Modular Compilation for Higher-Order Functional Choreographies

Cruz-Filipe, Luís; Graversen, Eva; Lugović, Lovro; Montesi, Fabrizio; Peressotti, Marco

Published in:
37th European Conference on Object-Oriented Programming (ECOOP 2023)

DOI:
10.4230/LIPIcs.ECOOP.2023.7

Publication date:
2023

Document version:
Final published version

Document license:
CC BY

Citation for pulished version (APA):
Cruz-Filipe, L., Graversen, E., Lugović, L., Montesi, F., & Peressotti, M. (2023). Modular Compilation for Higher-Order Functional Choreographies. In A. Karim, & S. Guido (Eds.), *37th European Conference on Object-Oriented Programming (ECOOP 2023)* (Vol. 263). [7] Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. Leibniz International Proceedings in Informatics, LIPIcs https://doi.org/10.4230/LIPIcs.ECOOP.2023.7

Modular Compilation for Higher-Order Functional Choreographies

Luís Cruz-Filipe
Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark

Eva Graversen
Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark

Lovro Lugović
Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark

Fabrizio Montesi
Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark

Marco Peressotti
Department of Mathematics and Computer Science,
University of Southern Denmark, Odense, Denmark

Abstract

Choreographic programming is a paradigm for concurrent and distributed software, whereby descriptions of the intended communications (choreographies) are automatically compiled into distributed code with strong safety and liveness properties (e.g., deadlock-freedom).

Recent efforts tried to combine the theories of choreographic programming and higher-order functional programming, in order to integrate the benefits of the former with the modularity of the latter. However, they do not offer a satisfactory theory of compilation compared to the literature, because of important syntactic and semantic shortcomings: compilation is not modular (editing a part might require recompiling everything) and the generated code can perform unexpected global synchronisations.

In this paper, we find that these shortcomings are not mere coincidences. Rather, they stem from genuine new challenges posed by the integration of choreographies and functions: knowing which participants are involved in a choreography becomes nontrivial, and divergence in applications requires rethinking how to prove the semantic correctness of compilation.

We present a novel theory of compilation for functional choreographies that overcomes these challenges, based on types and a careful design of the semantics of choreographies and distributed code. The result: a modular notion of compilation, which produces code that is deadlock-free and correct (it operationally corresponds to its source choreography).

2012 ACM Subject Classification Theory of computation → Lambda calculus; Theory of computation → Distributed computing models; Computing methodologies → Distributed programming languages

Keywords and phrases Choreographies, Concurrency, λ-calculus, Type Systems

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.7

Related Version *Full Version*: `https://arxiv.org/abs/2111.03701`

Funding This work was partially supported by Villum Fonden, grants no. 29518 and 50079, and the Independent Research Fund Denmark, grant no. 0135-00219.

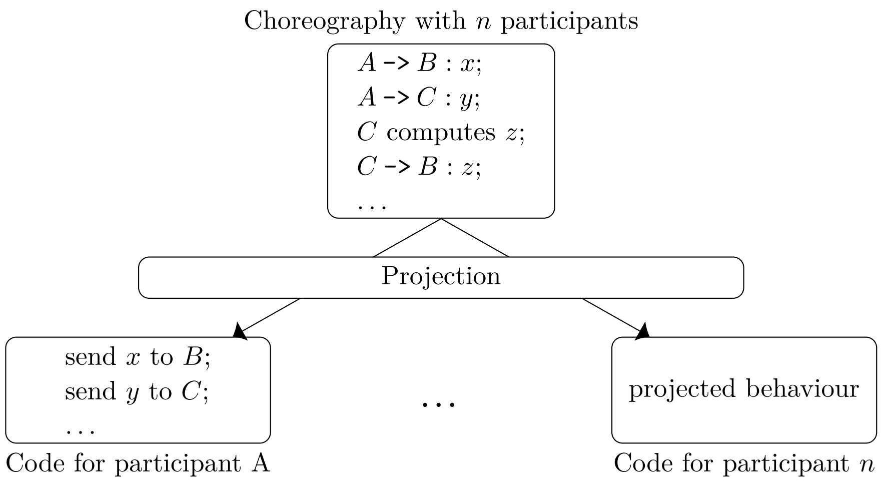

Figure 1 Choreographic programming: the communication and computation behaviour of a system is defined in a choreography, which is then projected (compiled) to deadlock-free distributed code (adapted from [17]).

1 Introduction

Functional and choreographic programming

Higher-order functional programming is a popular paradigm, which allows programmers to write modular code with strong guarantees through types. However, when dealing with concurrent and distributed programs, functional programming still requires developers to manually write a separate program for each participant, using send and receive actions to communicate data. This makes it easy to write programs that deadlock, or perform in other unexpected ways [22].

Choreographic programming (Figure 1) is a simple and powerful method to produce distributed code that does what it is supposed to do [23, 21, 18]. In this paradigm, programs are choreographies: structured compositions of the intended communications and computations that participants should perform, given from a joint perspective. A communication is expressed in some variation of the communication term from security protocol notation, Alice -> Bob: M, which reads "Alice communicates the message M to Bob" [26]. Given a choreography, a compiler produces executable distributed code. In the theory of choreographies, this compilation is called Endpoint Projection (EPP) [1]. A correct EPP has the powerful consequence of guaranteeing deadlock-freedom "for free": it is syntactically impossible to specify mismatched communication actions in choreographies, so the resulting distributed code cannot get stuck (deadlock-freedom by design) [2].

Recently, there have been two attempts at developing theories that combine the paradigms of choreographic and functional programming, in the hope of reaping the benefits of both [18, 6]. Finding an adequate notion of EPP in this setting has been an issue. In [6] the λ-calculus is extended with choreographic primitives for communications, yielding a simple yet expressive model called Chorλ, but no EPP is presented. In [18] an EPP is given for a choreographic language that extends a standard imperative choreographic language with primitives for abstraction and application (for higher-order composition). However, this theory comes at two important costs when compared to the expected properties of choreographic programming [24]. First, EPP is not modular: changing a part of a choreography that involves only some participants can change also the code projected for other participants. This means that updating a choreography requires reprojecting and redeploying the entire system, which is not necessary in previous work. Second, participants perform more synchronisations than those written in the choreography. This breaks the design principle that all communications are made syntactically manifest in choreographies.

These issues are not consequences of careless work. Rather, we find that they are both caused by a novel challenge that arises precisely from the combination of functional and choreographic programming – explained in the next paragraph. The aim of this work is to develop a new theory that overcomes this challenge.

The problem

When projecting a choreography to a participant, say Alice, the parts of the choreography not involving Alice should be ignored [1, 24]. Doing this is simple with traditional imperative choreographies, which are essentially sequences of commands $(c_1; c_2; \ldots)$. For each command: if the participant that we are projecting for is involved, we return some (appropriate) code; otherwise, we just skip the command and go to the next. For example, given the choreography Carol -> Bob: M; Alice -> Bob: M', a standard EPP would produce for Alice only the code to execute the second command (a send action towards Bob).

In a higher-order functional setting, checking if a participant is "involved" in a choreographic term is not an easy syntactic check anymore. Consider a choreography C that takes another choreography x as parameter, runs it, and communicates the result from Alice to Bob. Since x can be an arbitrary choreography, the participants involved in C are known only after x is instantiated. This is the technical issue that makes defining EPP for functional choreographies nontrivial. In [18], the proposed solution sacrifices modularity: every function application is projected to all participants, who then have to perform a global system synchronisation for every function call.

This work

We define a notion of EPP for Chorλ, capitalising on the design of its type system and semantics.

We start our development by focusing on the finite fragment Chorλ, i.e., without recursion. First, we introduce a target language for representing distributed code: a distributed λ-calculus, which consists of well-known terms extended with primitives for sending and receiving messages. Then, we use this language to define a modular EPP for (finite) Chorλ. The key insight for achieving modularity is the inclusion of a no-op term in the target language, which is the projection of any choreographic term in which a participant is not involved. In this way, if some choreographic subterm does not involve a participant p, it is projected as no-op. And if this term is later edited without involving p, then the projection for p remains no-op and does not need to be recompiled. This is explained in detail in Example 6.

The rule for generating no-ops benefits from the careful design of the rule for typing abstractions in Chorλ. This is not an accident: in [6] this particular rule was claimed to be designed with the future development of a suitable EPP in mind, but this was not substantiated. In this paper we show that our EPP satisfies the expected operational correspondence between choreographies and their projections (Theorems 25 and 26). As a consequence, projections of choreographies cannot deadlock.

Furthermore, we define a type system for the target language based on standard techniques, and show that well-typed choreographies are projected onto well-typed target terms whose types are projections of the source choreographic types (Proposition 10). This result is relevant for applicability: knowing the type of projected functions lets programmers compose them in larger projects through APIs under the control of the programmer, as is commonly done with projected code [15, 17].

A unique feature of Chorλ is that conditionals can use whole choreographies as conditions, and in particular ones that return distributed data structures – data structures that compose data residing at different participants. For the first time, our EPP leverages this feature to offer a new method for capturing *knowledge of choice* – distributed agreement regarding choices between alternative choreographic behaviours [4]. Specifically, we can statically guarantee that two (or more) participants will agree on the instantiation of a sum type (representing alternative choices) solely by performing independent local checks. When this is used in a conditional, it means that all participants are guaranteed to make the same choice at runtime. This gives a simpler alternative to existing verification methods for distributed choices [21]. We call types used in this way *distributed choice types*.

Lastly, we extend our development to the full language of Chorλ, including recursion. Recursion allows for divergent behaviour, which gives an interesting problem: a divergent term does not necessarily involve all participants, so generalising the operational correspondence between choreographies and their projections requires allowing choreographies to perform actions involving participants that are not blocked by divergent computations. The semantics of Chorλ include rules for performing reductions out of order, which again were designed with the future development of EPP in mind. We show that these rules are adequate to generalise our results.

Contribution

We define the first notion of EPP for a functional choreographic programming language that is modular and does not add extra communications. This necessitates using not only the information contained in the syntactic structure of a choreography, but also the one contained in the typing derivation that accompanies it. These sources of information give a number of cases for projection that need to be designed carefully, in order to distinguish correctly when a process is potentially involved in the realisation of part of a choreography. We show that EPP satisfies the usual operational correspondence property between choreographies and their projections. Our development also proves two unsubstantiated claims from [6]: that the typing system of Chorλ is expressive enough to support a modular notion of EPP, and that the semantics of Chorλ capture how distributed participants behave in the presence of divergence. Furthermore, we check the practical applicability of our theory by using it to project the model of the Extensible Authentication Protocol (EAP) [28] given in [6], a nontrivial choreography that makes use of higher-order composition, distributed data structures, and distributed choice types.

We anticipate that our developments on the theory of higher-order choreographies will allow higher-order functions to be added to implementations of existing choreographic and similar languages. We discuss this in Section 7.

Structure

We provide a review of the main features of recursion-free Chorλ in Section 2. In Section 3 we describe the local endpoint language Chorλ is projected to and how to project a choreography. We reintroduce recursion into Chorλ and introduce it to our endpoint language in Section 4. An example of a realistic use case (the Extensible Authentication Protocol) projected using our method can be seen in Section 5. Related work is given in Section 6. Conclusions are presented in Section 7. Full definitions and proofs of results for the full language of Chorλ can be found in Appendix A.

2 Background

In this section, we recap the theory of the choreographic λ-calculus (Chorλ) without recursion, from [6]. Chorλ extends the simply typed λ-calculus [5] with primitives that make distribution and communication syntactically manifest.

