Choreographies, Logically

Marco Carbone[1], Fabrizio Montesi[2⋆], and Carsten Schürmann[1]

[1]IT University of Copenhagen [2]University of Southern Denmark

Abstract. In Choreographic Programming, a distributed system is programmed by giving a *choreography*, a global description of its interactions, instead of separately specifying the behaviour of each of its processes. Process implementations in terms of a distributed language can then be automatically projected from a choreography.
We present Linear Compositional Choreographies (LCC), a proof theory for reasoning about programs that modularly combine choreographies with processes. Using LCC, we logically reconstruct a semantics and a projection procedure for programs. For the first time, we also obtain a procedure for extracting choreographies from process terms.

1 Introduction

Choreographic Programming is a programming paradigm for distributed systems inspired by the "Alice and Bob" notation, where programs, called *choreographies*, are global descriptions of how endpoint processes interact during execution [14, 21, 1]. The typical set of programs defining the actions performed by each process is then generated by *endpoint projection* (EPP) [17, 12, 8, 5, 9, 15].

The key aspect of choreography languages is that process interactions are treated linearly, i.e., they are executed exactly once. Previous work [8, 9, 15] developed correct notions of EPP by using session types [11], linear types for communications inspired by linear logic [10]. Despite the deep connections between choreographies and linearity, the following question remains unanswered:

Is there a formal connection between choreographies and linear logic?

Finding such a connection would contribute to a more precise understanding of choreographies, and possibly lead to answering open questions about them.

A good starting point for answering our question is a recent line of work on a Curry-Howard correspondence between the internal π-calculus [18] and linear logic [7, 22]. In particular, proofs in Intuitionistic Linear Logic (ILL) correspond to π-calculus terms (proofs-as-programs) and ILL propositions correspond to session types [7]. An ILL judgement describes the interface of a process, for example:

$$P \triangleright x : A, y : B \vdash z : C$$

Above, process P needs to be composed with other processes that provide the behaviours (represented as types) A on channel x and B on channel y, in order to provide behaviour C on channel z. The focus is on how the process can be composed with other *external* processes, abstracting from the *internal* communications enacted inside the process itself (which may contain communicating

⋆ Work performed while the author was employed at the IT University of Copenhagen.

sub-processes). On the contrary, choreographies are descriptions of the internal interactions among the processes inside a system, and therefore type systems for choreographies focus on checking such internal interactions [8, 9]. It is thus unclear how the linear typing of ILL can be related to choreographies.

In this paper, we present Linear Compositional Choreographies (LCC), a proof theory inspired by linear logic for typing programs that modularly combine choreographies with processes in the internal π-calculus. The key aspect of LCC is to extend ILL judgements to describe interactions among internal processes in a system. Thanks to LCC, not only do we obtain a logical understanding of choreographic programming, but we also provide the foundations for tackling the open problem of extracting a choreography from a system of processes.

Main Contributions. We summarise our main contributions:

Linear Compositional Choreographies (LCC). We present LCC, a generalisation of ILL where judgements can also describe the internal interactions of a system (§ 3). LCC proofs are equipped with unique proof terms, called LCC programs, following the Curry-Howard interpretation of proofs-as-programs. LCC programs are in a language where choreographies and processes are modularly combined by following protocols given in the type language of LCC (à la session types [11]).

Logically-derived semantics. We derive a semantics for LCC programs from our proof theory (§ 4): (i) some rule applications in LCC proofs can be permuted (commuting conversions), defining equivalences (structural congruence) on LCC programs (§ 4.1); (ii) some proofs can be safely reduced to smaller proofs, corresponding to executing communications (§ 4.2). By following our semantics, we prove that all internal communications in a system can be reduced (proof normalisation), i.e., LCC programs are deadlock-free by construction (§ 4.3).

Choreography Extraction and Endpoint Projection. LCC consists of two fragments: the *action fragment*, which manipulates the external interfaces of processes, and the *interaction fragment*, which handles internal communications. We derive automatic transformations from proofs in either fragment to proofs in the other, yielding procedures of endpoint projection and choreography extraction (§ 5) that preserve the semantics of LCC programs. This is the first work addressing extraction for a fragment of the π-calculus, providing the foundations for a new development methodology where programmers can compose choreographies with existing process code (e.g., software libraries) and then obtain a choreography that describes the overall behaviour of the entire composition.

2 From ILL to LCC

In this section, we informally introduce processes and choreographies, and revisit the Curry-Howard correspondence between the internal π-calculus and ILL [7]. Building on ILL, we introduce the intuition behind the proof theory of LCC.

Processes and Choreographies. Consider the following processes:

$$\underbrace{\overline{x}(tea);\ x(tr);\ \overline{tr}(p)}_{P_{\mathsf{client}}} \qquad \underbrace{x(tea);\ \overline{x}(tr);\ tr(p);\ \overline{b}(m)}_{P_{\mathsf{server}}} \qquad \underbrace{b(m)}_{P_{\mathsf{bank}}} \tag{1}$$

The three processes above, given as internal π-calculus terms [18], denote a system composed by three endpoints (client, server, and bank). Their parallel execution is such that: client sends to server a request for tea on a channel x; then, server replies to client on the same channel x with a new channel tr (for transaction); client uses tr for sending to server the payment p; after receiving the payment, server deposits some money m by sending it over channel b to bank.

Programming with processes is error-prone, since they do not give an explicit description of how endpoints interact [14]. By contrast, choreographies specify how messages flow during execution [21]. For example, the choreography

$$\begin{array}{l} 1.\ \mathsf{client} \to \mathsf{server} : x(tea);\ \mathsf{server} \to \mathsf{client} : x(tr); \\ 2.\ \mathsf{client} \to \mathsf{server} : tr(p);\ \mathsf{server} \to \mathsf{bank} : b(m) \end{array} \tag{2}$$

defines the communications that occur in (1). We read $\mathsf{client} \to \mathsf{server} : x(tea)$ as "process client sends tea to process server through channel x".

ILL and the π-calculus. The processes in (1) can be typed by ILL, using propositions as session types that describe the usage of channels. For example, channel x in P_{client} has type $\mathbf{string} \otimes (\mathbf{string} \multimap \mathsf{end}) \multimap \mathsf{end}$, meaning: send a string; then, receive a channel of type $\mathbf{string} \multimap \mathsf{end}$ and, finally, stop (end). Concretely, in P_{client}, the channel of type $\mathbf{string} \multimap \mathsf{end}$ received through x is channel tr. The type of tr says that the process sending tr, i.e., P_{server}, will use it to receive a **string**; therefore, process P_{client} must implement the dual operation of that implemented by P_{server}, i.e., the output $\overline{tr}(p)$. Similarly, channel b has type $\mathbf{int} \otimes \mathsf{end}$ in P_{server}. We can formalise this intuition with the following three ILL judgements, where $A = \mathbf{string} \otimes (\mathbf{string} \multimap \mathsf{end}) \multimap \mathsf{end}$ and $B = \mathbf{int} \otimes \mathsf{end}$:

$$P_{\mathsf{client}} \triangleright \cdot \vdash x : A \qquad P_{\mathsf{server}} \triangleright x : A \vdash b : B \qquad P_{\mathsf{bank}} \triangleright b : B \vdash z : \mathsf{end}$$

Recall that $P_{\mathsf{server}} \triangleright x : A \vdash b : B$ reads as "given a context that implements channel x with type A, process P_{server} implements channel b with type B". Given these judgements, we compose P_{client}, P_{server}, and P_{bank} using channels x and b as:

$$(\boldsymbol{\nu} x)\,(\ P_{\mathsf{client}} \quad |_x \quad (\boldsymbol{\nu} b)\,(\ P_{\mathsf{server}} \quad |_b \quad P_{\mathsf{bank}}\ )\ ) \tag{3}$$

The compositions in (3) can be typed using the Cut rule of ILL:

$$\frac{P \triangleright \Delta_1 \vdash x : A \qquad Q \triangleright \Delta_2, x : A \vdash y : B}{(\boldsymbol{\nu} x)\,(P \mid Q) \triangleright \Delta_1, \Delta_2 \vdash y : B}\ \mathsf{Cut} \tag{4}$$

Above, Δ_1 and Δ_2 are sets of typing assignments, e.g., $z : D$. We interpret rule Cut as "If a process provides A on channel x, and another requires A on channel x to provide B on channel y, their parallel execution provides B on channel y".

