i.e. $\mathsf{pn}(\mathsf{p} \to \mathsf{q} : k) = \{\mathsf{p}, \mathsf{q}, k\}$ and $\mathsf{pn}(\mathsf{p} : k) = \{\mathsf{p}, k\}$.

In the semantics, Com executes a communication while Choice allows a process p to make an internal choice. Rule Label then communicates a label from p to q, which then continue with the choreography C_ℓ (but never with a sequential process S_ℓ, see Remark 8). Rule Rest lifts reductions under restrictions. Lastly, D-Com (resp. D-Choice) models the standard out-of-order execution of independent communications that can be reduced by rule Com (resp. both rules Choice and Label) – this is the choreographic equivalent of parallel composition in process calculi [78].

we do not allow nested parallel in the target language of the projection (see next subsection).

[12] By allowing the construct (νx) only in the case in which x is not bound in C^x, we ensure that choreographies are always written using *Barendregt's convention*. This means that each variable x can be bound by at most one restriction ν, and x cannot appear both free and bound in a choreography C. As a result, we can adopt a lighter labelling discipline for the reduction semantics compared to the one used in [25] – see the rule Rest in Figure 10.

Structural Equivalence

$$\mathsf{p} :: \mathsf{Nil} \mid P \equiv P \quad \text{and} \quad (\nu x_1)\cdots(\nu x_k)\prod_{i=1}^{n} \mathsf{p}_i :: S_i \equiv (\nu x_{\tau(1)})\cdot(\nu x_{\tau(k)})\prod_{i=1}^{n} \mathsf{p}_{\sigma(i)} :: S_{\sigma(i)}$$

for any σ permutation over $\{1,\ldots,n\}$ and τ over $\{1,\ldots,k\}$

Reduction Semantics

$$\textsf{E-Com}: \quad \mathcal{N}[\mathsf{p} :: k!\langle x\rangle.S \mid \mathsf{q} :: k?(y).S'] \xrightarrow{\mathsf{p}\to\mathsf{q}:k} \mathcal{N}[\mathsf{p} :: S \mid \mathsf{q} :: S'[x/y]]$$

$$\textsf{E-Choice}: \quad \mathcal{N}\left[\mathsf{p} :: k \triangleleft \{\ell : S_\ell\}_{\ell\in L}\right] \xrightarrow{\mathsf{p}:k} \mathcal{N}\left[\mathsf{p} :: k \triangleleft \{\ell_i : S_{\ell_i}\}\right] \quad \text{for each } \ell_i \in L$$

$$\textsf{E-Label}: \mathcal{N}\left[\mathsf{p} :: k \triangleleft \{\ell_i : S_{\ell_i}\} \mid \mathsf{q} :: k \triangleright \left\{\ell : S'_\ell\right\}_{\ell\in L}\right] \xrightarrow{\mathsf{p}\to\mathsf{q}:k} \mathcal{N}\left[\mathsf{p} :: S_{\ell_i} \mid \mathsf{q} :: S'_{\ell_i}\right] \quad \text{if } \ell_i \in L$$

Fig. 11. Simplified presentation of the structural equivalence and reduction semantics for the endpoint calculus, where $\mathcal{N}[P] \equiv (\nu x_1 \ldots x_k)\left(P \mid \prod_{i=1}^{n} \mathsf{p}_i :: T_i\right)$.

Example 1. The next choreography expresses the communication behaviour of the processes given in Equation (1).

$$\mathsf{p}.a \to \mathsf{q}.a : x; \mathsf{p}.\{\ell\} \to \mathsf{q}.\{\ell, \ell'\} : y\begin{Bmatrix} \ell : \mathsf{p}.b \to \mathsf{q}.b : y; \mathbf{0} \\ \ell' : z!\langle c\rangle.\mathsf{Nil} \end{Bmatrix} \tag{8}$$

It can be executed by applying rule Com and then rule Label. Note that we do not need to use rule Choice before applying Label, because the set of labels L in the choice constructor is a singleton.

Remark 8. From the programmer's viewpoint, choice instructions may contain some unnecessary information since no label $\ell' \in L' \setminus L$ will never be selected during the execution of a choreography – and thus no continuation will execute the process $S_{\ell'}$. This 'garbage' code is typical of works on choreographies and logic [26,21], and we share the same motivation: we want to be able to capture the entire flat fragment of the π-calculus, where such garbage code cannot be prohibited. For example, without garbage code, the choreography in Equation (8) would not be a complete representation of the endpoint process in Equation (10) (see also Equation (1)).

4.2 Endpoint Projection

Our choreographies can be mechanically translated into processes via the standard technique of endpoint projection (EPP) [78]. To simplify the presentation of projection, we adopt the standard convention of enriching the language of processes with process names labelling each sequential component of a flat process [50,78]. That is, a flat process of the form $(\nu x_1 \ldots x_k)\,(S_1 \mid \cdots \mid S_n)$ is represented by an ***endpoint process***

$$(\nu x_1 \ldots x_k)\,(\mathsf{p}_1 :: S_1 \mid \cdots \mid \mathsf{p}_n :: S_n) \quad \text{or} \quad (\nu x_1 \ldots x_k)\left(\prod_{i=1}^{n} \mathsf{p}_i :: S_i\right) \tag{9}$$

where all names $\mathsf{p}_1, \ldots, \mathsf{p}_n$ are distinct.[13] The process calculus over these processes is dubbed ***endpoint calculus***.

[13] In the literature, a process P with name p is usually written $\mathsf{p}[P]$ [50,78]. We adopt the alternative writing $\mathsf{p} :: P$ to avoid confusion with the notation used for contexts.

The definition of ***endpoint projection*** is provided by the partial function EPP in Figure 12 (it is defined only for flat choreographies, as in [25]). It is a straightforward adaptation to our syntax of the textbook presentation of projection [78]. In particular, it uses a ***merge*** operator $\sqcup$ (originally from [23]) to support propagation of knowledge about choices. That is, if a process r needs to behave differently in two branches of a choice communicated from p to q, it can do so by receiving different labels in these two branches. Merge then produces a term for r that behaves as prescribed by the first (respectively the second) branch when it receives the first (respectively the second) label. If $\mathsf{EPP}\,(C)$ is defined for C we say that C is ***projectable***.

Example 2. The EPP of the choreography in Equation (8) is

$$(\nu x)(\nu y)\,(\mathsf{p} :: x!\langle a\rangle.y \triangleleft \{\ell : y!\langle b\rangle.\mathsf{Nil}\} \mid \mathsf{q} :: x?(a).y \triangleright \{\ell : y?(b).\mathsf{Nil}, \ell' : z!\langle c\rangle.\mathsf{Nil}\}) \quad (10)$$

which is precisely the one in Equation (1) annotated with process names.

Structural equivalence and reduction semantics for the endpoint calculus are obtained by the one of the π-calculus assuming that each structural equivalence ($\equiv$) and reduction step ($\rightarrow$) preserves the process names. Note that, for the purpose of studying deadlock-freedom, structural equivalence and reduction rules can be simplified as shown in Figure 11. Each reduction step is labelled by the same labels used in the reduction semantics of choreographies (see Equation (7)), allowing us to retain the information about which sequential components and channel are involved in each reduction step.