System model

Chorλ is used to model systems of independent processes, which can interact by synchronous communication. Each process has a name, and knows the names of the other processes in the network. There are two kinds of messages that can be exchanged: *values* are results of computations; and *selection labels* are special constants used to implement agreement on choices about alternative distributed behaviour.

Syntax

The syntax of Chorλ is given by the following grammar

$$\begin{aligned}
M &::= V \mid M\ M \mid \textbf{case}\ M\ \textbf{of Inl}\ x \Rightarrow M\textbf{; Inr}\ x \Rightarrow M \mid \textbf{select}_{\mathsf{p,p}}\ l\ M\\
V &::= x \mid \lambda x : T.M \mid \textbf{Inl}\ V \mid \textbf{Inr}\ V \mid \textbf{fst} \mid \textbf{snd} \mid \textbf{Pair}\ V\ V \mid ()@\mathsf{p} \mid \textbf{com}_{\mathsf{p,p}}\\
T &::= T \rightarrow_\rho T \mid T + T \mid T \times T \mid ()@\mathsf{p}
\end{aligned}$$

where M is a choreography, V is a value, T is a type, x is a variable, l is a label, p is a process name, and ρ is a set of process names.

Terms are located at processes, to reflect distribution. For example, the value $()@\mathsf{A}$ reads "the unit value at A". Types are annotated with process names, as well. In the typing rules of Chorλ (shown later), term $()@\mathsf{A}$ has the type $()@\mathsf{A}$, read "the unit type at A". In our examples, for simplicity, we assume the presence of primitives for integer values and an integer type $\mathsf{Int}@\mathsf{p}$ ("an integer at p") – the formal treatment of these are straightforward and similar to that of units.

Abstraction $\lambda x : T.M$, variable x and application MM are as in the standard (simply typed) λ-calculus. Sums and products are constructed, respectively, by using **Inl**/**Inr** and **Pair**. They are deconstructed in the usual way, respectively with **case** and **fst**/**snd**. The constructors can take only values as arguments, but this does not restrict expressivity (cf. [6]).

The primitives $\textbf{com}_{\mathsf{p,q}}$ and $\textbf{select}_{\mathsf{p,q}}\ l\ M$ (where p and q are process names) model communications of, respectively, values and selection labels. A *communication* term $\textbf{com}_{\mathsf{p,q}}$ acts as a function that takes a value at the process named p and returns the same value at the process named q. In a *selection* term $\textbf{select}_{\mathsf{p,q}}\ l\ M$, instead, p informs q that it has selected the label l before continuing as M. Selections choreographically represent the communication of an internal choice made by p to q. As we shall see in our definition of EPP, they play a key role in establishing agreement among processes regarding what behaviour they should enact together.

Selections are standard in choreographic languages and should not to be confused with the distributed choice types that we anticipated in the introduction (these will be illustrated later, in the next section). The former used to implement agreement on choices, whereas the latter are used to codify the information that an agreement has been reached and can thus be used without requiring communication. We will touch on this topic later, in Example 15 and Section 5.

A key feature of Chorλ is distributed data structures. For example, **Pair** $()@\mathsf{p}$ $()@\mathsf{q}$ is a distributed pair where the first element resides at p and the second at q. Types record the distribution of values across processes: if p occurs in the type given to V then part of V will be located at p. A function may involve more processes than those listed in the types of its input and output, so the type of abstractions $T \to_\rho T'$ has the extra ingredient ρ, which denotes the processes that may participate in the computation of the function besides those occurring in T or T'. We simply write $T \to T'$ in place of $T \to_\varnothing T'$. For example, if Alice wants to communicate an integer to Bob directly (without intermediaries), she can use a choreography of type $\mathsf{Int}@\mathsf{Alice} \to \mathsf{Int}@\mathsf{Bob}$; however, if the communication might go through a proxy, then she can use a choreography of type $\mathsf{Int}@\mathsf{Alice} \to_{\{\mathsf{Proxy}\}} \mathsf{Int}@\mathsf{Bob}$. The information given by ρ gives control on what processes may participate in choreographies taken as arguments. As we show in Section 3, this information is essential to achieve a modular EPP.

We write $\mathrm{fv}(M)$ for the set of free variables in a term M, and $\mathrm{pn}(T)$ and $\mathrm{pn}(M)$ for the set of process names mentioned in respectively a type T and a choreography M. A choreography is *closed* if it has no free variables. Our key results apply to closed choreographies.

▶ **Example 1** (Remote Function [6]). The following choreography models a distributed computation in which a client, C sends an integer *val* to a server S and a local function *fun* located at S is applied to *val* before the result gets returned to C. The choreography is parametrised on both *fun* and *val*.

$$\lambda \mathit{fun} : \mathsf{Int}@\mathsf{S} \to_\varnothing \mathsf{Int}@\mathsf{S}.\ \lambda \mathit{val} : \mathsf{Int}@\mathsf{C}.\ \mathbf{com}_{\mathsf{S},\mathsf{C}}\ (\mathit{fun}\ (\mathbf{com}_{\mathsf{C},\mathsf{S}}\ \mathit{val}))$$

⌟

Typing

Choreographies are typed with judgements of the form $\Theta; \Gamma \vdash M : T$, where Θ is the set of process names that can be used for typing M and Γ is a function assigning types to variables. We recall a few key typing rules from [6]. Our rules use the notation $\mathrm{pn}(T)$ for the process names that appear in the type T.

$$\frac{\mathrm{pn}(T) = \{\mathsf{p}\} \quad \{\mathsf{p}, \mathsf{q}\} \subseteq \Theta}{\Theta; \Gamma \vdash \mathbf{com}_{\mathsf{p},\mathsf{q}} : T \to_\varnothing T[\mathsf{p} := \mathsf{q}]}\ [\text{TCom}]$$

$$\frac{\Theta; \Gamma \vdash N : T \to_\rho T' \quad \Theta; \Gamma \vdash M : T}{\Theta; \Gamma \vdash N\ M : T'}\ [\text{TApp}]$$

$$\frac{\Theta'; \Gamma, x : T \vdash M : T' \quad \rho \cup \mathrm{pn}(T) \cup \mathrm{pn}(T') = \Theta' \subseteq \Theta}{\Theta; \Gamma \vdash \lambda x : T.M : T \to_\rho T'}\ [\text{TAbs}]$$

A communication is typed as a function from any type T located entirely at the sender p to the same type moved to the receiver, as long as both process names are in Θ. Application and abstraction are typed similarly to simply-typed λ-calculus, extended with ρ and Θ (whose consistency is checked in rule TAbs). Note that ρ and Θ in rule TAbs are not necessarily minimal, and it is possible to type, e.g., $\{\mathsf{p}, \mathsf{q}\}; \varnothing \vdash \lambda x : \mathsf{Int}@\mathsf{p}.x : \mathsf{Int}@\mathsf{p} \to_{\{\mathsf{q}\}} \mathsf{Int}@\mathsf{p}$. A minimal ρ would consist of those processes that appear either in M or in the types of the free variables of M according to Γ.

▶ **Example 2.** Let h be the function $\lambda x : \mathsf{Int}@\mathsf{Alice}.\mathbf{com}_{\mathsf{Proxy},\mathsf{Bob}}\ (\mathbf{com}_{\mathsf{Alice},\mathsf{Proxy}}\ x)$, which communicates an integer from Alice to Bob by passing through an intermediary Proxy. Then, $\{\mathsf{Alice}, \mathsf{Bob}, \mathsf{Proxy}\}; \varnothing \vdash h : \mathsf{Int}@\mathsf{Alice} \to_{\{\mathsf{Proxy}\}} \mathsf{Int}@\mathsf{Bob}$. For any term M, the composition

$h\ M$ is well-typed if M has type $\mathsf{Int@Alice}$, denoting that the evaluation of M will yield an integer at Alice. By contrast, $h\ 5@\mathsf{Bob}$ is ill-typed because of wrong data locality (the argument is not at the process expected by h). ⌟

Semantics

Chorλ comes with an operational semantics given in terms of labelled reductions. Reduction labels are used to keep track of which processes interact in a reduction, which is going to be important for our development. We illustrate this with the two key rules below.

$$\frac{}{\lambda x : T.M\ V \xrightarrow{\varnothing} M[x := V]}\ [\textsc{AppAbs}] \qquad \frac{\mathrm{fv}(V) = \varnothing}{\mathbf{com}_{\mathsf{q},\mathsf{p}}\ V \xrightarrow{\{\mathsf{q},\mathsf{p}\}} V[\mathsf{q} \mapsto \mathsf{p}]}\ [\textsc{Com}]$$

Rule AppAbs is the standard application rule of call-by-value λ-calculus – annotated with an empty set, which indicates that no synchronisation is taking place. Rule Com, instead, implements a communication by "moving" the communicated value from the sender to the receiver (through a substitution). Thus, for example, $\mathbf{com}_{\mathsf{Alice},\mathsf{Bob}} 3@\mathsf{Alice} \xrightarrow{\{\mathsf{Alice},\mathsf{Bob}\}} 3@\mathsf{Bob}$. Since it makes no sense to communicate a variable whose value is stored at the sender rather than the value itself, we require that the communicated value has no free variables. Communicating a free variable would cause problems for Chorλ's type system, since it would require changing the type of the variable in the environment.

Reductions are labelled with the processes synchronising in them, but this only becomes relevant information in Section 4.

3 Endpoint Projection (EPP) for finite Chorλ

In this section we develop a theory of EPP for finite Chorλ.

3.1 Process Language

We write implementations of choreographies in a distributed λ-calculus, which we call process language. Processes run in parallel, each with its own behaviour, and can interact by message passing.

Syntax

The syntax of process behaviours is given by the following grammar

$$\begin{aligned}
B &::= L \mid B\ B \mid \mathbf{case}\ B\ \mathbf{of}\ \mathbf{Inl}\ x \Rightarrow B; \mathbf{Inr}\ x \Rightarrow B \mid \oplus_{\mathsf{p}}\ l\ B \\
&\quad\ \mid \&_{\mathsf{p}}\{l_1 : B_1, \ldots, l_n : B_n\} \\
L &::= x \mid \lambda x : T.B \mid \mathbf{Inl}\ L \mid \mathbf{Inr}\ L \mid \mathbf{fst} \mid \mathbf{snd} \mid \mathbf{Pair}\ L\ L \mid () \mid \mathbf{recv}_{\mathsf{p}} \mid \mathbf{send}_{\mathsf{p}} \mid \bot \\
T &::= T \to T \mid T + T \mid T \times T \mid () \mid \bot
\end{aligned}$$

where B is a behaviour, L is a local value, and T is a local type.

The terms from the λ-calculus are standard. Pairs and sums work as described for Chorλ, but note that now they are completely local (as usual) because there are no process name annotations anymore.