Proofs in ILL correspond to process terms in the internal π-calculus [7], and applications of rule Cut can always be eliminated, a proof normalisation procedure known as cut elimination. This procedure provides a model of computation for processes. We illustrate a *cut reduction*, a step of cut elimination, in the following (we omit process terms for readability):

$$\frac{\dfrac{C_1 \vdash A \qquad C_2 \vdash B}{C_1, C_2 \vdash A \otimes B}\otimes\mathsf{R} \qquad \dfrac{A, B \vdash D}{A \otimes B \vdash D}\otimes\mathsf{L}}{C_1, C_2 \vdash D}\ \mathsf{Cut} \quad \Longrightarrow \quad \frac{C_1 \vdash A \qquad \dfrac{C_2 \vdash B \qquad A, B \vdash D}{C_2, A \vdash D}\ \mathsf{Cut}}{C_1, C_2 \vdash D}\ \mathsf{Cut}$$

The proof on the left-hand side applies a cut to two proofs, one providing $A \otimes B$, and the other providing D when provided with $A \otimes B$. The cut-reduction above ($\Rightarrow$) shows how this proof can be simplified to a proof where the cut on $A \otimes B$ is reduced to two cuts on the smaller formulas A and B. A cut-reduction corresponds to executing a communication between two processes, one outputting on a channel of type $A \otimes B$, and another inputting from the same channel [7]. Executing the communication yields a new system corresponding to the proof on the right-hand side. Cut-free proofs correspond to systems that have successfully completed all their internal communications.

Towards LCC. Cut reductions in ILL model the interactions between the internal processes in a system, which is exactly what choreographies describe syntactically. Therefore, in order to capture choreographies, we wish our proof theory to reason about transformations such as the cut reduction above.

ILL judgements give us no information on the applications of rule Cut in a proof. In contrast, standard type systems for choreographies [8, 9, 15] have different judgements: instead of interfaces for later composition, they contain information about internal processes and their interactions. Following this observation, we make two important additions to ILL judgements. First, we extend them to describe multiple processes by using *hypersequents*, i.e., collections of multiple ILL sequents [2]. Second, we represent the *connections* between sequents in a hypersequent, since two processes need to share a common connection for interacting. The following is an LCC judgement:

$$P \triangleright \Delta_1 \vdash x : \bullet A \mid \Delta_2, x : \bullet A \vdash y : B$$

Above, we composed two ILL sequents with the operator $\mid$, which captures the parallel composition of processes. The two sequents are *connected* through channel x, denoted by the marking $\bullet$. We will use hypersequents and marking let us reason about interactions by handling both ends of a connection.

LCC judgements can express cut elimination as a proof. For example,

$$Q \triangleright z_1 : C_1, z_2 : C_2 \vdash x : \bullet A \otimes B \mid x : \bullet A \otimes B \vdash w : D$$

represents the left-hand side of the cut reduction seen previously, where a process requires C_1 and C_2 to perform an interaction of type $A \otimes B$ with another process that can then provide D. Importantly, the connection of type $A \otimes B$ between the two sequents cannot be composed with external systems since it is used for internal interactions. Using our judgements, we can capture cut reductions:

$$Q' \triangleright z_1 : C_1 \vdash y : \bullet A \mid z_2 : C_2 \vdash x : \bullet B \mid y : \bullet A, x : \bullet B \vdash w : D$$

The new judgement describes a system that still requires C_1 and C_2 in order to provide D, but now with three processes: one providing A from C_1, one providing B from C_2 and, finally, one using A and B for providing D. Also, the first two sequents are connected to the third one. This corresponds to the right-hand side of the cut reduction seen previously, where process Q reduces to process Q'.

We can now express the different internal states of a system before and after a cut reduction, by the structure of its connections in our judgements. This intuition is behind the new rules for typing choreographies presented in § 3.

3 Linear Compositional Choreographies

We present Linear Compositional Choreographies (LCC), a proof theory for typing programs that can modularly combine choreographies and processes.

Types. LCC propositions, or types, are defined as:

$$\textit{(Propositions)} \quad A, B ::= \mathbf{1} \quad | \quad A \otimes B \quad | \quad A \multimap B \quad | \quad A \oplus B \quad | \quad A \& B$$

LCC propositions are the same as in ILL: $\otimes$ and $\multimap$ are the multiplicative connectives, while $\oplus$ and $\&$ are additives. $\mathbf{1}$ is the atomic proposition. $A \otimes B$ is interpreted as "output a channel of type A and then behave as specified by type B". On the other hand, $A \multimap B$, the linear implication, reads "receive a channel of type A and then continue as B". Proposition $A \oplus B$ selects a branch of type A or B, while $A \& B$ offers the choice of A or B.

Hypersequents. Elements are types identified by variables, possibly marked by $\bullet$. Contexts are sets of elements, while hypersequents are sets of ILL sequents:

$$\textit{(Element)} \quad T ::= x{:}A \mid x{:}\bullet A \qquad \textit{(Contexts)} \quad \Delta, \Theta ::= \cdot \mid \Delta, T$$
$$\textit{(Hypersequents)} \quad \Psi ::= \Delta \vdash T \quad | \quad \Psi | \Psi$$

Contexts Δ and hypersequents Ψ are equivalent modulo associativity and commutativity. A hypersequent Ψ is the parallel composition of sequents. Given a sequent $\Delta \vdash T$, we call Δ its hypotheses and T its conclusion.

We make the standard assumption that a variable can appear at most once in any hypersequent, *unless* it is marked with $\bullet$. In LCC, bulleted variables appear exactly twice in a hypersequent, once as a hypothesis and once as a conclusion of two respective sequents which we say are then "connected". A provable hypersequent always has exactly one sequent with a non-bulleted conclusion, which we call the conclusion of the hypersequent. Similarly, we call non-bulleted hypotheses the hypotheses of the hypersequent. Intuitively, a provable hypersequent is a tree of sequents, whose root is the only sequent with a non-bulleted conclusion, and whose sequents have exactly one child for each of their bulleted hypotheses.

Processes and Choreographies. We give the syntax of our proof terms, or LCC programs, in Fig. 1. The syntax is an extension of that of the internal π-calculus with choreographic primitives. The internal π-calculus allows us to focus on a simple, yet very expressive fragment of the π-calculus [19], as in [7]. Terms can be *processes* performing I/O actions or *choreographies* of interactions.

Processes. An *(output)* $\overline{x}(y); (P|Q)$ sends a *fresh* name y over channel x and then proceeds with the parallel composition $P|Q$, whereas an *(input)* $x(y); P$ receives y over x and proceeds as P. In a *(left sel)* $x.\mathsf{inl}; P$, we send over channel x our choice of the left branch offered by the receiver. The term *(right sel)* $x.\mathsf{inr}; P$ selects the right branch instead. Selections communicate with the term *(case)* $x.\mathsf{case}(P, Q)$, which offers a left branch P and a right branch Q. The term *(par)* $P \mid_x P$ models parallel composition; here, differently from the output case, the two composed processes are not independent, but share the communication channel x. The term *(res)* is the standard restriction. Terms *(close)* and *(wait)*

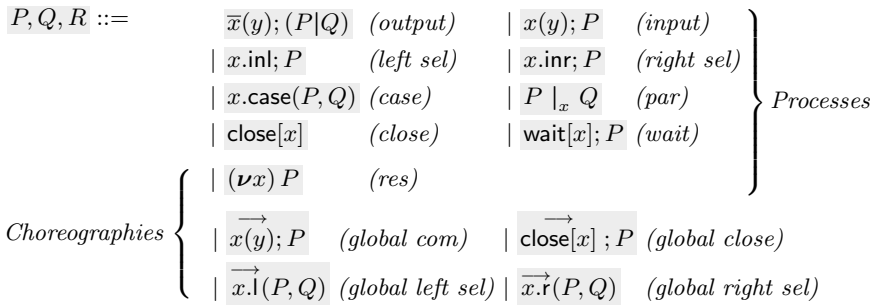

Fig. 1. LCC programs.

model, respectively, the request and acceptance for closing a channel, following real-world closing handshakes in communication protocols such as TCP.