Notation 2. *If P and Q are endpoint processes, we write $P \sqsupseteq Q$ iff $P \sqcup Q = P$.*

Theorem 5. *Let C be a projectable flat choreography.*

- ***Completeness****: if $C \xrightarrow{\mu} C'$, then $\mathsf{EPP}\,(C) \xrightarrow{\mu} P \sqsupseteq \mathsf{EPP}\,(C')$;*
- ***Soundness****: if $P \sqsupseteq \mathsf{EPP}\,(C)$ and $P \xrightarrow{\mu} P'$, then there is a choreography C' such that $C \xrightarrow{\mu} C'$ and $P' \sqsupseteq \mathsf{EPP}\,(C')$.*

Proof. The proof is obtained by adapting the proof provided in, e.g., [77,78,79] to the language we consider in this paper. Details can be found in [4].

4.3 A Sequent Calculus for the Endpoint Calculus

To establish the correspondence between proofs of (formulas encoding) endpoint processes and choreographies, we enrich the syntax of formulas by adding labels (on sub-formulas) carrying the same information of the process names used in the syntax of the endpoint calculus. More precisely, we consider a translation $[\![\cdot]\!]^{\mathcal{N}}$ from endpoint processes to ***annotated formulas*** of the form $(A)_{\mathsf{p}}$.

Definition 4. *For any endpoint process $P = (\nu x_1 \dots x_k)\left(\prod_{i=1}^{n} \mathsf{p}_i :: S_i\right)$ we define the formula $[\![P]\!]^{\mathcal{N}} = И x_1 \dots И x_n.([\![S_1]\!])_{\mathsf{p}_1} \parr \cdots \parr ([\![S_n]\!])_{\mathsf{p}_n}$, and the sequent $\llbracket P \rrbracket = ([\![S_1]\!])_{\mathsf{p}_1}, \dots, ([\![S_n]\!])_{\mathsf{p}_n}$.*

$$\mathsf{EPP}\,(C) = \begin{cases} (\nu x_1 \dots x_k)\mathsf{EPP}\left(C^{\mathsf{rf}}\right) & \text{if } C = (\nu x_1 \dots x_k)C^{\mathsf{rf}} \text{ with } C^{\mathsf{rf}} \text{ restriction-free} \\ \mathsf{p}_0 :: \mathsf{Nil} & \text{if } C = \mathbf{0} \text{ (for a given } \mathsf{p}_0) \\ \mathsf{EPP}_{\mathsf{p}_1}\,(C) \mid \dots \mid \mathsf{EPP}_{\mathsf{p}_n}\,(C) & \text{otherwise, with } \mathsf{p}_1, \dots, \mathsf{p}_n \text{ all process names in } C \end{cases}$$

$$\mathsf{EPP}_{\mathsf{p}_i}\,(\mathbf{0}) = \mathsf{Nil} \qquad \mathsf{EPP}_{\mathsf{p}_i}\,(S) = \begin{cases} \mathsf{EPP}_{\mathsf{p}_i}\,(S) & \text{if } S \text{ is a choreography} \\ \mathsf{EPP}_{\mathsf{p}_i}\,(\mathsf{p}_i :: S_i) & \text{if } S = \prod_{i=1}^{n} \mathsf{p}_i :: S_i \text{ is a process} \end{cases}$$

$$\mathsf{EPP}_{\mathsf{p}_i}\,(\mathsf{p}.x \to \mathsf{q}.y : k; C) = \begin{cases} k!\langle x\rangle.\mathsf{EPP}_{\mathsf{p}_i}\,(C) & \text{if } \mathsf{p}_i = \mathsf{p} \\ k?(y).\mathsf{EPP}_{\mathsf{p}_i}\,(C) & \text{if } \mathsf{p}_i = \mathsf{q} \\ \mathsf{EPP}_{\mathsf{p}_i}\,(C) & \text{if } \mathsf{p}_i \notin \{\mathsf{p}, \mathsf{q}\} \end{cases}$$

$$\mathsf{EPP}_{\mathsf{p}_i}\left(\mathsf{p}.L \to \mathsf{q}.L' : k \left\{ \begin{array}{l|l} \ell : C_\ell & \ell \in L \\ \ell : S_\ell & \ell \in L' \setminus L \end{array} \right\}\right) = \begin{cases} k \triangleleft \{\ell : \mathsf{EPP}_{\mathsf{p}_i}\,(C_\ell)\}_{\ell \in L} & \text{if } \mathsf{p}_i = \mathsf{p} \\ k \triangleright \left\{ \begin{array}{l|l} \ell : \mathsf{EPP}_{\mathsf{p}_i}\,(C_\ell) & \ell \in L \\ \ell : S_\ell & \ell \in L' \setminus L \end{array} \right\} & \text{if } \mathsf{p}_i = \mathsf{q} \\ \bigsqcup_{\ell \in L} \mathsf{EPP}_{\mathsf{p}_i}\,(C_\ell) & \text{if } \mathsf{p}_i \notin \{\mathsf{p}, \mathsf{q}\} \end{cases}$$

$$\left((\nu x_1 \dots x_k)\left(\textstyle\prod_{i=1}^{n} \mathsf{p}_i :: S_i\right)\right) \sqcup \left((\nu x_1 \dots x_k)\left(\textstyle\prod_{i=1}^{n} \mathsf{p}_i :: T_i\right)\right) = (\nu x_1 \dots x_k)\left(\textstyle\prod_{i=1}^{n} \mathsf{p}_i :: S_i \sqcup T_i\right)$$

$$\mathsf{Nil} \sqcup \mathsf{Nil} = \mathsf{Nil} \qquad (x!\langle y\rangle.T) \sqcup (x!\langle y\rangle.S) = x!\langle y\rangle.T \sqcup S \qquad (x?(y).T) \sqcup (x?(y).S) = x?(y).T \sqcup S$$

$$(x \triangleright \{\ell : P_\ell\}_{\ell \in L}) \sqcup (x \triangleright \{\ell : Q_\ell\}_{\ell \in L'}) = x \triangleright \left(\{\ell : P_\ell \sqcup Q_\ell\}_{\ell \in L \cap L'} \cup \{\ell : P_\ell\}_{\ell \in L \setminus L'} \cup \{\ell : Q_\ell\}_{\ell \in L' \setminus L}\right)$$

$$\text{only if } L = L' : \ (x \triangleleft \{\ell : P_\ell\}_{\ell \in L}) \sqcup (x \triangleleft \{\ell : Q_\ell\}_{\ell \in L'}) = x \triangleleft \{\ell : P_\ell \sqcup Q_\ell\}_{\ell \in L}$$

Fig. 12. Endpoint Projection for flat choreographies, and the merge operator ($\sqcup$).

Note that because of the subformula property[14] of rules in PiL, such labelling can be propagated in a derivation by labelling each active formula of a rule (which is a sub-formula of one of the principal formulas of the rule) with the same process name of the corresponding active formula.

Example 3. Consider the following derivation in PiL with conclusion the formula $[[\mathsf{p} :: x!\langle y\rangle.\mathsf{Nil} \mid \mathsf{q} :: x?(y).\mathsf{Nil}]]^{\mathcal{N}}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash (\langle x!y\rangle)_{\mathsf{p}}, ((x?y))_{\mathsf{q}}}\,\mathsf{ax} \quad \dfrac{\dfrac{}{\vdash (\circ)_{\mathsf{p}}}\circ \quad \dfrac{}{\vdash (\circ)_{\mathsf{q}}}\circ}{\vdash (\circ)_{\mathsf{p}}, (\circ)_{\mathsf{q}}}\,\mathsf{mix}}{\vdash (\langle x!y\rangle \blacktriangleleft \circ)_{\mathsf{p}}, ((x?y) \blacktriangleleft \circ)_{\mathsf{q}}}\blacktriangleleft}{\vdash (\langle x!y\rangle \blacktriangleleft \circ)_{\mathsf{p}}, (\exists x.((x?y) \blacktriangleleft \circ))_{\mathsf{q}}}\,\exists}{\vdash (\langle x!y\rangle \blacktriangleleft \circ)_{\mathsf{p}} \parr (\exists x.((x?y) \blacktriangleleft \circ))_{\mathsf{q}}}\,\parr \tag{11}$$

We now introduce a sequent calculus for the endpoint calculus, given in Figure 13, which consists purely of rules that are derivable in PiL. That is, for every rule r in ChorL there is an open derivation in PiL with the same open premises and conclusion as r.