The terms for message passing are the local counterparts of choreographic communication terms. Selections are implemented by the *offer* branching term $\&_{\mathsf{p}}\{l_1 : B_1, \ldots, l_n : B_n\}$, which offers a number of different ways it can continue for another process p to choose from,

$$\frac{\Sigma;\Gamma \vdash B : T}{\Sigma;\Gamma \vdash \oplus_{\mathsf{p}}\ l\ B : T}\ [\text{NTC}\textsc{hor}] \qquad \frac{\Sigma;\Gamma \vdash B_i : T \text{ for } 1 \leqslant i \leqslant n}{\Sigma;\Gamma \vdash \&_{\mathsf{p}}\{l_1 : B_1, \dots l_n : B_n\} : T}\ [\text{NTO}\textsc{ff}]$$

$$\frac{}{\Sigma;\Gamma \vdash \mathbf{send}_{\mathsf{p}} : T \to \bot}\ [\text{NTS}\textsc{end}] \qquad \frac{}{\Sigma;\Gamma \vdash \mathbf{recv}_{\mathsf{p}} : \bot \to T}\ [\text{NTR}\textsc{ecv}]$$

$$\frac{}{\Sigma;\Gamma \vdash \bot : \bot}\ [\text{NT}\textsc{botm}] \qquad \frac{\Sigma;\Gamma \vdash B : \bot \quad \Sigma;\Gamma \vdash B' : \bot}{\Sigma;\Gamma \vdash B\ B' : \bot}\ [\text{NTA}\textsc{pp2}]$$

Figure 2 Typing rules for behaviours (selected rules).

and the *choice* term $\oplus_{\mathsf{p}}\ l\ B$, which directs p to continue as the behaviour labelled l. Likewise, value communication is divided into a *send* to p action, $\mathbf{send}_{\mathsf{p}}$, and a *receive* from p action, $\mathbf{recv}_{\mathsf{p}}$.

We also add the no-op term mentioned in the introduction, $\bot$, and its type, $\bot$. A term $\bot$ represents a terminated behaviour with no result. This term is used in the semantics of send and receive: locally, $\mathbf{send}_{\mathsf{p}}$ acts as a function that can take any input and returns $\bot$, and $\mathbf{recv}_{\mathsf{p}}$ a function that given $\bot$ returns some value. More interestingly, $\bot$ also plays an important role wrt modularity in our notion of EPP, which we will discuss later in our presentation of projection. All types but $\bot$ are standard (as in Chorλ, but without process name annotations).

A system of running processes is called a *network*.

▶ **Definition 3.** *A network $\mathcal{N}$ is a finite map from a set of process names to behaviours.*

Given two networks $\mathcal{N}$ and $\mathcal{N}'$ with disjoint domains, their parallel composition $\mathcal{N} \mid \mathcal{N}'$ maps each process name to the behaviour in the network defining the process. Any network is equivalent to a parallel composition of networks with singleton domains, so we write $\mathsf{p}_1[B_1] \mid \dots \mid \mathsf{p}_n[B_n]$ for the network where each process p_i has behaviour B_i [24].

▶ **Example 4.** Consider the choreography $\mathbf{com}_{\mathsf{B},\mathsf{C}}\ (\mathbf{com}_{\mathsf{A},\mathsf{B}}\ ()@\mathsf{A})$. A correct implementation is the network $\mathsf{A}[\mathbf{send}_{\mathsf{B}}\ ()] \mid \mathsf{B}[\mathbf{send}_{\mathsf{C}}\ (\mathbf{recv}_{\mathsf{A}}\ \bot)] \mid \mathsf{C}[\mathbf{recv}_{\mathsf{B}}\ \bot]$. ⌟

Typing

Behaviours are typed with judgements of the form $\Gamma \vdash B : T$. The typing rules are the local counterparts of those in Chorλ, obtained by removing Θ and process names in types. We add the $\bot$ type for terms that can result in $\bot$. Figure 2 displays representative typing rules to deal with $\bot$ and communications.

Semantics

The semantics of networks is given as a labelled transition system. Figure 3 displays some representative transition rules.

Labels for network transitions have the form τ_{P}, where P ranges over sets of one or two process names. Rule NP<span style="font-variant:small-caps">ro</span> annotates an internal transition by a process with its name, and rule NPar lifts transitions in parallel compositions.

The transition axioms for send and receive are typical of process calculi with early semantics. Send and receive transitions are matched in rule NCom to perform a communication The label $\tau_{\mathsf{p},\mathsf{q}}$ denotes an internal move (τ) and manifests the names of processes that contribute to performing it (p and q). We treat the subscript p, q as an unordered set that consists of the two process names.

$$\frac{\mathrm{fv}(L) = \varnothing}{\mathbf{send}_{\mathsf{p}}\ L \xrightarrow{\mathbf{send}_{\mathsf{p}}\ L} \bot}\ [\text{NSend}] \qquad \frac{}{\mathbf{recv}_{\mathsf{p}}\ \bot \xrightarrow{\mathbf{recv}_{\mathsf{p}}\ L} L}\ [\text{NRecv}]$$

$$\frac{B_1 \xrightarrow{\mathbf{send}_{\mathsf{q}}\ L} B_1' \quad B_2 \xrightarrow{\mathbf{recv}_{\mathsf{p}}\ L} B_2'}{\mathsf{p}[B_1] \mid \mathsf{q}[B_2] \xrightarrow{\tau_{\mathsf{p},\mathsf{q}}} \mathsf{p}[B_1'] \mid \mathsf{q}[B_2']}\ [\text{NCom}]$$

$$\frac{}{\oplus_{\mathsf{p}}\ l\ B \xrightarrow{\oplus_{\mathsf{p}}\ l} B}\ [\text{NCho}] \qquad \frac{}{\&_{\mathsf{p}}\{\ell_1 : B_1, \dots, \ell_n : B_n\} \xrightarrow{\&_{\mathsf{p}}\ell_i} B_i}\ [\text{NOff}]$$

$$\frac{B_1 \xrightarrow{\oplus_{\mathsf{q}}\ \ell} B_1' \quad B_2 \xrightarrow{\&_{\mathsf{p}}\ \ell} B_2'}{\mathsf{p}[B_1] \mid \mathsf{q}[B_2] \xrightarrow{\tau_{\mathsf{p},\mathsf{q}}} \mathsf{p}[B_1'] \mid \mathsf{q}[B_2']}\ [\text{NSel}]$$

$$\frac{}{(\lambda x : T.B)\ L \xrightarrow{\tau} B[x := L]}\ [\text{NAbsApp}] \qquad \frac{}{\bot\ \bot \xrightarrow{\tau} \bot}\ [\text{NBotm}]$$

$$\frac{B \xrightarrow{\tau} B'}{\mathsf{p}[B] \xrightarrow{\tau_{\mathsf{p}}} \mathsf{p}[B']}\ [\text{NPro}] \qquad \frac{\mathcal{N} \xrightarrow{\tau_{\mathsf{P}}} \mathcal{N}''}{\mathcal{N} \mid \mathcal{N}' \xrightarrow{\tau_{\mathsf{P}}} \mathcal{N}'' \mid \mathcal{N}'}\ [\text{NPar}]$$

Figure 3 Network semantics (representative rules).

The P-annotations in labels enable the formulation of the next lemma, which we use in some of our proofs to focus on the processes involved in a transition. The proof of this result and others for the full Chorλ language are provided in Appendix A.

▶ **Lemma 5.** *For any* p *and* $\mathcal{N}$*, if* $\mathcal{N} \xrightarrow{\tau_{\mathsf{P}}} \mathcal{N}'$ *and* $\mathsf{p} \notin \mathsf{P}$ *then* $\mathcal{N}(\mathsf{p}) = \mathcal{N}'(\mathsf{p})$*.*

Most of the other rules follow the same intuition and are otherwise standard. The exception is rule NBotm, which garbage collects $\bot$ terms. We discuss the role of this rule in Example 9, after having presented our notion of EPP.

3.2 Endpoint Projection (EPP)

We now move to defining the endpoint projection (EPP) of a choreography M for an individual process p, assuming that M is well-typed; that is, $\Theta; \Gamma \vdash M : T$ for some Θ, Γ, and T. The definition of EPP formally depends on this typing derivation, but to keep notation simple we write just $[\![M]\!]_{\mathsf{p}}$ for the projection of M on p and refer to the type T associated to M in the specific derivation we are looking at as type(M).

Projection translates each choreographic term to a corresponding local behaviour. For example, a communication term $\mathbf{com}_{\mathsf{p},\mathsf{q}}$ is projected to a send action for the sender p and a receive action for the receiver q.

Abstraction presents a novel challenge compared to previous, non-functional choreographic languages. We discuss it in the next example, which also illustrates the importance of $\bot$ in our theory of EPP.

▶ **Example 6.** Let $M = \lambda x : \mathsf{Int}@\mathsf{p}.M'$ for some M', and consider the issue of defining its projection on a process q different than p, $[\![M]\!]_{\mathsf{q}}$. Since EPP is usually defined inductively on the structure of the choreography, this definition should not depend on the context that M is used in.

The standard principle for EPP found in the literature is to ignore the parts that do not mention the process we are currently projecting to. Following this principle, we should omit the initial abstraction (λx) of M in the implementation of q.

For example, for $M = \lambda x : \mathsf{Int@p}.2@\mathsf{q}$, we could design EPP such that $[\![M]\!]_{\mathsf{q}} = 2$. This works when M is used in an application as $(\lambda x : \mathsf{Int@p}.2@\mathsf{q})\ 1@\mathsf{p}$, where $[\![M]\!]_{\mathsf{q}} = 2$ is still reasonable (since q has nothing to do with the argument).

Unfortunately, this standard approach is not robust in the case of functional choreographies: even if q is not mentioned in the type of x in $\lambda x : \mathsf{Int@p}$, in general it could still participate in the context that produces the value that x is going to be replaced with. For example, let $M'' = (\lambda x : \mathsf{Int@p}.\mathbf{com}_{\mathsf{q},\mathsf{p}}\ 2@\mathsf{q})\ (\mathbf{com}_{\mathsf{q},\mathsf{p}}\ 1@\mathsf{q})$, which expresses a sequence of communications between q and p (first of 1 and then of 2, in order). If we insist on excluding the abstraction from the projection on q, then we obtain $[\![M'']\!]_{\mathsf{q}} = (\mathbf{send}_{\mathsf{p}}\ 2)\ (\mathbf{send}_{\mathsf{p}}\ 1)$. This is wrong, because it would send 2 before 1. Therefore, we cannot just skip abstractions that do not involve the process we are projecting on. In this case, a correct implementation of q in M'' would be $(\lambda x : \bot.\mathbf{send}_{\mathsf{p}}\ 2)\ (\mathbf{send}_{\mathsf{p}}\ 1)$. Our process language is carefully designed to make terms like this normalise gracefully: after executing $\mathbf{send}_{\mathsf{p}}\ 1$ the righthandside is $\bot$, thus allowing for the application to be resolved and for the second send action to be executed.

Sometimes, however, abstractions should be skipped. For example, if M is $\lambda x : \mathsf{Int@p}.1@\mathsf{p}$, then $[\![M]\!]_{\mathsf{q}}$ should clearly be $\bot$. The alternative, $\lambda x : \bot.\bot$, would break modularity of EPP because the structure of $[\![M]\!]_{\mathsf{q}}$ would depend on the internal behaviour of p. To solve this issue, we take the approach of skipping an abstraction like $\lambda x : T.M'$ only if both T and M' do not mention the process that we are projecting on. Type information is therefore key to our EPP, in addition to the usual syntactic checks, which is why we have made the EPP dependent on a typing derivation.

We will come back to $\bot$ and its companion rule NBotm in Example 9. ⌟

In order to define EPP precisely, we need a few additional ingredients.

Projecting a term M requires knowing the processes involved in its type. As our EPP takes an entire typing derivation of M as input, the type is implicitly given in the derivation provided to EPP. So we write without ambiguity $\mathrm{pn}(\mathrm{type}(M))$ for this set of process names.