Choreographies. The term *(res)*for name restriction is the same as for processes. A *(global com)* $\overrightarrow{x(y)};P$ describes a system where a fresh name y is communicated over a channel x, and then continues as P, where y is bound in P. The terms *(global left sel)* and *(global right sel)* model systems where, respectively, a left branch or a right branch is selected on channel x. Unused branches in global selections, e.g., Q in $\overrightarrow{x.\mathsf{l}}(P,Q)$, are unnecessary in our setting since they are never executed; however, their specification will be convenient for our technical development of endpoint projection, which will follow our concretisation transformation in LCC. Finally, term *(global close)* models the closure of a channel.

Note that, differently from § 2, we omit process identifiers in choreographies since our typing will make them redundant (cf. § 6).

Judgements. An LCC judgement has the form $P \triangleright \Psi$ where Ψ is a hypersequent and P is a proof term. If we regard LCC as a type theory for our term language, we say that the hypersequent Ψ types the term P.

3.1 Rules

The proof theory of LCC consists of the *action fragment* and the *interaction fragment*, which reason respectively about processes and choreographies.

Action Fragment. The action fragment includes ILL-style left and right rules, reported in Fig. 2, and the structural rules Conn and Scope, described separately.

Unit. The rules for unit are standard. Rule 1R types a process that requests to close channel x and terminates. Symmetrically, rule 1L types a process that waits for a request to close x, making sure that x does not occur in P.

Tensor. Rule $\otimes$R types the output $\overline{x}(y);(P|Q)$, combining the conclusions of the hypersequents of P and Q. The continuations P and Q will handle, respectively, the transmitted channel y and channel x. Ensuring that channels y and x are handled by different parallel processes avoids potential deadlocks caused by

$$\frac{P \triangleright \Psi_1 | \Delta_1 \vdash y : A \quad Q \triangleright \Psi_2 | \Delta_2 \vdash x : B}{\overline{x}(y); (P|Q) \triangleright \Psi_1 | \Psi_2 | \Delta_1, \Delta_2 \vdash x : A \otimes B} \otimes \mathsf{R} \qquad \frac{P \triangleright \Psi | \Delta, y : A, x : B \vdash T}{x(y); P \triangleright \Psi | \Delta, x : A \otimes B \vdash T} \otimes \mathsf{L}$$

$$\frac{P \triangleright \Psi | \Delta, y : A \vdash x : B}{x(y); P \triangleright \Psi | \Delta \vdash x : A \multimap B} \multimap \mathsf{R} \qquad \frac{P \triangleright \Psi_1 | \Delta_1 \vdash y : A \quad Q \triangleright \Psi_2 | \Delta_2, x : B \vdash T}{\overline{x}(y); (P|Q) \triangleright \Psi_1 | \Psi_2 | \Delta_1, \Delta_2, x : A \multimap B \vdash T} \multimap \mathsf{L}$$

$$\frac{}{\mathsf{close}[x] \triangleright \cdot \vdash x : \mathbf{1}} \mathsf{1R} \qquad \frac{P \triangleright \Psi | \Delta, x : A \vdash T}{x.\mathsf{inl}; P \triangleright \Psi | \Delta, x : A \& B \vdash T} \&\mathsf{L}_1 \qquad \frac{Q \triangleright \Psi | \Delta, x : B \vdash T}{x.\mathsf{inr}; Q \triangleright \Psi | \Delta, x : A \& B \vdash T} \&\mathsf{L}_2$$

$$\frac{P \triangleright \Psi | \Delta \vdash T}{\mathsf{wait}[x]; P \triangleright \Psi | \Delta, x : \mathbf{1} \vdash T} \mathsf{1L} \qquad \frac{P \triangleright \Psi | \Delta \vdash x : A}{x.\mathsf{inl}; P \triangleright \Psi | \Delta \vdash x : A \oplus B} \oplus\mathsf{R}_1 \qquad \frac{Q \triangleright \Psi | \Delta \vdash x : B}{x.\mathsf{inr}; Q \triangleright \Psi | \Delta \vdash x : A \oplus B} \oplus\mathsf{R}_2$$

$$\frac{P \triangleright \Psi | \Delta \vdash x : A \quad Q \triangleright \Psi | \Delta \vdash x : B}{x.\mathsf{case}(P, Q) \triangleright \Psi | \Delta \vdash x : A \& B} \&\mathsf{R} \qquad \frac{P \triangleright \Psi | \Delta, x : A \vdash T \quad Q \triangleright \Psi | \Delta, x : B \vdash T}{x.\mathsf{case}(P, Q) \triangleright \Psi | \Delta, x : A \oplus B \vdash T} \oplus\mathsf{L}$$

Fig. 2. Left and Right Rules of the Action Fragment.

their interleaving [7, 22]. Dually, rule $\otimes\mathsf{L}$ types an input $x(y); P$, by requiring the continuation to use channels y and x following their respective types.

Linear Implication. The proof term for rule $\multimap\,\mathsf{R}$ is an input $x(y); P$, meaning that the process needs to receive a name of type A before offering behaviour B on channel x. Rule $\multimap\,\mathsf{L}$ types the dual term $\overline{x}(y); (P|Q)$. Note that the prefixes in the proof terms are the same as for the tensor rules. This does not introduce ambiguity, since continuations are typed differently and thus it is never the case that both connectives could be used for typing the same term [7].

Additives. The rules for the additive connectives are standard. In a left selection $x.\mathsf{inl}; P$, we send over x our choice of the left branch offered by the receiver. The term $x.\mathsf{inr}; P$, is similar, but selects the right branch instead. Selections communicate with the term $x.\mathsf{case}(P, Q)$, which offers a left branch P and a right branch Q. In LCC, for example, rule $\&\mathsf{R}$ states that $x.\mathsf{case}(P, Q)$ provides x with type $A\&B$ whenever P and Q provide x with type A and B respectively.

Connection and Scoping. We pull apart the standard Cut rule of ILL, as (4) in § 2, and obtain two rules that depend on hypersequents as an interim place to store information. The first rule, Conn, merges two hypersequents by forming a connection:

$$\frac{P \triangleright \Psi_1 | \Delta_1 \vdash x : A \qquad\qquad Q \triangleright \Psi_2 | \Delta_2, x : A \vdash T}{P \mid_x Q \triangleright \Psi_1 | \Psi_2 | \Delta_1 \vdash x : \bullet A | \Delta_2, x : \bullet A \vdash T}\ \mathsf{Conn}$$

The proof term for Conn is parallel composition: in the conclusion, the two terms P and Q are composed in parallel and share channel x.

The second rule, called Scope, delimits the scope of a connection:

$$\frac{P \triangleright \Psi \mid \Delta_1 \vdash x : \bullet A \mid \Delta_2, x : \bullet A \vdash T}{(\boldsymbol{\nu} x)\, P \triangleright \Psi \mid \Delta_1, \Delta_2 \vdash T}\ \mathsf{Scope}$$

The proof term for Scope is a restriction of the scoped channel.

Interaction Fragment. Connections are first-class citizens in LCC and are object of logical reasoning. We give rules for composing connections, one for

each connective, which correspond to choreographies. Such rules form, together with rule Scope, the interaction fragment of LCC.

Unit. A connection of type **1** between two sequents can always be introduced:

$$\frac{P \triangleright \Psi | \Delta \vdash T}{\overrightarrow{\mathsf{close}}[x]; P \triangleright \Psi | \cdot \vdash x : \bullet\mathbf{1} | \Delta, x : \bullet\mathbf{1} \vdash T}\ \mathsf{1C}$$

Observe that the choreography term $\overrightarrow{\mathsf{close}}[x]; P$ describes the same behaviour as the process term $\mathsf{close}[x] \mid_x \mathsf{wait}[x]; P$, and indeed their typing is the same. In general, in LCC the typing of process terms and choreographic terms describing equivalent behaviour is the same. We will formalise this intuition in § 5.

Tensor. The connection rule for $\otimes$ combines two connections between three sequents. Technically, when two sequents $\Delta_1 \vdash y : \bullet A$ and $\Delta_2 \vdash x : \bullet B$ are connected to a third sequent $\Delta_3, y : \bullet A, x : \bullet B \vdash T$, we can merge the two connections into a single one, obtaining the sequents $\Delta_1, \Delta_2 \vdash x : \bullet A \otimes B$ and $\Delta_3, x : \bullet A \otimes B \vdash T$:

$$\frac{P \triangleright \Psi | \Delta_1 \vdash y : \bullet A | \Delta_2 \vdash x : \bullet B | \Delta_3, y : \bullet A, x : \bullet B \vdash T}{\overrightarrow{x(y)}; P \triangleright \Psi | \Delta_1, \Delta_2 \vdash x : \bullet A \otimes B | \Delta_3, x : \bullet A \otimes B \vdash T}\ \otimes\mathsf{C}$$

Rule $\otimes\mathsf{C}$ corresponds to typing a choreographic communication $\overrightarrow{x(y)}; P$. This rule is the formalisation in LCC of the cut reduction discussed in § 2. Term P will then perform communications on channel y with type A and x with type B.