[14] Assuming an initial α-renaming on P such that P is unambiguous, and such each variable bound by a receive action is the same as the unique (because of race-freedom) variable sent by the matching send action. For example, we would write $x!\langle a\rangle \mid x?(a)$ instead of $x!\langle a\rangle \mid x?(b)$.

$$C\text{-flat}\ \frac{\vdash ([\![T_1]\!])_{\mathsf{p}_1}, \ldots, ([\![T_n]\!])_{\mathsf{p}_n}}{\vdash И x_1 \ldots И x_m. \left(([\![T_1]\!])_{\mathsf{p}_1} \parr \cdots \parr ([\![T_n]\!])_{\mathsf{p}_n}\right)}\ m \in \mathbb{N} \qquad C\text{-Com}\ \frac{\vdash \Gamma, ([\![T]\!])_{\mathsf{p}}, ([\![T'\ [y/z]]\!])_{\mathsf{q}}}{\vdash \Gamma, ([\![x?(z).T']\!])_{\mathsf{q}}}$$

$$C\text{-Init}\ \frac{}{\vdash (\circ)_{\mathsf{p}_1}, \ldots, (\circ)_{\mathsf{p}_n}} \qquad C\text{-Sel}\ \frac{\left\{\vdash \Gamma, ([\![T_\ell]\!])_{\mathsf{p}}, \left(\left[\!\left[T'_\ell\right]\!\right]\right)_{\mathsf{q}}\right\}_{\ell \in L}}{\vdash \Gamma, \left([\![k \triangleleft \{\ell : T_\ell\}_{\ell \in L}]\!]\right)_{\mathsf{p}}, \left(\left[\!\left[k \triangleright \left\{\ell : T'_\ell\right\}_{\ell \in L'}\right]\!\right]\right)_{\mathsf{q}}}\ L \subseteq L'$$

Fig. 13. Sequent calculus rules for the system ChorL.

C-Init
$$\mathcal{D}_n = \mathrm{mix}\ \frac{\overline{\vdash (\circ)_{\mathsf{p}_1}} \quad \cdots \quad \overline{\vdash (\circ)_{\mathsf{p}_n}}}{\vdash (\circ)_{\mathsf{p}_1}, \ldots, (\circ)_{\mathsf{p}_n}}$$

C-Com
$$\mathcal{D}_{\langle \mathsf{p},x,\mathsf{q},y,k\rangle} = \blacktriangleleft\ \frac{\mathrm{ax}\ \overline{\vdash (\langle k!x\rangle)_{\mathsf{p}}, ((k?x))_{\mathsf{q}}} \qquad \vdash \Gamma, ([\![S]\!])_{\mathsf{p}}, ([\![S'\ [x/y]]\!])_{\mathsf{q}}}{\exists\ \frac{\vdash \Gamma, (\langle k!x\rangle \blacktriangleleft [\![S]\!])_{\mathsf{p}}, ((k?x) \blacktriangleleft ([\![S'\ [x/y]]\!]))_{\mathsf{q}}}{\vdash \Gamma, (\langle k!x\rangle \blacktriangleleft [\![S]\!])_{\mathsf{p}}, (\exists y.((k?y) \blacktriangleleft [\![S']\!]))_{\mathsf{q}}}}$$

C-Sel
$$\mathcal{D}_{\langle \mathsf{p},L,\mathsf{q},L',k\rangle} = \&\ \frac{\left\{\blacktriangleleft\ \frac{\mathrm{ax}\ \overline{\vdash (\langle k!\ell\rangle)_{\mathsf{p}}, ((k?\ell))_{\mathsf{q}}} \qquad \vdash ([\![S_\ell]\!])_{\mathsf{p}}, \left(\left[\!\left[S'_\ell\right]\!\right]\right)_{\mathsf{q}}}{\vdash \Gamma, (\langle k!\ell\rangle \blacktriangleleft [\![S_\ell]\!])_{\mathsf{p}}, \left((k?\ell) \blacktriangleleft \left[\!\left[S'_\ell\right]\!\right]\right)_{\mathsf{q}}}\right\}_{\ell \in L}}{\vdash \Gamma, \left(\&_{\ell \in L}\ (\langle k!\ell\rangle \blacktriangleleft [\![S_\ell]\!])\right)_{\mathsf{p}}, \left(\oplus_{\ell \in L'}\ \left((k?\ell) \blacktriangleleft \left[\!\left[S'_\ell\right]\!\right]\right)\right)_{\mathsf{q}}}$$

C-flat
$$\mathcal{D}_\varnothing = \parr\ \frac{\vdash [\![S_1]\!], \ldots, [\![S_n]\!]}{\vdash [\![S_1]\!] \parr \ldots \parr [\![S_n]\!]} \quad \text{or} \quad \mathcal{D}_{\langle x_1,\ldots,x_m\rangle} = И\ \frac{\parr\ \frac{\vdash [\![S_1]\!], \ldots, [\![S_n]\!]}{\vdash [\![S_1]\!] \parr \ldots \parr [\![S_n]\!]}}{\vdash И x_1 \ldots И x_m. ([\![S_1]\!] \parr \ldots \parr [\![S_n]\!])}$$

Fig. 14. Derivability (in PiL) of the rules in ChorL.

Lemma 3. *Each rule in* ChorL *is derivable in* PiL.

Moreover, if the premise (resp. conclusion) of a rule in ChorL *is of the form* $[\![P]\!]^{\mathcal{N}}$*, then its conclusion is so (resp. all its premise are so).*

Proof. Derivations with the same premise(s) and conclusion of a rule in ChorL are shown in Figure 14. The second part of the statement follows by rule inspection.

We can refine Theorem 3 for race-free flat processes thanks to the fact that in endpoint processes parallel and restrictions can only occur at the top level. This allows us to consider derivations in PiL made of blocks of rule applications as those in Figure 14, each corresponding to a single instance of a rule in ChorL.

Theorem 6. *A race-free endpoint process P is deadlock-free iff* $\vdash_{\mathsf{ChorL}} [\![P]\!]^{\mathcal{N}}$.