The second ingredient concerns knowledge of choice. When projecting a conditional $\mathbf{case}\ M\ \mathbf{of}\ \mathbf{Inl}\ x \Rightarrow M';\ \mathbf{Inr}\ y \Rightarrow M''$, processes not occurring in M cannot know what branch of the choreography is chosen; therefore, the projections of M' and M'' must be combined in a uniquely-defined behaviour. We thus define a partial *merge* operator ($\sqcup$), adapted from [1, 8, 19], whose key property is

$$\&\{l_i : B_i\}_{i\in I} \sqcup \&\{l_j : B'_j\}_{j\in J} = \&\left(\{l_k : B_k \sqcup B'_k\}_{k\in I\cap J} \cup \{l_i : B_i\}_{i\in I\setminus J} \cup \{l_j : B'_j\}_{j\in J\setminus I}\right)$$

and which is homomorphically defined for the remaining constructs (see Appendix A for the full definition). The idea is that a process not in M must either perform the same actions in M' and M'' (so the choice does not matter) or receive an appropriate selection to know which branch has been chosen. Merging of incompatible behaviours is undefined.

▶ **Example 7.** Consider the choreography

$$C = \mathbf{case}\ \mathbf{Inl}\ ()@\mathsf{p}\ \mathbf{of}\ \mathbf{Inl}\ x \Rightarrow \mathbf{select}_{\mathsf{p},\mathsf{q}}\ \mathsf{left}\ 0@\mathsf{q};\ \mathbf{Inr}\ y \Rightarrow \mathbf{select}_{\mathsf{p},\mathsf{q}}\ \mathsf{right}\ 1@\mathsf{q}\,.$$

Using merging, its projection on process q is $[\![C]\!]_{\mathsf{q}} = \&_{\mathsf{p}}\{\mathsf{left} : 0, \mathsf{right} : 1\}$. ⌟

▶ **Definition 8.** *The EPP of a choreography M on a specific process p ($[\![M]\!]_{\mathsf{p}}$) is defined by the rules in Figure 4. The EPP of a choreography ($[\![M]\!]$) is the parallel composition of the EPPs on its processes: $[\![M]\!] = \prod_{\mathsf{p}\in\mathrm{pn}(M)} \mathsf{p}\left[[\![M]\!]_{\mathsf{p}}\right]$.*

Intuitively, projecting a choreography on a process that is not involved in it returns a $\bot$. In general, however, a choreography may involve processes not mentioned in its type. This explains the first clause for projecting an application: even if p does not appear in the type of

Choreographies

$$[\![M\ N]\!]_{\mathsf{p}} = \begin{cases} [\![M]\!]_{\mathsf{p}}\ \ [\![N]\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(M)) \text{ or } \mathsf{p} \in \mathrm{pn}(M) \cap \mathrm{pn}(N) \\ [\![M]\!]_{\mathsf{p}} & \text{if } [\![N]\!]_{\mathsf{p}} = \bot \\ [\![N]\!]_{\mathsf{p}} & \text{otherwise} \end{cases}$$

$$[\![\lambda x : T.M]\!]_{\mathsf{p}} = \begin{cases} \lambda x : [\![T]\!]_{\mathsf{p}} . [\![M]\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(\lambda x : T.M)) \\ \bot & \text{otherwise} \end{cases}$$

$$\left[\!\!\left[\textbf{case } M \textbf{ of Inl } x \Rightarrow N; \textbf{Inr } x' \Rightarrow N'\right]\!\!\right]_{\mathsf{p}} =$$

$$\begin{cases} \textbf{case } [\![M]\!]_{\mathsf{p}} \textbf{ of Inl } x \Rightarrow [\![N]\!]_{\mathsf{p}}; \textbf{Inr } x' \Rightarrow [\![N']\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(M)) \\ [\![M]\!]_{\mathsf{p}} & \text{if } [\![N]\!]_{\mathsf{p}} = [\![N']\!]_{\mathsf{p}} = \bot \\ [\![N]\!]_{\mathsf{p}} \sqcup [\![N']\!]_{\mathsf{p}} & \text{if } [\![M]\!]_{\mathsf{p}} = \bot \\ (\lambda x'' : \bot . [\![N]\!]_{\mathsf{p}} \sqcup [\![N']\!]_{\mathsf{p}})\ \ [\![M]\!]_{\mathsf{p}} & \text{otherwise, for some} \\ & \quad x'' \notin \mathrm{fv}(N) \cup \mathrm{fv}(N') \end{cases}$$

$$[\![\textbf{Inl } V]\!]_{\mathsf{p}} = \begin{cases} \textbf{Inl } [\![V]\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(\textbf{Inl } V)) \\ \bot & \text{otherwise} \end{cases} \qquad [\![\textbf{fst}]\!]_{\mathsf{p}} = \begin{cases} \textbf{fst} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(\textbf{fst})) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\textbf{select}_{\mathsf{q},\mathsf{q}'}\ l\ M]\!]_{\mathsf{p}} = \begin{cases} \oplus_{\mathsf{q}'}\ l\ \ [\![M]\!]_{\mathsf{p}} & \text{if } \mathsf{p} = \mathsf{q} \neq \mathsf{q}' \\ \&_{\mathsf{q}}\{l : [\![M]\!]_{\mathsf{p}}\} & \text{if } \mathsf{p} = \mathsf{q}' \neq \mathsf{q} \\ [\![M]\!]_{\mathsf{p}} & \text{otherwise} \end{cases}$$

$$[\![\textbf{com}_{\mathsf{q},\mathsf{q}'}]\!]_{\mathsf{p}} = \begin{cases} \lambda x : [\![T]\!]_{\mathsf{p}} . x & \text{if } \mathsf{p} = \mathsf{q} = \mathsf{q}' \text{ and } \mathrm{type}(\textbf{com}_{\mathsf{q},\mathsf{q}'}) = T \rightarrow_{\varnothing} T' \\ \textbf{send}_{\mathsf{q}'} & \text{if } \mathsf{p} = \mathsf{q} \neq \mathsf{q}' \\ \textbf{recv}_{\mathsf{q}} & \text{if } \mathsf{p} = \mathsf{q}' \neq \mathsf{q} \\ \bot & \text{otherwise} \end{cases}$$

$$[\![()@\mathsf{q}]\!]_{\mathsf{p}} = \begin{cases} () & \text{if } \mathsf{q} = \mathsf{p} \\ \bot & \text{otherwise} \end{cases} \qquad [\![x]\!]_{\mathsf{p}} = \begin{cases} x & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(x)) \\ \bot & \text{otherwise} \end{cases}$$

Types

$$[\![()@\mathsf{q}]\!]_{\mathsf{p}} = \begin{cases} () & \text{if } \mathsf{q} = \mathsf{p} \\ \bot & \text{otherwise} \end{cases} \qquad [\![T \times T']\!]_{\mathsf{p}} = \begin{cases} [\![T]\!]_{\mathsf{p}} \times [\![T']\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(T \times T') \\ \bot & \text{otherwise} \end{cases}$$

$$[\![T \rightarrow_{\rho} T']\!]_{\mathsf{p}} = \begin{cases} [\![T]\!]_{\mathsf{p}} \rightarrow [\![T']\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \rho \cup \mathrm{pn}(T) \cup \mathrm{pn}(T') \\ \bot & \text{otherwise} \end{cases}$$

Figure 4 Projecting a choreography in Chorλ onto a process – when cases overlap, the first one takes precedence (representative rules).

M, it may participate in interactions in M. Vice versa, a process can appear in the type of a choreography without appearing in the choreography itself. The difference between a process appearing in a choreography or its type becomes important when we look at the projection of **case** M **of Inl** $x \Rightarrow N$**; Inr** $x' \Rightarrow N'$. Here, p appearing in the type of M indicates that p will, at the end of the computation of M, know what branch will be chosen; therefore, the projection on p is a **case**. However, it is possible that p is involved in the computation of the condition M without knowing the final choice, e.g., if $M = \mathbf{com}_{\mathsf{p},\mathsf{q}}\ M'$. In this case, the projection on p is not a **case** but still needs code to participate in the implementation of M correctly. If p is involved in the branches as well, then we need to project code for them too: we inject an abstraction in order to maintain the correct order of computation (M before N and N') and make the resulting process well typed (since p does not appear in the type of M, that type will be projected to $\bot$).

The projection of abstraction illustrates the necessity of the ρ annotation on abstraction types. For example, consider an application of a communication via a proxy $(\lambda x : \mathsf{Int@p} \rightarrow_{\{\mathsf{r}\}} \mathsf{Int@q}.x\ 3@\mathsf{p})\ (\lambda y : \mathsf{Int@p}.\mathbf{com}_{\mathsf{r},\mathsf{q}}\ \mathbf{com}_{\mathsf{p},\mathsf{r}}\ y)$. Without the annotation $\{\mathsf{r}\}$ in subterm $(\lambda x : \mathsf{Int@p} \rightarrow_{\{\mathsf{r}\}} \mathsf{Int@q}.x\ 3@\mathsf{p})$, the projection of this subterm on r would just be $\bot$, which is wrong for the overall application since r will actually be involved.

Selections and communications follow the intuition given before, with one interesting detail: self-selections are ignored, and self-communications are projected to the identity function. This is different from previous works, where self-communication is not allowed – here we lift this restriction.

Likewise, projecting a type T yields $\bot$ at any process not used in T.

▶ **Example 9.** Let $M = (\mathbf{com}_{\mathsf{p},\mathsf{q}}\ (\lambda x : \mathsf{Int@p}.3@\mathsf{p}))\ (\mathbf{com}_{\mathsf{p},\mathsf{q}}\ 5@\mathsf{p})$, where a function and a value are both sent from p to q before being applied at q. The implementation of q is $[\![M]\!]_{\mathsf{q}} = (\mathbf{recv}_{\mathsf{p}}\ \bot)\ (\mathbf{recv}_{\mathsf{p}}\ \bot)$, whose execution is straightforward. At p, however, we have that $[\![M]\!]_{\mathsf{p}} = (\mathbf{send}_{\mathsf{q}}(\lambda x : \mathsf{Int}.3))\ (\mathbf{send}_{\mathsf{q}}\ 5)$, which after executing the two send actions becomes $\bot\ \bot$. After executing its two communications, the choreography M becomes $M' = (\lambda x : \mathsf{Int@q}.3@\mathsf{q})\ 5@\mathsf{q}$. M' is located entirely at q, and therefore $[\![M']\!]_{\mathsf{p}} = \bot$, which is different than the $\bot\ \bot$ reached by $[\![M]\!]_{\mathsf{p}}$. We therefore need a way to make the application $\bot\ \bot$ become $\bot$. Rule NBotm serves this purpose. The fact that this is not possible with two units is the key semantic difference between $\bot$ and $()$. ⌟

▶ **Proposition 10.** *Let M be a closed choreography. If $\Theta; \Gamma \vdash M : T$, then for any process p appearing in M, we have that $[\![\Gamma]\!] \vdash [\![M]\!]_{\mathsf{p}} : [\![T]\!]_{\mathsf{p}}$, where $[\![\Gamma]\!]$ are defined by applying EPP to all types occurring Γ.*

▶ **Example 11.** Let M be the remote function choreography in Example 1. Its projections on C and S are as follows.

$$[\![M]\!]_{\mathsf{C}} = \lambda f : \bot.\ \lambda val : \mathsf{Int}.\ \mathbf{recv}_{\mathsf{S}}\ (\mathbf{send}_{\mathsf{S}}\ val)$$
$$[\![M]\!]_{\mathsf{S}} = \lambda f : (\mathsf{Int} \rightarrow \mathsf{Int}).\ \lambda val : \bot.\ \mathbf{send}_{\mathsf{C}}\ (f\ (\mathbf{recv}_{\mathsf{C}}\ \bot))$$

This example illustrates the key features discussed in the text: projection of communications as two dual actions; and the way function applications are projected when the process does not appear in the function's type. ⌟

We describe what we consider modularity of EPP, formally defined in Definition 12. Modular projection means that for any context $C[]$ the projection of $C[M]$ at p will be the same for any M which does not involve p. The definition of context is as expected and can be found in Appendix A. Modularity is typical (and expected) of EPP, because the projection of p should not be generating junk code based on the behaviour of other processes.