Linear Implication. The rule for $\multimap$ manipulates connections with a causal dependency: if $\Delta_1 \vdash y : \bullet A$ is connected to $\Delta_2, y : \bullet A \vdash x : \bullet B$, which is connected to $\Delta_3, x : \bullet B \vdash T$, then $\Delta_2 \vdash x : \bullet A \multimap B$ is connected to $\Delta_1, \Delta_3, x : \bullet A \multimap B \vdash T$.

$$\frac{P \triangleright \Psi | \Delta_1 \vdash y : \bullet A | \Delta_2, y : \bullet A \vdash x : \bullet B | \Delta_3, x : \bullet B \vdash T}{\overrightarrow{x(y)}; P \triangleright \Psi | \Delta_2 \vdash x : \bullet A \multimap B | \Delta_1, \Delta_3, x : \bullet A \multimap B \vdash T}\ \multimap\mathsf{C}$$

Rule $\multimap\ \mathsf{C}$ types a communication $\overrightarrow{x(y)}; P$. The prefix $\overrightarrow{x(y)}$ is the same as that of rule $\otimes\mathsf{C}$, similarly to the action fragment for the connectives $\otimes$ and $\multimap$. Differently from rule $\otimes\mathsf{C}$, the usage of channel x in the continuation P has a causal dependency on y whereas in $\otimes\mathsf{C}$ the two channels proceed separately.

Additives. The rules for the additive connectives follow similar reasoning:

$$\frac{P \triangleright \Psi | \Psi' | \Delta_1 \vdash x : \bullet A | \Delta_2, x : \bullet A \vdash T \qquad Q \triangleright \Psi' | \Delta_1 \vdash x : B}{\overrightarrow{x.\mathsf{l}}(P, Q) \triangleright \Psi | \Psi' | \Delta_1 \vdash x : \bullet A \& B | \Delta_2, x : \bullet A \& B \vdash T}\ \&\mathsf{C}_1$$

$$\frac{P \triangleright \Psi | \Delta_1 \vdash x : A \qquad Q \triangleright \Psi | \Psi' | \Delta_1 \vdash x : \bullet B | \Delta_2, x : \bullet B \vdash T}{\overrightarrow{x.\mathsf{r}}(P, Q) \triangleright \Psi | \Psi' | \Delta_1 \vdash x : \bullet A \& B | \Delta_2, x : \bullet A \& B \vdash T}\ \&\mathsf{C}_2$$

$$\frac{P \triangleright \Psi | \Psi' | \Delta_1 \vdash x : \bullet A | \Delta_2, x : \bullet A \vdash T \qquad Q \triangleright \Psi' | \Delta_2, x : B \vdash T}{\overrightarrow{x.\mathsf{l}}(P, Q) \triangleright \Psi | \Psi' | \Delta_1 \vdash x : \bullet A \oplus B | \Delta_2, x : \bullet A \oplus B \vdash T}\ \oplus\mathsf{C}_1$$

$$\frac{P \triangleright \Psi | \Delta_2, x : A \vdash T \qquad Q \triangleright \Psi | \Psi' | \Delta_1 \vdash x : \bullet B | \Delta_2, x : \bullet B \vdash T}{\overrightarrow{x.\mathsf{r}}(P, Q) \triangleright \Psi | \Psi' | \Delta_1 \vdash x : \bullet A \oplus B | \Delta_2, x : \bullet A \oplus B \vdash T}\ \oplus\mathsf{C}_2$$

Rule $\&\mathsf{C}_1$ types a choreography that selects the left branch on x and then proceeds P, provided that x is not used in Q since the latter is unused.

We call C-rules the interaction rules for manipulating connections. C-rules represent of cut reductions in ILL, following the intuition presented in § 2.

Example 1. We formalise and extend our example from § 2 as follows:

$$
\begin{array}{lcl}
P_{\mathsf{client}'} & = & x.\mathsf{inr};\ \overline{x}(tea);\ \left(\ \mathsf{close}[tea]\ \mid\ x(tr);\ \overline{tr}(p);\ (\mathsf{close}[p] | \mathsf{wait}[tr];\ \mathsf{close}[x]\ )\ \right) \\
P_{\mathsf{server}'} & = & x.\mathsf{case}\left(\begin{array}{l} x(water);\ b.\mathsf{inl};\ \mathsf{wait}[water];\ \mathsf{wait}[x];\ \mathsf{close}[b], \\ x(tea);\ \overline{x}(tr);\ \begin{pmatrix} tr(p);\ \mathsf{wait}[tea];\ \mathsf{wait}[p];\ \mathsf{close}[tr]\ \mid \\ b.\mathsf{inr};\ \overline{b}(m);\left(\mathsf{close}[m]\ \mid\ \mathsf{wait}[x];\ \mathsf{close}[b]\right) \end{pmatrix} \end{array}\right) \\
P_{\mathsf{bank}'} & = & b.\mathsf{case}(\quad \mathsf{wait}[b];\ \mathsf{close}[z],\quad b(m);\ \mathsf{wait}[m];\ \mathsf{wait}[b];\ \mathsf{close}[z]\quad) \\
P & = & (\boldsymbol{\nu} x)\,(P_{\mathsf{client}'}\ |_x\ (\boldsymbol{\nu} b)\,(P_{\mathsf{server}'}\ |_b\ P_{\mathsf{bank}'})) \\
\hline
C & = & (\boldsymbol{\nu} x)\,(\boldsymbol{\nu} b)\ \overrightarrow{x.\mathsf{r}}\left(\begin{array}{l} x(water);\ b.\mathsf{inl};\ \mathsf{wait}[water];\ \mathsf{wait}[x];\ \mathsf{close}[b], \\ \\ \overrightarrow{x(tea)};\ \overrightarrow{x(tr)};\ \overrightarrow{tr(p)};\ \overrightarrow{b.\mathsf{r}}\begin{pmatrix} \mathsf{wait}[b];\ \mathsf{close}[z] \\ \\ \overrightarrow{b(m)};\ \mathsf{close}[tea, \overrightarrow{p, tr}, m, x, b] \end{pmatrix} \end{array}\right)
\end{array}
$$

Process $P_{\mathsf{client}'}$ implements a new version of the client, which selects the right choice of a branching on channel x and then asks for some tea; then, it proceeds as P_{client} from § 2. Note that we have enhanced the processes with all expected closing of channels. The server $P_{\mathsf{server}'}$, instead, now offers to the client a choice between buying a tea (as in § 2) and getting a free glass of water. Since the water is free, the payment to the bank is not performed in this case. In either case, the bank is notified of whether a payment will occur or not, respectively right and left branch in P_{bank}. The processes are composed as a system in P.