Proof. If $P = (\nu x_1 \ldots x_k)\left(\prod_{i=1}^{n} \mathsf{p}_i :: S_i\right)$ is deadlock-free, then by Theorem 3 there is a derivation $\widetilde{\mathcal{D}}$ in PiL with conclusion $[\![P]\!]^{\mathcal{N}}$. Using rule permutations, we can transform $\widetilde{\mathcal{D}}$ into a derivation made (bottom-up) of possibly some $И$-rules followed by $\parr$-rules (i.e., an open derivation of the same shape of $\mathcal{D}_{\langle x_1,\ldots,x_k\rangle}$ or $\mathcal{D}_\varnothing$

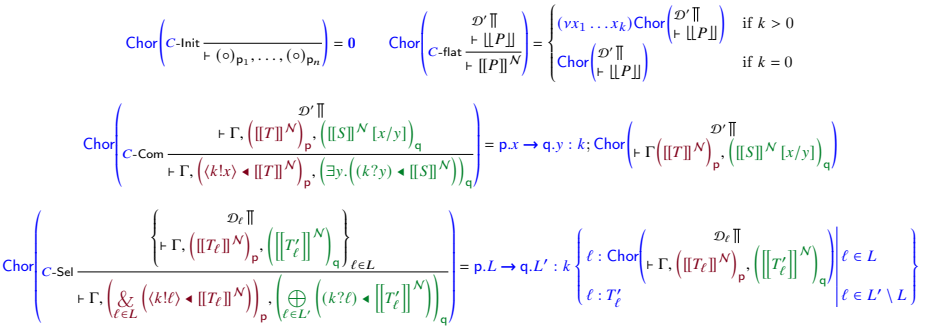


Fig. 15. Interpretation of a derivation in ChorL as a choreography.

from Figure 14), followed by a derivation $\mathcal{D}$ of the sequent $[\![P_1]\!]^{\mathcal{N}}, \ldots, [\![P_n]\!]^{\mathcal{N}}$ which is organized in blocks of rules as open derivations in Figure 14. Note that derivations of the form $\mathcal{D}_{\langle \mathsf{p},x,\mathsf{q},y,k\rangle}$ (resp. $\mathcal{D}_{\langle \mathsf{p},L,\mathsf{q},L',k\rangle}$) correspond to the open derivations in Equation (6). Thus we can replace $\mathcal{D}_{\langle x_1,\ldots,x_k\rangle}$ or $\mathcal{D}_{\varnothing}$ by a C-flat, each $\mathcal{D}_{\langle \mathsf{p},x,\mathsf{q},y,k\rangle}$ (resp. $\mathcal{D}_{\langle \mathsf{p},L,\mathsf{q},L',k\rangle}$) by a C-Com (resp. by a C-Sel), and each $\mathcal{D}_n$ with a C-Init, obtaining the desired derivation in ChorL.

The converse is a consequence of Lemma 3 and Theorem 3.

4.4 Proofs as Choreographies

We can now prove a completeness result of choreographies with respect to the set of deadlock-free flat processes: each deadlock-free flat process is the EPP of a (flat) choreography. To prove this result, we rely on Theorem 6 to establish a direct correspondence between derivations of a sequent encoding a race-free endpoint process in the sequent system ChorL, and execution trees of the same endpoint process. An example the process of choreography extraction from a derivation in ChorL of a deadlock-free endpoint process can be found in the Appendix of [9].

Theorem 7. *Let P be a race-free endpoint process. Then*

$$P \text{ is deadlock-free} \iff \text{there is a choreography } C \text{ such that } \mathsf{EPP}\,(C) = P.$$

Proof. If P is deadlock-free, then by Theorem 6 there is a derivation in ChorL with conclusion $[\![P]\!]^{\mathcal{N}}$. We define the choreography $\mathsf{Chor}(\mathcal{D})$ by case analysis on the bottom-most rule r in $\mathcal{D}$ as shown in Figure 15. We conclude by showing that $\mathsf{EPP}\,(\mathsf{Chor}(\mathcal{D})) = P$ by induction on the structure of $\mathcal{D}$ reasoning on the bottom-most rule r in $\mathcal{D}$. The rigth-to-left implication follows by Theorem 5.

Remark 9. Note that the statement could be made stronger by requiring the choreography C to be flat. In this case, the proof of the right-to-left implication can be proven directly using the inverse of the translation in Figure 15.

	Independent ν and |	Cyclic dependencies	ν-free interaction	Choreography expressivity	Proof System
Caires & Pfenning [20]	✗	✗	✗	N/A	iLL
Wadler [96]	✗	✗	✗	N/A	LL
Dardha & Gay [35]	✓	✓	✗	N/A	LL + mix + ordering
Kokke & Montesi & Peressotti [64]	✓	✗	✓	N/A	LL + hyperenvironments
Carbone & Montesi & Schürmann [26]	✗	✗	✗	✓	LL
This paper	✓	✓	✓	✓	PiL

Fig. 16. Summary of key results in the literature. We describe each column in order: the term constructors for restriction and parallel are separate (independent) in the syntax of processes; cyclic dependencies are allowed; processes can interact (communicate) on a free name (the name does not need to be restricted in the context); choreographies are proven complete for a class of processes (N/A means that this was not considered).

Since every flat process in the π-calculus can be decorated with process names, we can easily extend the completeness result to the π-calculus. We need to pay attention to the difference that $P \twoheadrightarrow \mathsf{EPP}\,(C)$ (instead of $P = \mathsf{EPP}\,(C)$) because of the definition of $\mathsf{EPP}\,(\mathbf{0})$. For example, the choreography that captures $\mathsf{Nil} \mid \mathsf{Nil}$ is $\mathbf{0}$, and $\mathsf{Nil} \mid \mathsf{Nil} \twoheadrightarrow \mathsf{EPP}\,(\mathbf{0}) = \mathsf{Nil}$.

Corollary 2. *Every race-free and deadlock-free flat process P admits a choreography C such that $P \twoheadrightarrow \mathsf{EPP}\,(C)$.*

An important consequence of our results is that the processes that can be captured by choreographies can have cyclic dependencies, as we exemplified with Equations (1) and (8). This significantly extends the proven expressivity of choreographic languages with name mobility, which so far have been shown to capture only the acyclic processes typable with linear logic [26,21].

5 Related Work

We now report on relevant related work. A summarising table of the differences between our work and others based on logic is given in Figure 16.

Proofs as Processes: Linear Logic and Session Types. The proofs-as-processes agenda investigates how linear logic [43] can be used to reason about the behaviour of concurrent processes [1,17]. It has inspired a number of works that aim at preventing safety issues, like processes performing incompatible actions in erroneous attempts to interact (e.g., sending a message with the wrong type). Notable examples include *session types* [55,56] and linear types for the π-calculus [63]. The former can actually be encoded into the latter – a formal reminder of their joint source of inspiration [36].

A more recent line of research formally interprets propositions and proofs in linear logic as, respectively, session types and processes [20,96]. This proofs-as-processes correspondence based on linear logic works for race-free processes, as we consider here. However, it also presents some limitations compared to our framework. Parallel composition and restriction are not offered as independent operators, because of a misalignment with the structures given by the standard rules of linear logic. For example, the cut rule in linear logic handles both

parallel composition and hiding, yielding a 'fused' restriction-parallel operator $(\nu x)(- \mid -)$. Also, the $\otimes$ rule for typing output has two premises, yielding another fused output-parallel operator $x!(y).(- \mid -)$ – note that only bound names can be sent, as in the internal π-calculus [87]. In particular, interaction between processes does not arise simply from parallel composition as in the standard π-calculus, but rather requires both parallel composition and restricting all names on which communication can take place (so communication is always an internal action). This syntactic and semantic gap prevents linear logic from typing safe cyclic dependencies among processes, as in this simplification of Equation (1):

$$(\nu x)(\nu y)\,(x!\langle a\rangle.y?(b).\mathsf{Nil} \mid x?(a).y!\langle b\rangle.\mathsf{Nil}) \tag{12}$$

The same gap prevents having communication on unrestricted channels (as in $x!\langle a\rangle.\mathsf{Nil} \mid x?(a).\mathsf{Nil}$) and having a private channel used by more than two processes. Using $⅋$ and $\otimes$ to type input and output is also in tension with the associativity and commutativity isomorphisms that come with these connectives. These isomorphisms yield unexpected equivalences at the process level, like $x?(a).y?(b).\mathsf{Nil} \equiv y?(b).x?(a).\mathsf{Nil}$.

These shortcomings are not present in our approach, thanks to the use of:

1. The connective $\blacktriangleleft$ for prefixing. The latter then has the expected 'rigid' non-commutative and non-associative semantics.
2. The connective $⅋$ for parallelism. The latter then has the expected equivalences supported by the isomorphisms for $⅋$.
3. Nominal quantifiers, which allow for restricting names without imposing artificial constraints on the structure of processes.