▶ **Definition 12** (Modularity of EPP). *An EPP* $[\![-]\!]$ *is called* modular *if* $[\![C[M]]\!]_{\mathsf{p}} = [\![C[N]]\!]_{\mathsf{p}}$ *for any process* p*, context* $C[]$*, and choreographies* M *and* N *such that* $\Theta;\Gamma \vdash M : T$ *and* $\Theta;\Gamma \vdash N : T$ *with* $\mathsf{p} \notin \Theta$*.*

Modularity ensures that if we modify part of a choreography in which a process p is not involved, we do not need to recompile the projection of the choreography onto p because this projection is unaffected. In general, the strong equality requirement could be relaxed to allow for some extra local actions that do not change the observable behaviour of a process, e.g., adding "empty" applications like $\lambda x.\bot : \bot$. This would yield some extra flexibility to deal with cases such as the one seen in Example 6, so long as the interactions with other processes and return value at p do not change. However, this design would come at some costs: an increase in complexity due to the addition of a suitable notion of behavioural equivalence; a potential loss in efficiency, since processes might gain unnecessary reductions in their projections; and a potential leak of information, since the local code projected on a process would reveals some information about the behaviours of other processes.

The following proposition, Proposition 14, shows that our EPP is modular.

▶ **Lemma 13.** *Given a choreography* M*, if* $\Theta;\Gamma \vdash M : T$ *and* $\mathsf{p} \notin \Theta$ *then* $[\![M]\!]_{\mathsf{p}} = \bot$*.*

Proof. Follows from $\mathsf{p} \notin \Theta$ implying $\mathsf{p} \notin \mathrm{pn}(T) \cup \mathrm{pn}(M)$ and induction on the derivation of $[\![M]\!]_{\mathsf{p}}$. ◀

▶ **Proposition 14.** *The EPP* $[\![-]\!]$ *given in Definition 8 is modular.*

Proof. Follows from Lemma 13 and observing that the projection of any context always treats $\bot$ the same. ◀

▶ **Example 15** (Distributed choice types). Now that we can project a choreography, we return to the idea of distributed choice types from the introduction. Consider a choreography

$$M = \lambda x : \mathsf{Bool@(p,q)}.\mathbf{case}\ x\ \mathbf{of\ Inl}\ y \Rightarrow \mathbf{com}_{\mathsf{p},\mathsf{q}} 3@\mathsf{p};\ \mathbf{Inr}\ y \Rightarrow 5@\mathsf{q}$$

Here $\mathsf{Bool@(p,q)}$ is equivalent to the type $(()@\mathsf{p} \times ()@\mathsf{q}) + (()@\mathsf{p} \times ()@\mathsf{q})$, and in general we can encode a "distributed boolean" as

$$\mathsf{Bool}@\vec{\mathsf{p}} = (()@\mathsf{p}_1 \times \cdots \times ()@\mathsf{p}_n) + (()@\mathsf{p}_1 \times \cdots \times ()@\mathsf{p}_n)$$

We can use distributed booleans to codify distributed choices, in this case by having both p and q be able to make local choice without interacting but still guaranteeing that they choose their respective behaviours correctly.

Specifically, when we project M we get two local choices made at p and q, both of which are guaranteed to make the same choice. First we have the projections

$$[\![M]\!]_{\mathsf{p}} = \lambda x : (() \times \bot) + (() \times \bot).\mathbf{case}\ x\ \mathbf{of\ Inl}\ y \Rightarrow \mathbf{send}_{\mathsf{q}}\ 3;\ \mathbf{Inr}\ y \Rightarrow \bot$$

and

$$[\![M]\!]_{\mathsf{q}} = \lambda x : (\bot \times ()) + (\bot \times ()).\mathbf{case}\ x\ \mathbf{of\ Inl}\ y \Rightarrow \mathbf{recv}_{\mathsf{p}}\ \bot;\ \mathbf{Inr}\ y \Rightarrow 5$$

For these processes to be deadlock-free when put in parallel, we need both of them to make the same choice. Thankfully, the distributed boolean type ensures that x will always be instantiated as either $\mathbf{Inl}\ (\mathbf{Pair}\ ()@\mathsf{p}\ ()@\mathsf{q})$ or $\mathbf{Inr}\ (\mathbf{Pair}\ ()@\mathsf{p}\ ()@\mathsf{q})$. From the projection we get $[\![\mathbf{Inl}\ (\mathbf{Pair}\ ()@\mathsf{p}\ ()@\mathsf{q})]\!]_{\mathsf{p}} = \mathbf{Inl}\ (\mathbf{Pair}\ ()\ \bot)$ and $[\![\mathbf{Inl}\ (\mathbf{Pair}\ ()@\mathsf{p}\ ()@\mathsf{q})]\!]_{\mathsf{q}} = \mathbf{Inl}\ (\mathbf{Pair}\ \bot\ ())$, and similar for the **Inr** case. We therefore know that Chorλ's distributed choice works as intended when projected. As we shall see in Section 5, one use for this technique is to have different processes independently agree on the size of a distributed list.

Note that if we tried to model a distributed boolean as $(()@\mathsf{p} + ()@\mathsf{p}) \times (()@\mathsf{q} + ()@\mathsf{q})$, it would not be useful to represent a distributed choice because it would allow the processes to make different choices. (Also, M would obviously not be well-typed, as a condition must have a sum type.) ⌟

We now show that there is a close correspondence between the executions of choreographies and of their projections. Intuitively, this correspondence states that a choreography can execute an action if, and only if, its projection can execute the same action, and both transition to new terms in the same relation. Technically, we need to be more precise: if a choreography M reduces by rule Case, then the result has fewer branches than the network obtained by performing the corresponding reduction in the projection of M. (This is a standard issue with choreographic conditionals [24].)

In order to capture this, we define a partial order $\sqsupseteq$ that relates a behaviour to a version with fewer branches: $B \sqsupseteq B'$ iff $B \sqcup B' = B$. Intuitively, if $B \sqsupset B'$, then B offers the same or more branches than B' (also in subterms). This notion extends to networks by defining $\mathcal{N} \sqsupseteq \mathcal{N}'$ to mean that, for any process p, $\mathcal{N}(\mathsf{p}) \sqsupseteq \mathcal{N}'(\mathsf{p})$. Example 16 shows the necessity of $\sqsupseteq$ in order to get a meaningful notion of operational correspondence between choreographies and their projection.

▶ **Example 16.** Consider again the choreography from Example 7,

$$C = \textbf{case Inl } ()@\mathsf{p} \textbf{ of Inl } x \Rightarrow \textbf{select}_{\mathsf{p},\mathsf{q}} \ \mathsf{left}\ 0@\mathsf{q}; \textbf{ Inr } y \Rightarrow \textbf{select}_{\mathsf{p},\mathsf{q}} \ \mathsf{right}\ 1@\mathsf{q}\,,$$

and its projection B on q, $B = [\![C]\!]_{\mathsf{q}} = \&_{\mathsf{p}}\{\mathsf{left} : 0, \mathsf{right} : 1\}$.

When entering the **case**, C reduces to $C' = \textbf{select}_{\mathsf{p},\mathsf{q}}\ \mathsf{left}\ 0@\mathsf{q}$, but q is not involved in this action and its behaviour remains B, which is not $[\![C']\!]_{\mathsf{q}}$. However, $\&_{\mathsf{p}}\{\mathsf{left} : 0, \mathsf{right} : 1\} \sqcup \&_{\mathsf{p}}\{\mathsf{left} : 0\} = \&_{\mathsf{p}}\{\mathsf{left} : 0, \mathsf{right} : 1\}$, so $B \sqsupseteq [\![C']\!]_{\mathsf{q}}$. ⌟

In addition to $\sqsupseteq$, we need to equate behaviours that differ only by applications to $\perp$ like P and $(\lambda x : \perp.P)\ \perp$ introduced by the projection of applications.

▶ **Definition 17.** *We define $\equiv$ as the least equivalence relation on behaviours that is closed under context and $P \equiv (\lambda x : \perp.P)\ \perp$ for any behaviour P. We write $\mathcal{N} \equiv \mathcal{N}'$ for the pointwise extension of $\equiv$ to networks (i.e., $\Pi_{\mathsf{p}}\mathsf{p}[P_{\mathsf{p}}] \equiv \Pi_{\mathsf{p}}\mathsf{p}[P'_{\mathsf{p}}]$ iff $P_{\mathsf{p}} \equiv P'_{\mathsf{p}}$ for all* ps*) and $\mathcal{N} \sqsupseteqq \mathcal{N}'$ if there is a network $\mathcal{N}''$ such that $\mathcal{N} \sqsupseteq \mathcal{N}''$ and $\mathcal{N}'' \equiv \mathcal{N}'$.*

We can finally show that the EPP of a choreography can do all that (completeness) and only what (soundness) the choreography does. Here $\to^*$ denotes a sequence of transitions with any labels, and $\to^+$ a nonempty such sequence.

▶ **Theorem 18** (Completeness). *Given a closed choreography M, if $M \xrightarrow{\mathsf{P}} M'$, $\Theta; \Gamma \vdash M : T$, and $[\![M]\!]$ is defined, then there exist networks $\mathcal{N}$ and M'' such that: $[\![M]\!] \to^+ \mathcal{N}$; $M' \to^* M''$; and $\mathcal{N} \sqsupseteqq [\![M'']\!]$.*

▶ **Theorem 19** (Soundness). *Given a closed choreography M, if $\Theta; \Gamma \vdash M : T$ and $[\![M]\!] \to^* \mathcal{N}$ for some network $\mathcal{N}$, then there exist a choreography M', and a network $\mathcal{N}'$ such that: $M \to^* M'$; $\mathcal{N} \to^* \mathcal{N}'$; and $\mathcal{N}' \sqsupseteqq [\![M']\!]$.*

Since we have no recursion and only require that the choreography and projection eventually get to the same state, we can prove soundness and correctness without needing the out-of-order semantics usually required in choreographic languages [24].

From Theorems 18 and 19 and the type preservation and progress results from [6], we obtain deadlock-freedom: the EPP of a well-typed closed choreography can continue to reduce until all processes contain only local values.

▶ **Corollary 20** (Deadlock-freedom). *Given a closed choreography M, if $\Theta;\Gamma \vdash M : T$ then: whenever $[\![M]\!] \to^* \mathcal{N}$ for some network $\mathcal{N}$, either there exists* p *and $\mathcal{N}'$ such that $\mathcal{N} \xrightarrow{\tau_{\mathsf{p}}} \mathcal{N}'$ or $\mathcal{N} = \prod_{\mathsf{p}\in \mathrm{pn}(M)} \mathsf{p}[L_{\mathsf{p}}]$.*

4 Recursion

So far we have worked with a recursion-free subset of Chorλ. In this section, we extend our development to the full language presented of Chorλ, which includes recursive definitions [6]. As we will see, recursion is technically challenging because of the introduction of divergence.