Term C is the equivalent choreographic representation of P. We can type channel x as $(\mathbf{string} \otimes \mathsf{end}) \oplus (\mathbf{string} \otimes (\mathbf{string} \multimap \mathsf{end}) \multimap \mathsf{end})$ in both C and P. The type of channel b is: $\mathsf{end} \oplus (\mathbf{string} \otimes \mathsf{end})$. For clarity, we have used concrete data types instead of the abstract basic type $\mathbf{1}$. □

4 Semantics

We now derive an operational semantics for LCC programs from our proof theory, by obtaining the standard relations of structural equivalence $\equiv$ and reduction $\rightarrow$ as theorems of LCC. For example, the π-calculus rule $(\boldsymbol{\nu} w)\,(P\ |_x\ Q) \equiv (\boldsymbol{\nu} w)\,P\ |_x\ Q$ (for $w \notin \mathsf{fn}(Q)$) can be derived as a proof transformation, since:

$$
\dfrac{\dfrac{P \triangleright \Psi |\ \Delta_1 \vdash y : \bullet D |\ \Delta, y : \bullet D \vdash x : A \qquad Q \triangleright \Psi' |\ \Delta', x : A \vdash T}{P\ |_x\ Q \triangleright \Psi |\ \Psi' |\ \Delta_1 \vdash y : \bullet D |\ \Delta, y : \bullet D \vdash x : \bullet A |\ \Delta', x : \bullet A \vdash T}\ \mathsf{Conn}}{(\boldsymbol{\nu} y)\,(P\ |_x\ Q) \triangleright \Psi |\ \Psi' |\ \Delta_1, \Delta \vdash x : \bullet A |\ \Delta', x : \bullet A \vdash T}\ \mathsf{Scope}
$$

is equivalent to $(\equiv)$

$$
\dfrac{\dfrac{P \triangleright \Psi |\ \Delta_1 \vdash y : \bullet D |\ \Delta, y : \bullet D \vdash x : A}{(\boldsymbol{\nu} y)\,P \triangleright \Psi |\ \Delta_1, \Delta \vdash x : A}\ \mathsf{Scope} \qquad Q \triangleright \Psi' |\ \Delta', x : A \vdash T}{(\boldsymbol{\nu} y)\,P\ |_x\ Q \triangleright \Psi |\ \Psi' |\ \Delta_1, \Delta \vdash x : \bullet A |\ \Delta', x : \bullet A \vdash T}\ \mathsf{Conn}
$$

[Scope/Conn/L]	$(\boldsymbol{\nu} y)\,(P \mid_x Q) \equiv (\boldsymbol{\nu} y)\,P \mid_x Q$	$\big(y \notin \mathsf{fn}(Q)\big)$
[Scope/Conn/R]	$(\boldsymbol{\nu} y)\,(P \mid_x Q) \equiv P \mid_x (\boldsymbol{\nu} y)\,Q$	$\big(y \notin \mathsf{fn}(P)\big)$
[Scope/Scope]	$(\boldsymbol{\nu} y)\,(\boldsymbol{\nu} x)\,P \equiv (\boldsymbol{\nu} x)\,(\boldsymbol{\nu} y)\,P$	
[Scope/1L]	$(\boldsymbol{\nu} x)\,\mathsf{wait}[y]; P \equiv \mathsf{wait}[y]; (\boldsymbol{\nu} x)\,P$	
[Scope/ $\otimes$ R/L], [Scope/ $\multimap$ L/L]	$(\boldsymbol{\nu} w)\,\overline{x}(y); (P \mid Q) \equiv \overline{x}(y); ((\boldsymbol{\nu} w)\,P \mid Q)$	$\big(w \notin \mathsf{fn}(Q)\big)$
[Scope/ $\otimes$ R/R], [Scope/ $\multimap$ L/R]	$(\boldsymbol{\nu} w)\,\overline{x}(y); (P \mid Q) \equiv \overline{x}(y); (P \mid (\boldsymbol{\nu} w)\,Q)$	$\big(w \notin \mathsf{fn}(P)\big)$
[Scope/ $\otimes$ L], [Scope/ $\multimap$ R]	$(\boldsymbol{\nu} w)\,x(y); P \equiv x(y); (\boldsymbol{\nu} w)\,P$	
[Scope/ $\oplus$ R_1], [Scope/&L_1]	$(\boldsymbol{\nu} w)\,x.\mathsf{inl}; P \equiv x.\mathsf{inl}; (\boldsymbol{\nu} w)\,P$	
[Scope/ $\oplus$ R_2], [Scope/&L_2]	$(\boldsymbol{\nu} w)\,x.\mathsf{inr}; P \equiv x.\mathsf{inr}; (\boldsymbol{\nu} w)\,P$	
[Scope/ $\oplus$ L], [Scope/&R]	$(\boldsymbol{\nu} w)\,x.\mathsf{case}(P, Q) \equiv x.\mathsf{case}((\boldsymbol{\nu} w)\,P\ ,\ (\boldsymbol{\nu} w)\,Q)$	
[Scope/1C]	$(\boldsymbol{\nu} w)\ \overrightarrow{\mathsf{close}}[x]; P \equiv \overrightarrow{\mathsf{close}}[x]; (\boldsymbol{\nu} w)\,P$	
[Scope/ $\otimes$ C], [Scope/ $\multimap$ C]	$(\boldsymbol{\nu} w)\ \overrightarrow{x(y)}; P \equiv \overrightarrow{x(y)}; (\boldsymbol{\nu} w)\,P$	
[Scope/ $\oplus$ C_1/L], [Scope/&C_1/L]	$(\boldsymbol{\nu} w)\ \overrightarrow{x.\mathsf{l}}(P, Q) \equiv \overrightarrow{x.\mathsf{l}}((\boldsymbol{\nu} w)\,P, Q)$	$\big(w \notin \mathsf{fn}(Q)\big)$
[Scope/ $\oplus$ C_1/L/R], [Scope/&C_1/L/R]	$(\boldsymbol{\nu} w)\ \overrightarrow{x.\mathsf{l}}(P, Q) \equiv \overrightarrow{x.\mathsf{l}}((\boldsymbol{\nu} w)\,P, (\boldsymbol{\nu} w)\,Q)$	$\big(w \in \mathsf{fn}(Q)\big)$
[Scope/ $\oplus$ C_2/R], [Scope/&C_2/R]	$(\boldsymbol{\nu} w)\ \overrightarrow{x.\mathsf{r}}(P, Q) \equiv \overrightarrow{x.\mathsf{r}}(P, (\boldsymbol{\nu} w)\,Q)$	$\big(w \notin \mathsf{fn}(P)\big)$
[Scope/ $\oplus$ C_2/L/R], [Scope/&C_2/L/R]	$(\boldsymbol{\nu} w)\ \overrightarrow{x.\mathsf{r}}(P, Q) \equiv \overrightarrow{x.\mathsf{r}}((\boldsymbol{\nu} w)\,P, (\boldsymbol{\nu} w)\,Q)$	$\big(w \in \mathsf{fn}(P)\big)$

Fig. 3. Commuting Conversions ($\equiv$) for Scope (Restriction).

4.1 Commuting Conversions ($\equiv$)

The structural equivalence of LCC ($\equiv$) is defined in terms of *commuting conversions*, i.e., admissible permutations of rule applications in proofs. In ILL, commuting conversions concern the cut rule. However, since in LCC the cut rule has been split into Scope and Conn, we need to introduce two sets of commuting conversions, one for rule Scope, and one for rule Conn. In the sequel, we report commuting conversions between proofs by giving the corresponding process and choreography terms (cf. [14] for the complete LCC proofs).

Commuting Conversions for Scope. Commuting conversions for Scope correspond to permuting restriction with other operators in LCC programs. We report them in Fig. 3, where we assume variables to be distinct. For example, [Scope/ $\otimes$ R/L] says that an application of rule Scope to the conclusion of rule $\otimes$R can be commuted so that we can apply $\otimes$R to the conclusion of Scope. Note that the top-level LCC terms of some cases are identical, e.g., [Scope/ $\otimes$ R/L] and [Scope/ $\multimap$ L/L], but the subterms are different since they have different typing.

Commuting Conversions for Conn. The commuting conversions for rule Conn, reported in Fig. 4, correspond to commuting the parallel operator with other terms. For example, rule [Conn/Conn] is the standard associativity of parallel in the π-calculus. Also, [Conn/ $\otimes$ C/L] says that $\overrightarrow{x(y)}$ in $\overrightarrow{x(y)}; P \mid_w Q$ can always be executed before Q as far as x and y do not occur in Q. This captures the concurrent behaviour of choreographies in [9]. Note that some of the rules are not standard for the π-calculus, e.g., [Conn/ $\multimap$ R/R], but this does not alter the intended semantics of parallel (cf. § 6, Semantics).