While it is not the first time that these limitations are pointed out, our method is the first logical approach that overcomes them without ad-hoc machinery. Previous works have introduced additional structures to linear logic, like hyperenvironments or indexed families of connectives, in order to address some of these issues [35,92,83,64,24,27]. These additional structures are not necessary to our approach.

Choreographic Programming. Choreographic programming was introduced in [77] as a paradigm for simplifying concurrent and distributed programming. Crucial to the success of this paradigm is building choreographic programming languages that are expressive enough to capture as many safe concurrent behaviours as possible [78]. However, most of the work conducted so far on the study of such expressivity is driven by applications, and a systematic understanding of the classes of processes that can be captured in choreographies is still relatively green.

In [31,61] the authors present methods for *choreography extraction* – inferring from a network of processes an equivalent choreography – for an asynchronous process calculus, respectively without and with process spawning. Another purely algorithmic extraction procedure is provided in [66], for simple choreographies without data – global types, which are roughly the choreographic equivalent of session types. In linear logic, extracting global types from session types can be achieved via derivable rules [24].

The only previous completeness result for the expressivity of choreographic programming is given in [26,21], where it is shown that choreographies can capture the behaviours of all well-typed processes in linear logic. Our work extends the completeness of choreographies to processes that, notably, can (i) have cyclic dependencies (like Equation (12)), (ii) perform communication over free channels, (iii) respect the sequentiality of prefixing. Moreover, and similarly to our previous discussion for proofs-as-processes, extraction in [26,21] requires additional structures (hypersequents and modalities to represent connections) that are not necessary in our method.

Non-Commutative Logic and Nominal Quantifiers. Guglielmi proposed in [45,46] an extension of multiplicative linear logic with a non-commutative operator modelling the interaction of parallel and sequential operators. This led to the design of the *calculus of structures* [47], a formalism for proofs where inference rules can be applied at any depth inside a formula rather than at the top-level connective, and the logic BV including the (associative) non-commutative self-dual connective $\triangleleft$ to model sequentiality. In [19] Bruscoli has established a computation-as-deduction correspondence between specific derivations in BV and executions in a simple fragment of CCS. This correspondence has been extended in the works of Horne, Tiu et al. to include the choice operator (+) of Milner's CCS (modelled via the additive connective $\oplus$), as well as the restriction to model private channels in the π-calculus [58,59]. In these works, restriction has been modelled via *nominal quantifiers* in the spirit of the ones introduced by Pitts and Gabbay for *nominal logic* [85,41], by considering a pair of dual quantifiers[15], instead of a single self-dual quantifier as in [68,86,73].

The logic PiL we use as logical framework to establish our correspondences in this paper takes inspiration from Bruscoli's work and its extension, but it uses a non-associative non-commutative self-dual connective $\blacktriangleleft$ instead of the $\triangleleft$ from BV. This seemingly irrelevant difference (the non-associativity of $\blacktriangleleft$) guarantees the existence of a cut-free sequent calculus to be used as a framework for our correspondence, while for the logic BV and its extension cut-free sequent calculi cannot exist, as proven in [91]. Note that requiring non-associativity for the connective modelling sequentiality is not a syntactical stretch, because the same restriction naturally occurs in process calculi such as CCS and the π-calculus, where sequentiality is defined by an asymmetric prefix operation only allowing to sequentially compose (on the left) atomic instructions, such as send and receive. The other main difference is that in these works derivations represent a single execution, while our derivations represent execution trees. This allows us to state our Theorem 3 without quantifying on the set of derivations of $[\![P]\!]$.

[15] In [57] the authors report the use of a non-self-dual quantifier to model restriction was suggested them by Alessio Guglielmi in a private communication. As explained in detail in [8], the pair of dual nominal quantifiers in [57,59,58] is not the same pair we consider in this paper. This can be observed by looking at the implication $(И x.(A) \otimes И x.(B)) \multimap И x.(A \parr B)$, which is valid in these works, but it is not valid in PiL. For this reason we adopted a different symbol for the dual quantifier of И – i.e., our Я instead of their Э.

6 Conclusion and Future Works

We presented a new approach to the study of processes based on logic, which leverages an interpretation of processes in the π-calculus as formulas in the proof system PiL. By seeing derivations as computation trees, we obtained an elegant method to reason about deadlock-freedom that goes beyond the syntactic and semantic limitations of previous work based on logic. This led us to establishing the first completeness result for the expressivity of choreographic programming with respect to mobile processes with cyclic dependencies.

We discuss next some interesting future directions.

Recursion. Recursion could be modelled by extending PiL with fixpoint operators and rules like the ones in [16,15,3,4,10]. We foresee no major challenges in extending the proof of cut-elimination for μMALL to PiL, since the behaviour of the connective $\blacktriangleleft$ is purely multiplicative (in the sense of [34,44,6]) and the rules for nominal quantifiers do not require the employment of new techniques. In PiL with recursion, properties such as *justness* or *fairness* could be characterised by specific constraints on derivations, corresponding to constraints on threads and paths of the (possibly cyclic) execution trees.

Asynchronous π-calculus. We foresee the possibility of modelling asynchronous communication by including shared buffers, inspired by previous work on concurrent constraint programming [88,82] and its strong ties to logic programming [76,89,60,39,80,81,84]. However, buffers with capacity greater than 2 have non-sequential-parallel structures and therefore cannot be described efficiently using binary connectives [94,30]. We may thus need to consider *graphical connectives* [2,5].

Proof Nets. In [8] we define proof nets for PiL, capturing local rule permutations, and providing canonical representative for execution trees up-to interleaving concurrency. This syntax could be used to refine the correspondence between proofs and choreographies (Theorem 7). We plan to study the extension of the computation-as-deduction paradigm in the case of proof net expansion, following the ideas in [12,13,7], as well as to use a notion of orthogonality for modules of proof nets (in the sense of [34,13,7]) to study testing preorders [37,38,49,18].

Completeness of Choreographies. The literature of choreographic programming languages includes features of practical interest that extend the expressivity of choreographies – like process spawning [32] and nondeterminism [78]. Exploring extensions of PiL to capture these features is interesting future work. For process spawning, a simple solution could be achieved by defining a way to dynamically assign process names to properly define the map $\mathsf{Chor}(\cdot)$.

Acknowledgements. Partially supported by Villum Fonden (grants no. 29518 and 50079). Co-funded by the European Union's Horizon 2020 research and innovation program under the Marie Sklodowska-Curie grant agreement No 945332. Co-funded by the European Union (ERC, CHORDS, 101124225). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