4.1 Definitions

Choreographies

Recursion in Chorλ is achieved by named functions (f) parametrised on process names. We use D to range over mappings of parametrised functions names to choreographies (the bodies of the functions). To execute a choreography M containing calls to named functions, the choreography must be associated with a mapping D that contains all the named functions called by M. The grammar of choreographies is extended with $M ::= \cdots \mid f(\vec{\mathsf{p}})$. A function call $f(\vec{\mathsf{p}})$ invokes f by instantiating its parameters with the process names $\vec{\mathsf{p}}$, which evaluates to the body of the function. In a function call or definition, parameters must be distinct. Semantically, we add D as an annotation to the reduction relation for choreographies and use the following rule to evaluate functions. Labels in Chorλ with recursion are extended to the form ℓ, P, where the new ingredient ℓ can be either τ or λ. The need for ℓ is explained later.

$$\frac{D(f(\vec{\mathsf{p}'})) = M}{f(\vec{\mathsf{p}}) \xrightarrow{\tau,\varnothing}_D M[\vec{\mathsf{p}'} \mapsto \vec{\mathsf{p}}]}\ [\textsc{Def}]$$

To type recursive choreographies, we introduce recursive type variables ranged over by t. These are defined in a collection Σ, which contains type equations of the form $t@\vec{\mathsf{p}} = T$ – the elements of $\vec{\mathsf{p}}$ must be distinct. The grammar of types is extended with parametrised variables: $T ::= \cdots \mid t@\vec{\mathsf{p}}$. Essentially, assuming the presence of an equation $t@\vec{\mathsf{p}'} = T$, $t@\vec{\mathsf{p}}$ can be unfolded into $T[\vec{\mathsf{p}'} := \vec{\mathsf{p}}]$. Typing judgements are then of the form $\Theta;\Sigma;\Gamma \vdash M : T$, where Γ may now also contain type assignments for recursive functions of the form $f(\vec{\mathsf{p}}) : T$.

$$\frac{\Theta;\Sigma;\Gamma \vdash M : t@\vec{\mathsf{p}'} \quad t@\vec{\mathsf{p}} =_\Sigma T \quad \vec{\mathsf{p}'} \subseteq \Theta \quad ||\vec{\mathsf{p}}|| = ||\vec{\mathsf{p}'}|| \quad \vec{\mathsf{p}'} \text{ distinct}}{\Theta;\Sigma;\Gamma \vdash M : T[\vec{\mathsf{p}} := \vec{\mathsf{p}'}]}\ [\textsc{TEq}]$$

We also write $\Theta;\Sigma;\Gamma \vdash D$ to denote that each function in D can be typed accordingly to its type in Γ.

▶ **Example 21** (Remote Map). With recursive functions, we can write more complex choreographies that call themselves and each other. Let $\mathsf{remoteFunction}(\mathsf{C},\mathsf{S})$ be defined as the choreography in Example 1. We use it to define a function $\mathsf{remoteMap}(\mathsf{C},\mathsf{S})$, where a server S applies a function to not just one value, but instead to each element of a stream communicated from a client C. Then S returns the results, which C gathers into a list with the standard cons function used to construct a new list.

```
remoteMap(C, S) = $\lambda fun$ : Int@S → Int@S. $\lambda list$ : [Int]@C.
  case $list$ of
    Inl $x$ ⇒ select$_{C,S}$ stop ()@C;
    Inr $x$ ⇒ select$_{C,S}$ again
      cons(C) (remoteFunction(C, S) $fun$ (fst $x$)) (remoteMap(C, S) $fun$ (snd $x$))
```

Here, $[\mathsf{Int}]@\mathsf{C}$ is defined as $[\mathsf{Int}]@\mathsf{C} = ()@\mathsf{C} + (\mathsf{Int}@\mathsf{C} \times [\mathsf{Int}]@\mathsf{C})$, representing a list of integers. In general, we write $[t]@(\mathsf{p}_1, \ldots, \mathsf{p}_n)$ to mean the type satisfying $[t]@(\mathsf{p}_1, \ldots, \mathsf{p}_n) = (()@\mathsf{p}_1 \times \cdots \times ()@\mathsf{p}_n) + (t@(\mathsf{p}_1, \ldots, \mathsf{p}_n) \times [t]@(\mathsf{p}_1, \ldots, \mathsf{p}_n))$. ⌟

▶ **Example 22** (Diffie-Hellman [6]). We recall the choreography for the Diffie–Hellman key exchange protocol [13], which allows two processes to agree on a shared secret key without assuming secrecy of communications. Again, we use the primitive type Int.

To define this protocol, we use the local function modPow(R) of the type

$$\mathsf{modPow}(\mathsf{R}) : \mathsf{Int}@\mathsf{R} \to \mathsf{Int}@\mathsf{R} \to \mathsf{Int}@\mathsf{R} \to \mathsf{Int}@\mathsf{R}$$

which computes powers with a given modulo. Given modPow(R), we can implement Diffie–Hellman as the following choreography:

```
diffieHellman(P, Q) =
    $\lambda psk$ : Int@P. $\lambda qsk$ : Int@Q. $\lambda psg$ : Int@P.
    $\lambda qsg$ : Int@Q. $\lambda psp$ : Int@P. $\lambda qsp$ : Int@Q.
        pair (modPow(P) $psg$ (com$_{Q,P}$ (modPow(Q) $qsg$ $qsk$ $qsp$)) $psp$)
              (modPow(Q) $qsg$ (com$_{P,Q}$ (modPow(P) $psg$ $psk$ $psp$)) $qsp$)
```

Given the individual secret keys (psk and qsk) and a previously publicly agreed upon shared prime modulus and base ($psg = qsg, psp = qsp$), the participants exchange their locally-computed public keys in order to arrive at a shared key that can be used to encrypt all further communication. This means diffieHellman(P, Q) has the type:

$$\mathsf{Int}@\mathsf{P} \to \mathsf{Int}@\mathsf{Q} \to \mathsf{Int}@\mathsf{P} \to \mathsf{Int}@\mathsf{Q} \to \mathsf{Int}@\mathsf{P} \to \mathsf{Int}@\mathsf{Q} \to \mathsf{Int}@\mathsf{P} \times \mathsf{Int}@\mathsf{Q}$$

and represents the shared key as a pair of equal keys, one for each participant.

The choreography then takes a shared key as its parameter and produces a pair of unidirectional channels that wrap the communication primitive with the necessary encryption based on the key:

```
makeSecureChannels(P, Q) = $\lambda key$ : Int@P × Int@Q.
    Pair ($\lambda val$ : String@P. (dec(Q) (snd $key$) (com$_{P,Q}$ (enc(P) (fst $key$) $val$))))
          ($\lambda val$ : String@Q. (dec(P) (fst $key$) (com$_{Q,P}$ (enc(Q) (snd $key$) $val$))))
```

Here enc and dec are local function for encoding and decoding values based on keys.

The fact that this choreography returns a pair of channels can also be seen from its type:

$$(\mathsf{Int}@\mathsf{P} \times \mathsf{Int}@\mathsf{Q}) \to ((\mathsf{String}@\mathsf{P} \to \mathsf{String}@\mathsf{Q}) \times (\mathsf{String}@\mathsf{Q} \to \mathsf{String}@\mathsf{P}))$$

Using the channels is as easy as using **com** itself and amounts to a function application.

Process language

To implement recursive functions in Chorλ, we also add recursive functions to our process language: $B ::= \cdots \mid f(\vec{\mathsf{p}})$. They have the same syntax as in choreographies, being parametric on the names of any other processes our process may interact with as part of the function. Local function names are associated with their definition by a function $\mathbb{D}$, which works the same as D in the choreographic setting. Furthermore, we add a transition rule to the process language similar to rule Def for choreographies.

Endpoint Projection

We respectively project function calls, type variables, and function definitions as follows.

$$[\![f(\vec{\mathsf{p}})]\!]_{\mathsf{p}} = \begin{cases} f_i(\mathsf{p}_1,\dots,\mathsf{p}_{i-1},\mathsf{p}_{i+1},\dots,\mathsf{p}_n) & \text{if } \vec{\mathsf{p}} = \mathsf{p}_1,\dots,\mathsf{p}_{i-1},\mathsf{p},\mathsf{p}_{i+1},\dots,\mathsf{p}_n \\ \perp & \text{otherwise} \end{cases}$$

$$[\![t@\vec{\mathsf{p}}]\!]_{\mathsf{p}} = \begin{cases} t_i & \text{if } \vec{\mathsf{p}} = \mathsf{p}_1,\dots,\mathsf{p}_{i-1},\mathsf{p},\mathsf{p}_{i+1},\dots,\mathsf{p}_n \\ \perp & \text{otherwise} \end{cases}$$

$$[\![D]\!] = \{f_i(\mathsf{p}_1,\dots,\mathsf{p}_{i-1},\mathsf{p}_{i+1},\dots,\mathsf{p}_n) \mapsto [\![M]\!]_{\mathsf{p}_i} \mid D(f(\mathsf{p}_1,\dots,\mathsf{p}_n)) = M\}$$

Each named function gets projected to a different named function for each process in its list of parameters, with the projected environment now treating each of these as separate functions parametric on the remaining involved processes. These parameters are needed to implement interactions. Each process can enter a named function independently. Thus, for example, if $D(f(\mathsf{p},\mathsf{q})) = M$ we get $[\![D]\!]\,(f_1(\mathsf{q})) = [\![M]\!]_{\mathsf{p}}$ and $[\![D]\!]\,(f_2(\mathsf{p})) = [\![M]\!]_{\mathsf{q}}$.

On the other hand, projection of recursive types does not need to consider other processes than the one we are projecting on, since local types never mention any processes. Σ is otherwise projected similarly to D. For example, if $t@(\mathsf{p},\mathsf{q}) = T \in \Sigma$ then $t_1 = [\![T]\!]_{\mathsf{p}} \in [\![\Sigma]\!]$ and $t_2 = [\![T]\!]_{\mathsf{q}} \in [\![\Sigma]\!]$.

$$[\![\Sigma]\!] = \{t_i = [\![T]\!]_{\mathsf{p}_i} \mid t@(\mathsf{p}_1,\dots,\mathsf{p}_n) = T \in \Sigma\}$$

▶ **Example 23** (Projecting Example 21). Projecting the choreography in Example 21 yields the processes $\mathsf{remoteMap}_1$ (for the client) and $\mathsf{remoteMap}_2$ (for the server) below. The bodies of $\mathsf{remoteFunction}_1$ and $\mathsf{remoteFunction}_2$ are the terms in Example 11.

```
remoteMap₁(S) = λfun : ⊥. λlist : [Int].
  case list of
    Inl x ⇒ ⊕S stop ();
    Inr x ⇒ ⊕S again
      cons₁ (remoteFunction₁(S) ⊥ (fst x)) (remoteMap₁(S) ⊥ (snd x))
remoteMap₂(C) = λfun : Int → Int. λlist : ⊥.
  &C{stop : ⊥, again : (remoteFunction₂(C) fun ⊥) (remoteMap₂(C) fun ⊥)}
```

⌟

▶ **Example 24** (Projecting Example 22). Projecting our choreographies $\mathsf{diffieHellman}(\mathsf{P},\mathsf{Q})$ and $\mathsf{makeSecureChannels}(\mathsf{P},\mathsf{Q})$ for process P yields the following behaviours.

```
[[D(diffieHellman(P, Q))]]₁ (Q) = λpsk : Int. λqsk : ⊥. λpsg : Int. λqsg : ⊥. λpsp : Int. λqsp : ⊥.
  pair (modPow₁ psg (recvQ ⊥)) psp)
      (sendQ (modPow₁ psg psk psp))
```

$$\begin{aligned}&[\![D(\mathsf{makeSecureChannels}(\mathsf{P},\mathsf{Q}))]\!]_1\,(\mathsf{Q}) = \lambda key : \mathsf{Int} \times \bot.\\ &\quad \mathbf{Pair}\ (\lambda val : \mathsf{String}.\ ((\mathbf{snd}\ key)\ (\mathbf{send}_{\mathsf{Q}}\ (\mathsf{encrypt}_1\ (\mathbf{fst}\ key)\ val)))\\ &\qquad (\lambda val : \bot.\ (\mathsf{decrypt}_1\ (\mathbf{fst}\ key)\ (\mathbf{recv}_{\mathsf{Q}}\ (\mathbf{snd}\ key)))\end{aligned}$$

Note the way function calls such as $\mathsf{modPow}(\mathsf{P})$ in the choreography get projected to modPow_1 on P, since they are treated as degenerate choreographies (they have only one process) and P is the first and only process involved. Conversely, $\mathsf{modPow}(\mathsf{Q})$ on P gets projected as $\bot$ since it is located entirely at a different process.