[Conn/Conn]	$(P \mid_y Q) \mid_x R \equiv P \mid_y (Q \mid_x R)$	
[Conn/1L/L]	$\mathsf{wait}[x]; P \mid_y Q \equiv \mathsf{wait}[x]; (P \mid_y Q)$	
[Conn/1L/R]	$P \mid_y \mathsf{wait}[x]; Q \equiv \mathsf{wait}[x]; (P \mid_y Q)$	
[Conn/ $\otimes$R/R/L], [Conn/$\multimap$L/R/L]	$P \mid_w \overline{x}(y); (Q \mid R) \equiv \overline{x}(y); ((P \mid_w Q) \mid R)$	
[Conn/ $\otimes$R/R/R], [Conn/$\multimap$L/R/R]	$P \mid_w \overline{x}(y); (Q \mid R) \equiv \overline{x}(y); (Q \mid (P \mid_w R))$	
[Conn/ $\otimes$ L/L]	$x(y); P \mid_w Q \equiv x(y); (P \mid_w Q)$	
[Conn/ $\otimes$ L/R], [Conn/ $\multimap$ R/R]	$P \mid_w x(y); Q \equiv x(y); (P \mid_w Q)$	
[Conn/ $\multimap$ L/L/R]	$\overline{x}(y); (P \mid Q) \mid_w R \equiv \overline{x}(y); (P \mid (Q \mid_w R))$	
[Conn/ $\oplus$ R$_1$/R], [Conn/&L$_1$/R]	$P \mid_w x.\mathsf{inl}; Q \equiv x.\mathsf{inl}; (P \mid_w Q)$	
[Conn/ $\oplus$ R$_2$/R], [Conn/&L$_2$/R]	$P \mid_w x.\mathsf{inr}; Q \equiv x.\mathsf{inr}; (P \mid_w Q)$	
[Conn/ $\oplus$ L/L]	$x.\mathsf{case}(P,Q) \mid_w R \equiv x.\mathsf{case}((P \mid_w R),(Q \mid_w R))$	
[Conn/ $\oplus$ L/R], [Conn/&R/R]	$P \mid_w x.\mathsf{case}(Q,R) \equiv x.\mathsf{case}((P \mid_w Q),(P \mid_w R))$	
[Conn/&L$_1$/L]	$x.\mathsf{inl}; P \mid_w Q \equiv x.\mathsf{inl}; (P \mid_w Q)$	
[Conn/&L$_2$/L]	$x.\mathsf{inr}; P \mid_w Q \equiv x.\mathsf{inr}; (P \mid_w Q)$	
[Conn/1C/L]	$\overrightarrow{\mathsf{close}}[x]; P \mid_w Q \equiv \overrightarrow{\mathsf{close}}[x]; (P \mid_w Q)$	
[Conn/1C/R]	$P \mid_w \overrightarrow{\mathsf{close}}[x]; Q \equiv \overrightarrow{\mathsf{close}}[x]; (P \mid_w Q)$	
[Conn/ $\otimes$ C/L], [Conn/ $\multimap$ C/L]	$\overrightarrow{x(y)}; P \mid_w Q \equiv \overrightarrow{x(y)}; (P \mid_w Q)$	$\left(y \notin \mathsf{fn}(Q)\right)$
[Conn/ $\otimes$C/R], [Conn/ $\multimap$ C/R]	$P \mid_w \overrightarrow{x(y)}; Q \equiv \overrightarrow{x(y)}; (P \mid_w Q)$	$\left(y \notin \mathsf{fn}(P)\right)$
[Conn/ $\oplus$ C$_1$/L]	$\overrightarrow{x.\mathsf{l}}(P,Q) \mid_w R \equiv \overrightarrow{x.\mathsf{l}}((P \mid_w R),(Q \mid_w R))$	$\left(w \in \mathsf{fn}(P) \cap \mathsf{fn}(Q)\right)$
[Conn/ $\oplus$ C$_1$/R], [Conn/&C$_1$/R]	$P \mid_w \overrightarrow{x.\mathsf{l}}(Q,R) \equiv \overrightarrow{x.\mathsf{l}}((P \mid_w Q),(P \mid_w R))$	$\left(w \in \mathsf{fn}(Q) \cap \mathsf{fn}(R)\right)$
[Conn/ $\oplus$C$_1$/R/L],[Conn/&C$_1$/R/L]	$P \mid_w \overrightarrow{x.\mathsf{l}}(Q,R) \equiv \overrightarrow{x.\mathsf{l}}((P \mid_w Q), R)$	$\left(w \in \mathsf{fn}(Q), w \notin \mathsf{fn}(R)\right)$
[Conn/ $\oplus$ C$_2$/L]	$\overrightarrow{x.\mathsf{r}}(P,Q) \mid_w R \equiv \overrightarrow{x.\mathsf{r}}((P \mid_w R),(Q \mid_w R))$	$\left(w \in \mathsf{fn}(P) \cap \mathsf{fn}(Q)\right)$
[Conn/ $\oplus$ C$_2$/R], [Conn/&C$_2$/R]	$P \mid_w \overrightarrow{x.\mathsf{r}}(Q,R) \equiv \overrightarrow{x.\mathsf{r}}((P \mid_w Q),(P \mid_w R))$	$\left(w \in \mathsf{fn}(Q) \cap \mathsf{fn}(R)\right)$
[Conn/ $\oplus$C$_2$/R/R],[Conn/&C$_2$/R/L]	$P \mid_w \overrightarrow{x.\mathsf{r}}(Q,R) \equiv \overrightarrow{x.\mathsf{r}}(Q, (P \mid_w R))$	$\left(w \notin \mathsf{fn}(Q), w \in \mathsf{fn}(R)\right)$
[Conn/&C$_1$/L]	$\overrightarrow{x.\mathsf{l}}(P,Q) \mid_w R \equiv \overrightarrow{x.\mathsf{l}}((P \mid_w R), Q)$	$\left(w \in \mathsf{fn}(P), w \notin \mathsf{fn}(Q)\right)$
[Conn/&C$_2$/L]	$\overrightarrow{x.\mathsf{r}}(P,Q) \mid_w R \equiv \overrightarrow{x.\mathsf{r}}(P, (Q \mid_w R))$	$\left(w \notin \mathsf{fn}(P), w \in \mathsf{fn}(Q)\right)$

Fig. 4. Commuting Conversions ($\equiv$) for Conn (Parallel Composition).

Since conversions preserve the concluding judgement of a proof, we have that:

Theorem 1 (Subject Congruence). *$P \triangleright \Psi$ and $P \equiv Q$ implies that $Q \triangleright \Psi$.*

4.2 Reductions ($\rightarrow$)

As for structural equivalence, we derive the reduction semantics for LCC programs from proof transformations. The obtained rules, reported in Fig. 5, are standard for both processes and choreographies (cf. [19, 9]): processes are reduced when they are the parallel composition of compatible actions, while choreographies can always be reduced. With an abuse of notation, we labelled each reduction with the channel it uses. Choreography reductions are also annotated with $\bullet$. We use t to range over labels of the form x or $\bullet x$, and $\tilde{t}$ to denote a sequence of such labels. As for commuting conversions, reductions preserve judgements:

$$[\beta_{\mathbf{1}}]\ (\boldsymbol{\nu} x)\,(\mathsf{close}[x] \mid_x \mathsf{wait}[x]; Q) \xrightarrow{x} Q$$
$$[\beta_{\otimes}]\ (\boldsymbol{\nu} x)\,(\overline{x}(y); (P|Q) \mid_x x(y); R) \xrightarrow{x} (\boldsymbol{\nu} y)\,(\boldsymbol{\nu} x)\,(P \mid_y (Q \mid_x R))$$
$$[\beta_{\multimap}]\ (\boldsymbol{\nu} x)\,(x(y); P \mid_x \overline{x}(y); (Q|R)) \xrightarrow{x} (\boldsymbol{\nu} x)\,(\boldsymbol{\nu} y)\,((Q \mid_y P) \mid_x R)$$
$$[\beta_{\oplus_1}]\ (\boldsymbol{\nu} x)\,(x.\mathsf{inl}; P \mid_x x.\mathsf{case}(Q, R)) \xrightarrow{x} (\boldsymbol{\nu} x)\,(P \mid_w Q)$$
$$[\beta_{\oplus_2}]\ (\boldsymbol{\nu} x)\,(x.\mathsf{inr}; P \mid_x x.\mathsf{case}(Q, R)) \xrightarrow{x} (\boldsymbol{\nu} x)\,(P \mid_x R)$$
$$[\beta_{\&_1}]\ (\boldsymbol{\nu} x)\,(x.\mathsf{case}(P, Q) \mid_x x.\mathsf{inl}; R) \xrightarrow{x} (\boldsymbol{\nu} x)\,(P \mid_x R)$$
$$[\beta_{\&_2}]\ (\boldsymbol{\nu} x)\,(x.\mathsf{case}(P, Q) \mid_x x.\mathsf{inr}; R) \xrightarrow{x} (\boldsymbol{\nu} x)\,(Q \mid_x R)$$
$$[\beta_{1\mathsf{C}}]\ (\boldsymbol{\nu} x)\ \overrightarrow{\mathsf{close}}[x]; P \xrightarrow{\bullet x} P \qquad [\beta_{\otimes\mathsf{C}}], [\beta_{\multimap\mathsf{C}}]\ (\boldsymbol{\nu} x)\ \overrightarrow{x(y)}; P \xrightarrow{\bullet x} (\boldsymbol{\nu} y)\,(\boldsymbol{\nu} x)\,P$$
$$[\beta_{\&\mathsf{C}_1}], [\beta_{\oplus\mathsf{C}_1}]\ (\boldsymbol{\nu} x)\ \overrightarrow{x.\mathsf{l}}(P, Q) \xrightarrow{\bullet x} (\boldsymbol{\nu} x)\,P \qquad [\beta_{\&\mathsf{C}_2}], [\beta_{\oplus\mathsf{C}_2}]\ (\boldsymbol{\nu} x)\ \overrightarrow{x.\mathsf{r}}(P, Q) \xrightarrow{\bullet x} (\boldsymbol{\nu} x)\,Q$$

Fig. 5. Reductions.