References

1. Abramsky, S.: Proofs as processes. In: Selected Papers of the Conference on Meeting on the Mathematical Foundations of Programming Semantics, Part I: Linear Logic: Linear Logic. pp. 5–9. MFPS '92, Elsevier Science Publishers B. V., NLD (1992)
2. Acclavio, M.: Sequent systems on undirected graphs. In: Benzmüller, C., Heule, M.J., Schmidt, R.A. (eds.) Automated Reasoning. pp. 216–236. Springer Nature Switzerland, Cham (2024)
3. Acclavio, M., Curzi, G., Guerrieri, G.: Infinitary cut-elimination via finite approximations. CoRR **abs/2308.07789** (2023). https://doi.org/10.48550/ARXIV.2308.07789, https://doi.org/10.48550/arXiv.2308.07789
4. Acclavio, M., Curzi, G., Guerrieri, G.: Infinitary cut-elimination via finite approximations (extended version) (2024), https://arxiv.org/abs/2308.07789
5. Acclavio, M., Horne, R., Mauw, S., Straßburger, L.: A Graphical Proof Theory of Logical Time. In: Felty, A.P. (ed.) 7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022). Leibniz International Proceedings in Informatics (LIPIcs), vol. 228, pp. 22:1–22:25. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.FSCD.2022.22, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2022.22
6. Acclavio, M., Maieli, R.: Generalized Connectives for Multiplicative Linear Logic. In: Fernández, M., Muscholl, A. (eds.) 28th EACSL Annual Conference on Computer Science Logic (CSL 2020). Leibniz International Proceedings in Informatics (LIPIcs), vol. 152, pp. 6:1–6:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020). https://doi.org/10.4230/LIPIcs.CSL.2020.6, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CSL.2020.6
7. Acclavio, M., Maieli, R.: Logic programming with multiplicative structures. CoRR **abs/2403.03032** (2024). https://doi.org/10.48550/ARXIV.2403.03032, https://doi.org/10.48550/arXiv.2403.03032
8. Acclavio, M., Manara, G.: Proofs as execution trees for the π-calculus (2024), https://arxiv.org/abs/2411.08847
9. Acclavio, M., Manara, G., Montesi, F.: Formulas as processes, deadlock-freedom as choreographies (extended version) (2025), https://arxiv.org/abs/2501.08928
10. Acclavio, M., Montesi, F., Peressotti, M.: On propositional dynamic logic and concurrency (2024)
11. Andreoli, J.M.: Focussing and proof construction. Annals of Pure and Applied Logic **107**(1), 131–163 (2001). https://doi.org/https://doi.org/10.1016/S0168-0072(00)00032-4, https://www.sciencedirect.com/science/article/pii/S0168007200000324
12. Andreoli, J.M.: Focussing proof-net construction as a middleware paradigm. In: Voronkov, A. (ed.) Automated Deduction—CADE-18. pp. 501–516. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
13. Andreoli, J.M., Mazaré, L.: Concurrent construction of proof-nets. In: Baaz, M., Makowsky, J.A. (eds.) Computer Science Logic. pp. 29–42. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
14. Aschieri, F., Genco, F.A.: Par means parallel: multiplicative linear logic proofs as concurrent functional programs. Proc. ACM Program. Lang. **4**(POPL) (dec 2019). https://doi.org/10.1145/3371086, https://doi.org/10.1145/3371086

15. Baelde, D., Doumane, A., Saurin, A.: Infinitary proof theory: the multiplicative additive case. In: Talbot, J., Regnier, L. (eds.) 25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 - September 1, 2016, Marseille, France. LIPIcs, vol. 62, pp. 42:1–42:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPIcs.CSL.2016.42, https://doi.org/10.4230/LIPIcs.CSL.2016.42
16. Baelde, D., Miller, D.: Least and greatest fixed points in linear logic. In: Dershowitz, N., Voronkov, A. (eds.) Logic for Programming, Artificial Intelligence, and Reasoning, 14th International Conference, LPAR 2007, Yerevan, Armenia, October 15-19, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4790, pp. 92–106. Springer (2007). https://doi.org/10.1007/978-3-540-75560-9_9, https://doi.org/10.1007/978-3-540-75560-9_9
17. Bellin, G., Scott, P.: On the π-calculus and linear logic. Theoretical Computer Science **135**(1), 11–65 (1994). https://doi.org/https://doi.org/10.1016/0304-3975(94)00104-9, https://www.sciencedirect.com/science/article/pii/0304397594001049
18. Bernardi, G., Hennessy, M.: Mutually Testing Processes. Logical Methods in Computer Science **Volume 11, Issue 2** (Apr 2015). https://doi.org/10.2168/LMCS-11(2:1)2015, https://lmcs.episciences.org/776
19. Bruscoli, P.: A purely logical account of sequentiality in proof search. In: International Conference on Logic Programming. pp. 302–316. Springer (2002)
20. Caires, L., Pfenning, F.: Session types as intuitionistic linear propositions. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010 - Concurrency Theory. pp. 222–236. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
21. Carbone, M., Cruz-Filipe, L., Montesi, F., Murawska, A.: Multiparty classical choreographies. In: Mesnard, F., Stuckey, P.J. (eds.) Logic-Based Program Synthesis and Transformation - 28th International Symposium, LOPSTR 2018, Frankfurt/Main, Germany, September 4-6, 2018, Revised Selected Papers. Lecture Notes in Computer Science, vol. 11408, pp. 59–76. Springer (2018). https://doi.org/10.1007/978-3-030-13838-7_4, https://doi.org/10.1007/978-3-030-13838-7_4
22. Carbone, M., Dardha, O., Montesi, F.: Progress as compositional lock-freedom. In: Kühn, E., Pugliese, R. (eds.) Coordination Models and Languages. pp. 49–64. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
23. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. **34**(2), 8:1–8:78 (2012). https://doi.org/10.1145/2220365.2220367, https://doi.org/10.1145/2220365.2220367
24. Carbone, M., Lindley, S., Montesi, F., Schürmann, C., Wadler, P.: Coherence generalises duality: A logical explanation of multiparty session types. In: Desharnais, J., Jagadeesan, R. (eds.) 27th International Conference on Concurrency Theory, CONCUR 2016, August 23-26, 2016, Québec City, Canada. LIPIcs, vol. 59, pp. 33:1–33:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2016). https://doi.org/10.4230/LIPICS.CONCUR.2016.33, https://doi.org/10.4230/LIPIcs.CONCUR.2016.33
25. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: Giacobazzi, R., Cousot, R. (eds.) The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013. pp. 263–274. ACM (2013). https://doi.org/10.1145/2429069.2429101, https://doi.org/10.1145/2429069.2429101
26. Carbone, M., Montesi, F., Schürmann, C.: Choreographies, logically. Distributed Comput. **31**(1), 51–67 (2018). https://doi.org/10.1007/S00446-017-0295-1, https://doi.org/10.1007/s00446-017-0295-1