4.2 Out-of-order execution

In the presence of recursion, getting a correspondence between a process and choreographic language becomes much more challenging. In our results for Chorλ without recursion, we relied on the fact that a choreography would eventually reduce to a value. This is no longer true as choreographies can now diverge, and worse they can diverge at one process without diverging at another. Let, for example, $M = (\lambda x : \mathsf{Int@p}.\mathbf{fst}\ (\mathbf{Pair}\ 5@\mathsf{q}\ x))\ f(\mathsf{p})$. Assume that $D(f(\mathsf{p}_1)) = M'$, where M' diverges. Then the reduction rules that we have seen so far would not allow x to be instantiated. However, $[\![f(\mathsf{p})]\!]_{\mathsf{q}} = \bot$, so $[\![M]\!]_{\mathsf{q}}$ can reduce to 5. Therefore, we need a way to let M copy the reduction of $\mathbf{fst}\ (\mathbf{Pair}\ 5@\mathsf{q}\ x)$ to $5@\mathsf{q}$. In [6], we included corresponding reduction rules for Chorλ to deal with this kind of issues. These rules are all type preserving and avoid creating situations where processes disagree on which communication should be performed first [6]. These rules were unnecessary to deal with the recursion-free fragment, so we introduce them now.

Rule InAbs below addresses situations as in the previous example.

$$\frac{M \xrightarrow{\ell,\mathsf{P}}_D M'}{\lambda x : T.M \xrightarrow{\lambda,\mathsf{P}}_D \lambda x : T.M'}\ [\textsc{InAbs}] \qquad \frac{M \xrightarrow{\ell,\mathbf{R}}_D M' \quad \ell = \lambda \Rightarrow \mathsf{P} \cap \mathrm{pn}(N) = \varnothing}{M\ N \xrightarrow{\tau,\mathsf{P}}_D M'\ N}\ [\textsc{App1}]$$

Rule App1 use the ℓ-component in reduction labels to identify whether a reduction is performed under an abstraction ($\ell = \lambda$) or not ($\ell = \tau$). We need this distinction to prevent interactions under an abstraction performed by processes involved in the righthandside of an application. This restriction serves to avoid breaking causal dependencies between communications. Consider the choreography $(\lambda x : \mathsf{Int@p}.\mathbf{com}_{\mathsf{q},\mathsf{p}}\ 4@\mathsf{q})\ (\mathbf{com}_{\mathsf{q},\mathsf{p}}\ 5@\mathsf{q})$, where the righthandside communication should be performed first – without the restriction, this would not be guaranteed. Reductions under abstractions additionally necessitates a new safety condition on rule AppAbs, ensuring that the free variables of V are distinct from the bound variables of M to avoid problems with scope.

Our modification allows the choreography $M = (\lambda x : \mathsf{Int@q}.\mathbf{fst}\ (\mathbf{Pair}\ 5@\mathsf{q}\ x))\ f(\mathsf{p},\mathsf{q})$ to reduce to $M' = (\lambda x : \mathsf{Int@p}_2.5@\mathsf{q})\ f(\mathsf{p},\mathsf{q})$. Thus, the projections of M on p and q must be able to reduce to the projections of M'. For p this is easy, since $[\![M]\!]_{\mathsf{p}} = [\![M']\!]_{\mathsf{p}} = \bot\ f_1(\mathsf{q})$. For q, however, we need $[\![M]\!]_{\mathsf{q}} = (\lambda x : \mathsf{Int}.\mathbf{fst}\ (\mathbf{Pair}\ \bot\ x))\ f_2(\mathsf{p})$ to reduce to $[\![M']\!]_{\mathsf{q}} = (\lambda x : \mathsf{Int}.\bot)\ f_2(\mathsf{p})$, which requires the process language to have similar out-of-order semantics. We therefore add an equivalent rule NInAbs and modify rule NApp1 similarly to rule App1.

In the network, rather than checking for interacting processes, we do not allow communication actions ($\mathbf{send}$, $\mathbf{recv}$, $\oplus$, $\&$) from inside an abstraction. The reduction labels for the process language are thus simpler (τ or λ), since we do not need to track process names involved in actions.

Similar problems appear with applications that have divergent subterms on the lefthand-side, like $f(\mathsf{q})\ ((\lambda x : \mathsf{Int@p}.4@\mathsf{q})\ 3@\mathsf{p})$, and are treated similarly (the corresponding reduction rules are given in the appendix).

$$\frac{x \notin \mathrm{fv}(M')}{((\lambda x : T.M)\ N)\ M' \rightsquigarrow (\lambda x : T.(M\ M'))\ N}\ [\text{R-AbsR}]$$

$$\frac{x, x' \notin \mathrm{fv}(M) \quad \mathrm{spn}(M) \cap \mathrm{pn}(N) = \varnothing}{\begin{array}{l} M\ (\mathbf{case}\ N\ \mathbf{of\ Inl}\ x \Rightarrow M_1; \mathbf{Inr}\ x' \Rightarrow M_2) \rightsquigarrow \\ \qquad \mathbf{case}\ N\ \mathbf{of\ Inl}\ x \Rightarrow (M\ M_1); \mathbf{Inr}\ x' \Rightarrow (M\ M_2)\end{array}}\ [\text{R-CaseL}]$$

$$\frac{\mathrm{spn}(M) \cap \mathrm{pn}(N) = \varnothing}{M\ (\mathbf{select}_{\mathsf{q},\mathsf{p}}\ l\ N) \rightsquigarrow \mathbf{select}_{\mathsf{q},\mathsf{p}}\ l\ (M\ N)}\ [\text{R-SelL}]$$

$$\frac{y \text{ fresh for } M}{\lambda x : T.M \rightsquigarrow \lambda y : T.M[x := y]}\ [\text{R-alph}]$$

Figure 5 Rewriting of Chorλ (representative rules).

$$\frac{\mathrm{pn}(B) = \varnothing}{B\ (\&_{\mathsf{p}}\{l_1 : B_1, \dots, l_n : B_n\}) \rightsquigarrow \&_{\mathsf{p}}\{l_1 : B\ B_1, \dots, l_n : B\ B_n\}}\ [\text{LR-OffL}]$$

$$\frac{\mathrm{pn}(B') = \varnothing}{B'\ (\oplus_{\mathsf{p}}\ l\ B) \rightsquigarrow \oplus_{\mathsf{p}}\ l\ (B'\ B)}\ [\text{LR-ChoL}] \qquad \frac{}{\bot\ \bot \rightsquigarrow \bot}\ [\text{LR-Botm}]$$

Figure 6 Rewriting of behaviours (representative rules).

Dealing with recursive functions in nested applications requires another addition to the semantics of Chorλ. Consider the choreography $M = ((\lambda x : \mathsf{Int}@\mathsf{p}.\lambda y : \mathsf{Int}@\mathsf{q}.3@\mathsf{p})\ f(\mathsf{p}))\ 4@\mathsf{q}$. We have $[\![M]\!]_{\mathsf{q}} = ((\lambda x : \bot.\lambda y : \mathsf{Int}.\bot)\ \bot)\ 4$, which can reduce to $\bot$ in two steps. Reducing M accordingly requires being able to instantiate y as $4@\mathsf{q}$ even if $f(\mathsf{p})$ diverges. For this, and other cases of functions whose divergence blocks actions, Chorλ has a set of rewriting rules (see Figure 5). In our example, M can be rewritten as $(\lambda x : \mathsf{Int}@\mathsf{p}.(\lambda y : \mathsf{Int}@\mathsf{q}.3@\mathsf{p}\ 4@\mathsf{q}))\ f(\mathsf{p})$ by using rule R-AbsR, which can reduce to $(\lambda x : \mathsf{Int}@\mathsf{p}.3@\mathsf{p})\ f(\mathsf{p})$ as needed. In the rewriting rules that move a subterm in a lefthandside further in, the synchronising processes of the subterm, $\mathrm{spn}(M)$, is used to prevent rewritings that would change the order of communications. To use the rewritings in the semantics we add the rule

$$\frac{M \rightsquigarrow^* N \quad N \xrightarrow{\tau,\mathsf{P}} M'}{M \xrightarrow{\tau,\mathsf{P}}_D M'}\ [\text{Str}]$$

As before, equivalent rules must be added to the semantics of our process language (see Figure 6), and the reduction relation is closed under these rewritings. This allows $[\![M]\!]_{\mathsf{p}} = ((\lambda x : \mathsf{Int}.\lambda y : \bot.3)\ f_1())\ \bot$ to be rewritten to $(\lambda x : \mathsf{Int}.(\lambda y : \bot.3\ \bot))\ f_1()$, which can reduce to $(\lambda x : \mathsf{Int}.3)\ f_1()$.

4.3 Properties

Thanks to the extensions discussed in this section, our results can be generalised to the full language of Chorλ with recursion.

▶ **Theorem 25** (Completeness). *Given a closed choreography M, if $M \xrightarrow{\tau,\mathsf{P}}_D M'$ and $\Theta; \Sigma; \Gamma \vdash M : T$ and $[\![M]\!]$ is defined, then there exist networks $\mathcal{N}$ and M'' such that: $[\![M]\!] \rightarrow^+_{[\![D]\!]} \mathcal{N}$; $M' \rightarrow^* M''$; and $\mathcal{N} \sqsupseteq [\![M'']\!]$.*

▶ **Theorem 26** (Soundness). *Given a closed choreography M, if $\Theta;\Gamma \vdash M : T$ and $[\![M]\!] \to^* \mathcal{N}$ for some network $\mathcal{N}$, then there exist a choreography M', and a network $\mathcal{N}'$ such that: $M \to_D^* M'$; $\mathcal{N} \to^* \mathcal{N}'$; and $\mathcal{N}' \sqsupseteq [\![M']\!]$.*

From Theorems 25 and 26 and the type preservation and progress results from [6], we get the following corollary about deadlock-freedom. Specifically, the EPP of a well-typed closed choreography can keep reducing until all processes contain only local values (which denotes termination).

▶ **Corollary 27** (Deadlock-freedom). *Given a closed choreography M and a function environment D containing all the functions of M, if $\Theta;\Sigma;\Gamma \vdash M : T$ and $\Theta;\Sigma;\Gamma \vdash D$, then: whenever $[\![M]\!] \to_{[\![D]\!]}^* \mathcal{N}$ for some network $\mathcal{N}$, either there exists P and $\mathcal{N}'$ such that $\mathcal{N} \xrightarrow{\tau_{\mathsf{P}}}_{[\![D]\!]} \mathcal{N}'$ or $\mathcal{N} = \prod_{\mathsf{p} \in \mathrm{pn}(M)} \mathsf{p}[L_{\mathsf{p}}]$.*

We also show that adding recursion does not stop our projection being modular.