Theorem 2 (Subject Reduction). *$P \triangleright \Psi$ and $P \xrightarrow{t} Q$ implies that $Q \triangleright \Psi$.*

4.3 Scope Elimination (Normalisation)

We can use commuting conversions and reductions to permute and reduce all applications of Scope in a proof until the proof is Scope-free. Since applications of Scope correspond to restrictions in LCC programs, the latter can always progress until all communications on restricted channels are executed. We denote the reflexive and transitive closure of $\xrightarrow{t}$ up to $\equiv$ with $\xrightarrow{\tilde{t}}\!\!\!\!\rightarrow$.

Theorem 3 (Deadlock-freedom). *$P \triangleright \Psi$ implies there exist Q restriction-free and $\tilde{t}$ such that $P \xrightarrow{\tilde{t}}\!\!\!\!\rightarrow Q$ and $Q \triangleright \Psi$.*

5 Choreography Extraction and Endpoint Projection

In LCC, a judgement containing connections can be derived by either (i) using the action fragment, corresponding to processes, or (ii) using the interaction fragment, corresponding to choreographies. Consider the two following proofs:

$$\dfrac{\dfrac{\dfrac{}{\mathsf{close}[x] \triangleright \cdot \vdash x : \mathbf{1}}\,1\mathsf{R} \quad \dfrac{\dfrac{}{\mathsf{close}[y] \triangleright \cdot \vdash y : \mathbf{1}}\,1\mathsf{R}}{\mathsf{wait}[x]; \mathsf{close}[y] \triangleright x : \mathbf{1} \vdash y : \mathbf{1}}\,1\mathsf{L}}{\mathsf{close}[x] \mid_x \mathsf{wait}[x]; \mathsf{close}[y] \triangleright \cdot \vdash x : \bullet\mathbf{1} | x : \bullet\mathbf{1} \vdash y : \mathbf{1}}\,\mathsf{Conn}}{(\boldsymbol{\nu} x)\,(\mathsf{close}[x] \mid_x \mathsf{wait}[x]; \mathsf{close}[y]) \triangleright \cdot \vdash y : \mathbf{1}}\,\mathsf{Scope} \qquad \dfrac{\dfrac{\dfrac{}{\mathsf{close}[y] \triangleright \cdot \vdash y : \mathbf{1}}\,1\mathsf{R}}{\overrightarrow{\mathsf{close}}[x]\,; \mathsf{close}[y] \triangleright \cdot \vdash x : \bullet\mathbf{1} | x : \bullet\mathbf{1} \vdash y : \mathbf{1}}\,1\mathsf{C}}{(\boldsymbol{\nu} x)\,(\overrightarrow{\mathsf{close}}[x]\,; \mathsf{close}[y]) \triangleright \cdot \vdash y : \mathbf{1}}\,\mathsf{Scope}$$

The two proofs above reach the same hypersequent following, respectively, methodologies (i) and (ii). In this section, we formally relate the two methodologies, deriving procedures of choreography extraction and endpoint projection from proof equivalences. As an example, consider the following equivalence, $[\alpha\gamma_{\otimes}]$:

$$\dfrac{\dfrac{P \triangleright \Psi_1 | \Delta_1 \vdash y : A \quad Q \triangleright \Psi_2 | \Delta_2 \vdash x : B}{\overline{x}(y); (P|Q) \triangleright \Psi_1 | \Psi_2 | \Delta_1, \Delta_2 \vdash x : A \otimes B}\otimes\mathsf{R} \quad \dfrac{R \triangleright \Psi_3 | \Delta_3, y : A, x : B \vdash T}{x(y); R \triangleright \Delta_3, x : A \otimes B \vdash T}\otimes\mathsf{L}}{\overline{x}(y); (P|Q) \mid_x x(y); R \triangleright \Psi_1 | \Psi_2 | \Psi_3 | \Delta_1, \Delta_2 \vdash x : \bullet A \otimes B | \Delta_3, x : \bullet A \otimes B \vdash T}\,\mathsf{Conn}$$

$$
\begin{array}{lll}
[\alpha\gamma_{\mathbf{1}}] & \mathsf{close}[x] \mid_x \mathsf{wait}[x]; P & \overset{x}{\dashrightarrow} \ \overrightarrow{\mathsf{close}[x]}; P \\
[\alpha\gamma_{\otimes}] & \overline{x}(y); (P|Q) \mid_x x(y); R & \overset{x}{\dashrightarrow} \ \overrightarrow{x(y)}; (P \mid_y (Q \mid_x R)) \\
[\alpha\gamma_{\multimap}] & x(y); P \mid_x \overline{x}(y); (Q|R) & \overset{x}{\dashrightarrow} \ \overrightarrow{x(y)}; ((Q \mid_y P) \mid_x R) \\
[\alpha\gamma_{\&_1}] & x.\mathsf{case}(P,Q) \mid_x x.\mathsf{inl}; R & \overset{x}{\dashrightarrow} \ \overrightarrow{x.\mathsf{l}}((P \mid_x R), Q) \\
[\alpha\gamma_{\&_2}] & x.\mathsf{case}(P,Q) \mid_x x.\mathsf{inr}; R & \overset{x}{\dashrightarrow} \ \overrightarrow{x.\mathsf{r}}(P \ , \ Q \mid_x R) \\
[\alpha\gamma_{\oplus_1}] & x.\mathsf{inl}; P \mid_x x.\mathsf{case}(Q,R) & \overset{x}{\dashrightarrow} \ \overrightarrow{x.\mathsf{l}}((P \mid_x Q) \ , \ R) \\
[\alpha\gamma_{\oplus_2}] & x.\mathsf{inr}; P \mid_x x.\mathsf{case}(Q,R) & \overset{x}{\dashrightarrow} \ \overrightarrow{x.\mathsf{r}}(Q \ , \ (P \mid_x R))
\end{array}
$$

Fig. 6. Extraction and Projection.

can be extracted to ($\overset{x}{\dashrightarrow}$), can be projected from ($\overset{x}{\cdots\!\!\rightarrow}$)

$$
\dfrac{\dfrac{P \triangleright \Psi_1 | \Delta_1 \vdash y : A \qquad \dfrac{Q \triangleright \Psi_2 | \Delta_2 \vdash x : B \qquad R \triangleright \Psi_3 \mid \Delta_3, y : A, x : B \vdash T}{Q \mid_x R \triangleright \Psi_2 | \Psi_3 | \Delta_2 \vdash x : \bullet B | \Delta_3, y : A, x : \bullet B \vdash T}\ \mathsf{Conn}^x}{P \mid_y (Q \mid_x R) \triangleright \Psi_1 \mid \Psi_2 \mid \Psi_3 \mid \Delta_1 \vdash y : \bullet A \mid \Delta_2 \vdash x : \bullet B \mid \Delta_3, y : \bullet A, x : \bullet B \vdash T}\ \mathsf{Conn}^y}{\overrightarrow{x(y)}; (P \mid_y (Q \mid_x R)) \triangleright \Psi_1 \mid \Psi_2 \mid \Psi_3 \mid \Delta_1, \Delta_2 \vdash x : \bullet A \otimes B \mid \Delta_3, x : \bullet A \otimes B \vdash T}\ \otimes\mathsf{C}^x
$$

The equivalence $[\alpha\gamma_{\otimes}]$ allows to transform a proof of a connection of type $A \otimes B$ from the action fragment into an equivalent proof in the interaction fragment, and vice versa. We report the equivalences for extraction and projection in Fig. 6, presenting their proof terms. We read these equivalences from left to right for extraction, denoted by $\dashrightarrow$, and from right to left for projection, denoted by $\cdots\!\!\rightarrow$. Note how a choreography term corresponds to the parallel composition of two processes with the same behaviour. It is also clear why the unselected process Q in $\overrightarrow{x.\mathsf{l}}(P, Q)$ is necessary for projecting the corresponding case process.