27. Carbone, M., Montesi, F., Schürmann, C., Yoshida, N.: Multiparty session types as coherence proofs. Acta Informatica **54**(3), 243–269 (2017). https://doi.org/10.1007/S00236-016-0285-Y, https://doi.org/10.1007/s00236-016-0285-y
28. Cockett, J., Seely, R.: Weakly distributive categories. Journal of Pure and Applied Algebra **114**(2), 133–173 (1997). https://doi.org/https://doi.org/10.1016/0022-4049(95)00160-3, https://www.sciencedirect.com/science/article/pii/0022404995001603
29. Coppo, M., Dezani-Ciancaglini, M., Yoshida, N., Padovani, L.: Global progress for dynamically interleaved multiparty sessions. Math. Struct. Comput. Sci. **26**(2), 238–302 (2016). https://doi.org/10.1017/S0960129514000188, https://doi.org/10.1017/S0960129514000188
30. Corneil, D., Lerchs, H., Burlingham, L.: Complement reducible graphs. Discrete Applied Mathematics **3**(3), 163–174 (1981). https://doi.org/https://doi.org/10.1016/0166-218X(81)90013-5, https://www.sciencedirect.com/science/article/pii/0166218X81900135
31. Cruz-Filipe, L., Larsen, K.S., Montesi, F.: The paths to choreography extraction. In: Esparza, J., Murawski, A.S. (eds.) Foundations of Software Science and Computation Structures - 20th International Conference, FOSSACS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10203, pp. 424–440 (2017). https://doi.org/10.1007/978-3-662-54458-7_25, https://doi.org/10.1007/978-3-662-54458-7_25
32. Cruz-Filipe, L., Montesi, F.: Procedural choreographic programming. In: Bouajjani, A., Silva, A. (eds.) Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10321, pp. 92–107. Springer (2017). https://doi.org/10.1007/978-3-319-60225-7_7, https://doi.org/10.1007/978-3-319-60225-7_7
33. Cruz-Filipe, L., Montesi, F.: A core model for choreographic programming. Theor. Comput. Sci. **802**, 38–66 (2020). https://doi.org/10.1016/J.TCS.2019.07.005, https://doi.org/10.1016/j.tcs.2019.07.005
34. Danos, V., Regnier, L.: The structure of multiplicatives. Archive for Mathematical Logic **28**(3), 181–203 (1989). https://doi.org/10.1007/BF01622878, https://doi.org/10.1007/BF01622878
35. Dardha, O., Gay, S.J.: A new linear logic for deadlock-free session-typed processes. In: Baier, C., Dal Lago, U. (eds.) Foundations of Software Science and Computation Structures. pp. 91–109. Springer International Publishing, Cham (2018)
36. Dardha, O., Giachino, E., Sangiorgi, D.: Session types revisited. Inf. Comput. **256**, 253–286 (2017). https://doi.org/10.1016/J.IC.2017.06.002, https://doi.org/10.1016/j.ic.2017.06.002
37. De Nicola, R., Hennessy, M.: Testing equivalences for processes. Theoretical Computer Science **34**(1), 83–133 (1984). https://doi.org/https://doi.org/10.1016/0304-3975(84)90113-0, https://www.sciencedirect.com/science/article/pii/0304397584901130
38. De Nicola, R., Hennessy, M.: Ccs without τ's. In: Ehrig, H., Kowalski, R., Levi, G., Montanari, U. (eds.) TAPSOFT '87. pp. 138–152. Springer Berlin Heidelberg, Berlin, Heidelberg (1987)

39. Fages, F., Ruet, P., Soliman, S.: Linear concurrent constraint programming: Operational and phase semantics. Information and Computation **165**(1), 14–41 (2001). https://doi.org/https://doi.org/10.1006/inco.2000.3002, https://www.sciencedirect.com/science/article/pii/S0890540100930025
40. Fleury, A., Retoré, C.: The mix rule. Mathematical Structures in Computer Science **4**(2), 273–285 (1994). https://doi.org/10.1017/S0960129500000451
41. Gabbay, M.J., Pitts, A.M.: A new approach to abstract syntax with variable binding. Form. Asp. Comput. **13**(3–5), 341–363 (jul 2002). https://doi.org/10.1007/s001650200016, https://doi.org/10.1007/s001650200016
42. Gay, S., Hole, M.: Subtyping for session types in the pi calculus. Acta Informatica **42**, 191–225 (2005)
43. Girard, J.Y.: Linear logic. Theoretical Computer Science **50**(1), 1–101 (1987). https://doi.org/10.1016/0304-3975(87)90045-4
44. Girard, J.Y.: On the meaning of logical rules II: multiplicatives and additives. NATO ASI Series F Computer and Systems Sciences **175**, 183–212 (2000)
45. Guglielmi, A.: Concurrency and plan generation in a logic programming language with a sequential operator. In: ICLP. pp. 240–254. Citeseer (1994)
46. Guglielmi, A.: Sequentiality by linear implication and universal quantification. In: Desel, J. (ed.) Structures in Concurrency Theory. pp. 160–174. Springer London, London (1995)
47. Guglielmi, A.: A system of interaction and structure. ACM Trans. Comput. Logic **8**(1), 1–es (Jan 2007). https://doi.org/10.1145/1182613.1182614, https://doi.org/10.1145/1182613.1182614
48. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic, pp. 99–217. Springer Netherlands, Dordrecht (2002). https://doi.org/10.1007/978-94-017-0456-4_2, https://doi.org/10.1007/978-94-017-0456-4_2
49. Hennessy, M.: Algebraic theory of processes. MIT Press, Cambridge, MA, USA (1988)
50. Hennessy, M.: A distributed Pi-calculus. Cambridge University Press (2007)
51. Hennessy, M., Milner, R.: On observing nondeterminism and concurrency. In: de Bakker, J., van Leeuwen, J. (eds.) Automata, Languages and Programming. pp. 299–309. Springer Berlin Heidelberg, Berlin, Heidelberg (1980)
52. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (oct 1969). https://doi.org/10.1145/363235.363259, https://doi.org/10.1145/363235.363259
53. Hodas, J., Miller, D.: Logic programming in a fragment of intuitionistic linear logic. Information and Computation **110**(2), 327–365 (1994). https://doi.org/https://doi.org/10.1006/inco.1994.1036, https://www.sciencedirect.com/science/article/pii/S0890540184710364
54. Holmström, S.: Hennessy-milner logic with recursion as a specification language, and a refinement calculus based on it. In: Rattray, C. (ed.) Specification and Verification of Concurrent Systems. pp. 294–330. Springer London, London (1990)
55. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) CONCUR'93. pp. 509–523. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)
56. Honda, K., Vasconcelos, V.T., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Hankin, C. (ed.) Programming Languages and Systems. pp. 122–138. Springer Berlin Heidelberg, Berlin, Heidelberg (1998)

57. Horne, R., Tiu, A.: Constructing weak simulations from linear implications for processes with private names. Mathematical Structures in Computer Science **29**(8), 1275–1308 (2019). https://doi.org/10.1017/S0960129518000452
58. Horne, R., Tiu, A., Aman, B., Ciobanu, G.: Private Names in Non-Commutative Logic. In: Desharnais, J., Jagadeesan, R. (eds.) 27th International Conference on Concurrency Theory (CONCUR 2016). Leibniz International Proceedings in Informatics (LIPIcs), vol. 59, pp. 31:1–31:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2016). https://doi.org/10.4230/LIPIcs.CONCUR.2016.31, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.CONCUR.2016.31
59. Horne, R., Tiu, A., Aman, B., Ciobanu, G.: De morgan dual nominal quantifiers modelling private names in non-commutative logic. ACM Trans. Comput. Logic **20**(4) (jul 2019). https://doi.org/10.1145/3325821, https://doi.org/10.1145/3325821
60. Jaffar, J., Maher, M.J.: Constraint logic programming: a survey. The Journal of Logic Programming **19-20**, 503–581 (1994). https://doi.org/https://doi.org/10.1016/0743-1066(94)90033-7, https://www.sciencedirect.com/science/article/pii/0743106694900337, special Issue: Ten Years of Logic Programming
61. Kjær, B.A., Cruz-Filipe, L., Montesi, F.: From infinity to choreographies: Extraction for unbounded systems (2022)
62. Kobayashi, N.: A type system for lock-free processes. Information and Computation **177**(2), 122–159 (2002). https://doi.org/https://doi.org/10.1006/inco.2002.3171, https://www.sciencedirect.com/science/article/pii/S0890540102931718
63. Kobayashi, N., Pierce, B.C., Turner, D.N.: Linearity and the pi-calculus. ACM Trans. Program. Lang. Syst. **21**(5), 914–947 (1999). https://doi.org/10.1145/330249.330251, https://doi.org/10.1145/330249.330251
64. Kokke, W., Montesi, F., Peressotti, M.: Better late than never: a fully-abstract semantics for classical processes. Proc. ACM Program. Lang. **3**(POPL) (jan 2019). https://doi.org/10.1145/3290337, https://doi.org/10.1145/3290337
65. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science **27**(3), 333–354 (1983). https://doi.org/https://doi.org/10.1016/0304-3975(82)90125-6, https://www.sciencedirect.com/science/article/pii/0304397582901256, special Issue Ninth International Colloquium on Automata, Languages and Programming (ICALP) Aarhus, Summer 1982
66. Lange, J., Tuosto, E., Yoshida, N.: From communicating machines to graphical choreographies. SIGPLAN Not. **50**(1), 221–232 (jan 2015). https://doi.org/10.1145/2775051.2676964, https://doi.org/10.1145/2775051.2676964
67. Martin-Löf, P.: Constructive mathematics and computer programming. In: Studies in Logic and the Foundations of Mathematics, vol. 104, pp. 153–175. Elsevier (1982)
68. Menni, M.: About И-quantifiers. Applied categorical structures **11**, 421–445 (2003)
69. Miller, D.: Hereditary harrop formulas and logic programming. In: Proceedings of the VIII International Congress of Logic, Methodology, and Philosophy of Science. pp. 153–156 (1987)
70. Miller, D.: Abstract syntax and logic programming. In: Voronkov, A. (ed.) Logic Programming. pp. 322–337. Springer Berlin Heidelberg, Berlin, Heidelberg (1992)
71. Miller, D.: The π-calculus as a theory in linear logic: Preliminary results. In: Lamma, E., Mello, P. (eds.) Extensions of Logic Programming. pp. 242–264. Springer Berlin Heidelberg, Berlin, Heidelberg (1993)

72. Miller, D., Nadathur, G., Pfenning, F., Scedrov, A.: Uniform proofs as a foundation for logic programming. Annals of Pure and Applied Logic **51**(1), 125–157 (1991). https://doi.org/https://doi.org/10.1016/0168-0072(91)90068-W, https://www.sciencedirect.com/science/article/pii/016800729190068W
73. Miller, D., Tiu, A.: A proof theory for generic judgments. ACM Trans. Comput. Logic **6**(4), 749–783 (oct 2005). https://doi.org/10.1145/1094622.1094628, https://doi.org/10.1145/1094622.1094628
74. Milner, R.: A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92. Springer (1980). https://doi.org/10.1007/3-540-10235-3, https://doi.org/10.1007/3-540-10235-3
75. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, i. Information and Computation **100**(1), 1–40 (1992). https://doi.org/https://doi.org/10.1016/0890-5401(92)90008-4, https://www.sciencedirect.com/science/article/pii/0890540192900084
76. Montanari, U.: Networks of constraints: Fundamental properties and applications to picture processing. Information Sciences **7**, 95–132 (1974). https://doi.org/https://doi.org/10.1016/0020-0255(74)90008-5, https://www.sciencedirect.com/science/article/pii/0020025574900085
77. Montesi, F.: Choreographic Programming. Ph.D. thesis, IT University of Copenhagen (2013), https://www.fabriziomontesi.com/files/choreographic-programming.pdf
78. Montesi, F.: Introduction to Choreographies. Cambridge University Press (2023). https://doi.org/10.1017/9781108981491
79. Montesi, F., Yoshida, N.: Compositional choreographies. In: D'Argenio, P.R., Melgratti, H.C. (eds.) CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings. Lecture Notes in Computer Science, vol. 8052, pp. 425–439. Springer (2013). https://doi.org/10.1007/978-3-642-40184-8_30, https://doi.org/10.1007/978-3-642-40184-8_30
80. Olarte, C., Pimentel, E.: On concurrent behaviors and focusing in linear logic. Theoretical Computer Science **685**, 46–64 (2017). https://doi.org/https://doi.org/10.1016/j.tcs.2016.08.026, https://www.sciencedirect.com/science/article/pii/S0304397516304832, logical and Semantic Frameworks with Applications
81. Olarte, C., Pimentel, E., Nigam, V.: Subexponential concurrent constraint programming. Theoretical Computer Science **606**, 98–120 (2015). https://doi.org/https://doi.org/10.1016/j.tcs.2015.06.031, https://www.sciencedirect.com/science/article/pii/S0304397515005411, logical and Semantic Frameworks with Applications
82. Olarte, C., Rueda, C., Valencia, F.D.: Models and emerging trends of concurrent constraint programming. Constraints **18**, 535–578 (2013)
83. Padovani, L.: Deadlock and lock freedom in the linear π-calculus. In: Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS). CSL-LICS '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2603088.2603116, https://doi.org/10.1145/2603088.2603116
84. Pimentel, E., Olarte, C., Nigam, V.: Process-As-Formula Interpretation: A Substructural Multimodal View. In: Kobayashi, N. (ed.) 6th International Conference on Formal Structures for Computation and Deduction (FSCD 2021).

Leibniz International Proceedings in Informatics (LIPIcs), vol. 195, pp. 3:1–3:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.FSCD.2021.3, https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.FSCD.2021.3

85. Pitts, A.M.: Nominal logic, a first order theory of names and binding. Information and Computation **186**(2), 165–193 (2003). https://doi.org/https://doi.org/10.1016/S0890-5401(03)00138-X, https://www.sciencedirect.com/science/article/pii/S089054010300138X, theoretical Aspects of Computer Software (TACS 2001)
86. Roversi, L.: A deep inference system with a self-dual binder which is complete for linear lambda calculus. Journal of Logic and Computation **26**(2), 677–698 (2016). https://doi.org/10.1093/logcom/exu033
87. Sangiorgi, D.: Pi-i: A symmetric calculus based on internal mobility. In: Mosses, P.D., Nielsen, M., Schwartzbach, M.I. (eds.) TAPSOFT'95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995, Proceedings. Lecture Notes in Computer Science, vol. 915, pp. 172–186. Springer (1995). https://doi.org/10.1007/3-540-59293-8_194, https://doi.org/10.1007/3-540-59293-8_194
88. Saraswat, V.A., Rinard, M.: Concurrent constraint programming. In: Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. p. 232–245. POPL '90, Association for Computing Machinery, New York, NY, USA (1989). https://doi.org/10.1145/96709.96733, https://doi.org/10.1145/96709.96733
89. Shapiro, E.: The family of concurrent logic programming languages. ACM Comput. Surv. **21**(3), 413–510 (sep 1989). https://doi.org/10.1145/72551.72555, https://doi.org/10.1145/72551.72555
90. Sørensen, M.H., Urzyczyn, P.: Lectures on the Curry-Howard isomorphism. Elsevier (2006)
91. Tiu, A.: A System of Interaction and Structure II: The Need for Deep Inference. Logical Methods in Computer Science **Volume 2, Issue 2** (Apr 2006). https://doi.org/10.2168/LMCS-2(2:4)2006, https://lmcs.episciences.org/2252
92. Torres Vieira, H., Thudichum Vasconcelos, V.: Typing progress in communication-centred systems. In: De Nicola, R., Julien, C. (eds.) Coordination Models and Languages. pp. 236–250. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
93. Troelstra, A.S., Schwichtenberg, H.: Basic Proof Theory. Cambridge Tracts in Theoretical Computer Science, Cambridge University Press, 2 edn. (2000). https://doi.org/10.1017/CBO9781139168717
94. Valdes, J., Tarjan, R.E., Lawler, E.L.: The recognition of series parallel digraphs. In: Proceedings of the eleventh annual ACM symposium on Theory of computing. pp. 1–12. ACM (1979)
95. Vasconcelos, V.T.: Fundamentals of session types. Information and Computation **217**, 52–70 (2012). https://doi.org/https://doi.org/10.1016/j.ic.2012.05.002, https://www.sciencedirect.com/science/article/pii/S0890540112001022
96. Wadler, P.: Propositions as sessions. In: Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming. p. 273–286. ICFP '12, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2364527.2364568, https://doi.org/10.1145/2364527.2364568