Using commuting conversions, extraction can always be applied to proofs containing instances of Conn, i.e., programs containing subterms of the form $P \mid_x Q$. Similarly, projection can always be applied to proofs with instances of a C-rule, i.e., programs with choreography interactions. We denote the reflexive and transitive closure of $\overset{x}{\dashrightarrow}$ up to $\equiv$ with $\overset{\tilde{x}}{\dashrightarrow}\!\!\!\rightarrow$ (resp. $\overset{\tilde{x}}{\cdots\!\!\rightarrow}\!\!\!\rightarrow$ for $\overset{x}{\cdots\!\!\rightarrow}$).

Theorem 4 (Extraction and Projection). *Let $P \triangleright \Psi$. Then:*

(choreography extraction) $P \overset{\tilde{x}}{\dashrightarrow}\!\!\!\rightarrow Q$ for some $\tilde{x}$ and Q such that $Q \triangleright \Psi$ and Q does not contain subterms of the form $R \mid_x R'$;

(endpoint projection) $P \overset{\tilde{x}}{\cdots\!\!\rightarrow}\!\!\!\rightarrow Q$ for some $\tilde{x}$ and Q such that $Q \triangleright \Psi$ and Q does not contain choreography terms.

Example 2. Using the equivalences in Fig. 6 and $\equiv$, we can transform the processes to the choreography in Example 1 and vice versa. □

We now present the main property guaranteed by LCC: the extraction and projection procedures preserve the semantics of the transformed programs.

Theorem 5 (Correspondence). *Let* $P \triangleright \Psi$ *and* P' *be restriction-free. Then:*
(choreography extraction) $P \xrightarrow{\tilde{x}} P'$ *implies* $P \overset{\tilde{x}}{\dashrightarrow} Q$ *such that* $Q \xrightarrow{\bullet\tilde{x}} P'$.
(endpoint projection) $P \xrightarrow{\bullet\tilde{x}} P'$ *implies* $P \overset{\tilde{x}}{\cdots\!\!\gg} Q$ *such that* $Q \xrightarrow{\tilde{x}} P'$.

6 Related Work and Discussion

Related Work. Our action fragment is inspired by π-DILL [7]. The key difference is that we split rule Cut into Conn and Scope, which allows us to (i) reason about choreographies and (ii) type processes where restriction and parallel are used separately. Extra typable processes are always convertible to those where a Conn is immediately followed by a Scope, hence equivalent to those in [7]. Wadler [22] introduces a calculus where processes correspond to proofs in classical linear logic. We conjecture that LCC can be adapted to the classical setting.

Our commuting conversions can be seen as a logical characterisation of *swapping* [9], which permutes independent communications in a choreography. Previous works [12, 8, 9, 15] have formally addressed choreographies and EPP but without providing choreography extraction. Choreography extraction is a known hard problem [4], and our work is the first to address it for a language supporting channel passing. Probably, the work closest to ours wrt extraction is [13], where global types are extracted from session types; choreographies are more expressive than global types, since they capture the interleaving of different sessions. In the future, we plan to address standard features supported by [8, 9, 15] such as multiparty sessions, asynchrony, replicated services and nondeterminism.

Our mixing of choreographies with processes is similar to that found in [3] for global protocols and [15] for choreographies. The work [3] deals with the simpler setting of protocols, whereas we handle programs supporting name passing and session interleaving, both nontrivial problems [6, 9, 15]. The type system in [15] does not keep information on where the endpoints of connections are actually located as in our hypersequents, which enables extraction in our setting.

Process identifiers. In standard choreography calculi, the processes involved in a communication are usually identified explicitly as in the choreography (2) in § 2 [12, 8, 9, 15]. In LCC, processes are implicitly identified in judgements by using separate sequents in a hypersequent. Omitting process identifiers is thus just a matter of presentational convenience: a way of retaining them would be to annotate each sequent in a hypersequent with a process identifier (cf. [14]).

Exponentials and Infinite Behaviour. Our work focuses on the multiplicative and additive fragments of linear logic, but we conjecture that the known cut rule for exponentials can be split into a connection rule and a scope rule such as the ones for the linear case. We believe that the results in this paper can be generalised to exponentials without altering its foundations. A logical characterisation of infinite behaviour for ICC may similarly be added to our framework, following the developments in [20]. We leave both extensions as future work.

ILL. LCC is a generalisation of ILL, since we can represent any instance of the Cut rule in ILL with consecutive applications of rules Conn and Scope.

Semantics. LCC includes more term equivalences than the π-calculus, e.g., [Conn / $\multimap$ R/R/2] in Fig. 4. We inherit this from linear logic [22]. However, the extra equivalences do not produce any new reductions in well-typed systems (cf. [16]).

References

1. Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0/.
2. A. Avron. Hypersequents, logical consequence and intermediate logics for concurrency. *Ann. Math. Artif. Intell.*, 4:225–248, 1991.
3. P. Baltazar, L. Caires, V. T. Vasconcelos, and H. T. Vieira. A Type System for Flexible Role Assignment in Multiparty Communicating Systems. In *TGC*, 2012.
4. S. Basu and T. Bultan. Choreography conformance via synchronizability. In *WWW*, pages 795–804, 2011.
5. S. Basu, T. Bultan, and M. Ouederni. Deciding choreography realizability. In *POPL*, pages 191–202, 2012.
6. L. Bettini, M. Coppo, L. D'Antoni, M. D. Luca, M. Dezani-Ciancaglini, and N. Yoshida. Global progress in dynamically interleaved multiparty sessions. In *CONCUR*, volume 5201 of *LNCS*, pages 418–433. Springer, 2008.
7. L. Caires and F. Pfenning. Session types as intuitionistic linear propositions. In *CONCUR*, pages 222–236, 2010.
8. M. Carbone, K. Honda, and N. Yoshida. Structured communication-centered programming for web services. *ACM TOPLAS*, 34(2):8, 2012.
9. M. Carbone and F. Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.
10. J.-Y. Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987.
11. K. Honda, V. Vasconcelos, and M. Kubo. Language primitives and type disciplines for structured communication-based programming. In *ESOP'98*, volume 1381 of *LNCS*, pages 22–138, Heidelberg, Germany, 1998. Springer-Verlag.
12. I. Lanese, C. Guidi, F. Montesi, and G. Zavattaro. Bridging the gap between interaction- and process-oriented choreographies. In *Proc. of SEFM*, pages 323–332. IEEE, 2008.
13. J. Lange and E. Tuosto. Synthesising choreographies from local session types. In *CONCUR*, pages 225–239, 2012.
14. F. Montesi. *Choreographic Programming.* Ph.D. thesis, IT University of Copenhagen, 2013. http://www.itu.dk/people/fabr/papers/phd/thesis.pdf.
15. F. Montesi and N. Yoshida. Compositional choreographies. In *CONCUR*, pages 425–439, 2013.
16. J. A. Pérez, L. Caires, F. Pfenning, and B. Toninho. Linear logical relations for session-based concurrency. In *ESOP*, pages 539–558, 2012.
17. Z. Qiu, X. Zhao, C. Cai, and H. Yang. Towards the theoretical foundation of choreography. In *WWW*, pages 973–982. IEEE, 2007.
18. D. Sangiorgi. pi-calculus, internal mobility, and agent-passing calculi. *Theor. Comput. Sci.*, 167(1&2):235–274, 1996.
19. D. Sangiorgi and D. Walker. *The π-calculus: a Theory of Mobile Processes.* Cambridge University Press, 2001.
20. B. Toninho, L. Caires, and F. Pfenning. Higher-order processes, functions, and sessions: A monadic integration. In *ESOP*, pages 350–369, 2013.
21. W3C WS-CDL Working Group. Web services choreography description language version 1.0. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/, 2004.
22. P. Wadler. Propositions as sessions. In *ICFP*, pages 273–286, 2012.