▶ **Proposition 28.** *The EPP $[\![-]\!]$ given in Definition 8 and extended with the equations in Section 4.1 is modular.*

Proof. The only change to the projection of choreographies is adding the projection of $f(\vec{\mathsf{p}})$, for which Lemma 13 still holds. Since no new contexts have been added, projection is then still modular. ◀

5 EAP

We now use our theory of EPP to obtain an implementation of the core of the Extensible Authentication Protocol (EAP) [28], which was modelled as a choreography in [6]. EAP is a widely-employed link-layer protocol that defines an authentication framework allowing a peer P to authenticate with a backend authentication server S, with the communication passing through an authenticator A that acts as an access point for the network.

The framework provides a core protocol parametrised over a set of authentication methods (either predefined or custom vendor-specific ones), modelled as individual choreographies with type $\mathsf{AuthMethod@(P,A,S)} = \mathsf{String@S} \to_{\{\mathsf{P,A}\}} \mathsf{Bool@S}$.

For reasons of modularity, it is desirable that the core of the protocol be written in a way that does not assume any particular authentication method. The $\mathsf{eap(P,A,S)}$ choreography does exactly that by leveraging higher-order composition of choreographies:

```
eap(P,A,S) = λmethods : [AuthMethod]@(P,A,S).
  eapAuth(P,A,S) (eapIdentity(P,A,S) "Auth request"@S) methods

eapAuth(P,A,S) = λid : String@S. λmethods : [AuthMethod]@(P,A,S).
  if empty(P,A,S) methods then
    eapFailure(P,A,S) "Try again later"@S
  else
    if (fst methods) id then
      select_{S,P} ok (select_{S,A} ok (eapSuccess(P,A,S) "Welcome"@S))
    else
      select_{S,P} ko (select_{S,A} ko (eapAuth(P,A,S) id (snd methods)))
```

For the sake of simplicity, we have left out the definitions of a couple of helper choreographies that are referenced in the example:

$\mathsf{eapIdentity(P,A,S)} : \mathsf{String@S} \to_{\{\mathsf{P,A}\}} \mathsf{String@S}$

$$\begin{aligned}\mathsf{empty}(\mathsf{P},\mathsf{A},\mathsf{S}) &: [\mathsf{AuthMethod}]@(\mathsf{P},\mathsf{A},\mathsf{S}) \to \mathsf{Bool}@(\mathsf{P},\mathsf{A},\mathsf{S})\\ \mathsf{eapSuccess}(\mathsf{P},\mathsf{A},\mathsf{S}) &: \mathsf{String}@\mathsf{S} \to (\mathsf{String}@\mathsf{P} \times \mathsf{String}@\mathsf{A})\\ \mathsf{eapFailure}(\mathsf{P},\mathsf{A},\mathsf{S}) &: \mathsf{String}@\mathsf{S} \to (\mathsf{String}@\mathsf{P} \times \mathsf{String}@\mathsf{A})\end{aligned}$$

First, eap(P, A, S) fetches the client's identity using eapIdentity(P, A, S), a function which exchanges the necessary EAP packets and delivers the client's identity to the server. Once the identity is known, eapAuth(P, A, S) is invoked in order to try the list of authentication methods until one succeeds, or the list is exhausted and authentication fails.

EAP is parametric on a list of choreographies called *methods*. We use the notation for lists in [AuthMethod]@(P, A, S) as described in Example 21, as well as the **if** M **then** M' **else** M'' construct which is just syntactic sugar for the previously described **case** M **of** **Inl** $x \Rightarrow M'$; **Inr** $x \Rightarrow M''$. Each authentication method can be an arbitrarily-complex choreography with its own communication structures that can involve all three involved processes, and it implements a particular authentication method on top of EAP.

The function empty(P, A, S) is used to determine whether the list of methods is empty. Recall the distributed boolean from Example 15, and note how we now use the same idea to minimise unnecessary communication while still guaranteeing that every process has the necessary information. The return type of this function, Bool@(P, A, S), denotes that the function uniformly returns either true (**Inl** ()) or false (**Inr** ()) at all of P, A, and S. That is, the result is guaranteed to be the same at these three processess. Since agreement is guaranteed, each process can locally check its own value without having to perform any selections. This is in contrast to the return type of each authentication method, Bool@S, meaning that only the server S has the authority of determining whether the authentication method was successful or not.

Finally, depending on the outcome of the authentication, an appropriate EAP packet is delivered by using either eapSuccess(P, A, S) or eapFailure(P, A, S) to indicate the result to the client.

```
$\mathsf{eap}_1(\mathsf{A},\mathsf{S}) = \lambda methods : [\mathsf{AuthMethod}].$
   $\mathsf{eapAuth}_1(\mathsf{A},\mathsf{S})\ (\mathsf{eapIdentity}_1(\mathsf{A},\mathsf{S})\ \bot)\ methods$

$\mathsf{eap}_2(\mathsf{P},\mathsf{S}) = \lambda methods : [\mathsf{AuthMethod}].$
   $\mathsf{eapAuth}_2(\mathsf{P},\mathsf{S})\ (\mathsf{eapIdentity}_2(\mathsf{P},\mathsf{S})\ \bot)\ methods$

$\mathsf{eap}_3(\mathsf{P},\mathsf{A}) = \lambda methods : [\mathsf{AuthMethod}].$
   $\mathsf{eapAuth}_3(\mathsf{P},\mathsf{A})\ (\mathsf{eapIdentity}_3(\mathsf{P},\mathsf{A})$ "Auth request"$)\ methods$
```

It is interesting to look at the projections of eapAuth(P, A, S) for each of the three participants, which follow below. For the purposes of projection, we desugar the if-then-else construct.

```
$\mathsf{eapAuth}_1(\mathsf{A},\mathsf{S}) = \lambda id : \bot.\ \lambda methods : [\mathsf{AuthMethod}].$
   case $\mathsf{empty}_1(\mathsf{A},\mathsf{S})\ methods$ of
      Inl _ $\Rightarrow \mathsf{eapFailure}_1(\mathsf{A},\mathsf{S})\ \bot$
      Inr _ $\Rightarrow \&_S\{\mathsf{ok} : \mathsf{eapSuccess}_1(\mathsf{A},\mathsf{S})\ \bot$
                  $\mathsf{ko} : \mathsf{eapAuth}_1(\mathsf{A},\mathsf{S})\ \bot\ (\mathbf{snd}\ methods)\}$

$\mathsf{eapAuth}_2(\mathsf{P},\mathsf{S}) = \lambda id : \bot.\ \lambda methods : [\mathsf{AuthMethod}].$
   case $\mathsf{empty}_2(\mathsf{P},\mathsf{S})\ methods$ of
      Inl _ $\Rightarrow \mathsf{eapFailure}_2(\mathsf{P},\mathsf{S})\ \bot$
      Inr _ $\Rightarrow \&_S\{\mathsf{ok} : (\mathsf{eapSuccess}_2(\mathsf{P},\mathsf{S})\ \bot)$
                  $\mathsf{ko} : (\mathsf{eapAuth}_2(\mathsf{P},\mathsf{S})\ \bot\ (\mathbf{snd}\ methods))$
```

```
$\mathsf{eapAuth}_3(\mathsf{P},\mathsf{A}) = \lambda id : \mathsf{String}.\ \lambda methods : [\mathsf{AuthMethod}].$
    case $\mathsf{empty}_3(\mathsf{P},\mathsf{A})\ methods$ of
        Inl _ ⇒ $\mathsf{eapFailure}_3(\mathsf{P},\mathsf{A})$ "Try again later"
        Inr _ ⇒ case (fst $methods$) $id$ of
                    Inl _ ⇒ $\oplus_{\mathsf{P}}$ ok ($\oplus_{\mathsf{A}}$ ok ($\mathsf{eapSuccess}_3(\mathsf{P},\mathsf{A})$ "Welcome"))
                    Inr _ ⇒ $\oplus_{\mathsf{P}}$ ko ($\oplus_{\mathsf{A}}$ ko ($\mathsf{eapAuth}_3(\mathsf{P},\mathsf{A})$ $id$ (snd $methods$)))
```

Note that the implementation of the check $\mathsf{empty}(\mathsf{P},\mathsf{A},\mathsf{S})$ *methods* at each process is completely local, i.e., it does not perform communications. This is possible because all processes have access to the same list. Afterwards however, only the server S is capable of determining whether the authentication method was successful or not, and has to communicate that result to the other two participants by means of selections.

6 Related Work

We already discussed the most related work on choreographic programming and EPP in Section 1. In this section, we discuss some technical aspects of our development in the context of previous work more in detail.

In our process language, the terms for communication actions (send, receive, selection, and branching) are adaptations to the functional setting of standard primitives from traditional imperative choreographic programming [8, 10, 21] and the local language of multiparty session types (choreographies without computation) [20, 19, 3]. A similar adaptation was carried out in [27] for the different setting of multi-threading (their primitives are not based on process names, but shared channels). Modelling a network as a map from process names to programs was previously done in [9, 24]. The idea of reporting the names of the involved processes in transition labels comes from [2, 19, 9, 24].

The first attempt at adding higher-order composition to choreographies goes back to [11], for a choreographic language that cannot express data nor computation (it is an abstract specification language). The approach in [11] adopts centralised coordination: resolving a choreographic application ($M\ M'$ in Chorλ, with M' involving more than one process) requires that the programmer picks a process as central coordinator, which then orchestrates the other processes with multicasts. This coordination effectively acts as a barrier, so processes cannot perform their own local computations independently of each other when higher-order composition is involved. Ten years after [11], another attempt at a notion of EPP for higher-order choreographies was proposed in [18]. The language in [18] is more expressive, i.e., it supports expressing computation at processes. However, this feature came at a cost: it is even more centralised than [11]. In particular, every application in a choreography requires that all processes generated by projection go through a global barrier that involves the entire system. The global barrier is modelled as a middleware in the semantics of the language, and involves even processes that do not contribute at all to the function or its arguments. Because processes need to participate also in the resolution of applications that do not involve them, the notion of EPP in [18] is not modular.

In contrast to [11] and [18], Chorλ presents no "hidden" barriers: coordination among processes is left to the programmer of the choreography, and EPP inserts no hidden synchronisations. Our EPP thus generates more concurrent and faithful implementations. It is also the first modular EPP for functional choreographic programming: changing the behaviours of some processes in a choreography requires re-running EPP only for those processes. This is important for the application of choreographic programming to DevOps (continuous integration and deployment), library management, and modularity in general.

Another related line of work is that on multitier programming and its progenitor calculus, Lambda 5 [25]. Similarly to Chorλ, Lambda 5 and multitier languages have data types with locations [29]. However, they are used very differently. In choreographic languages (thus Chorλ), programs have a "global" point of view and express how multiple processes interact with each other. By contrast, in multitier programming programs have the usual "local" point of view of a single process but they can nest (local) code that is supposed to be executed remotely. The reader interested in a detailed comparison of choreographic and multitier programming can consult [17], which presents algorithms for translating choreographies to multitier programs and vice versa. The correctness of these algorithms has never been proven, because they use an informally-specified fragment of Choral as a representative choreographic language. We conjecture that the introduction of an EPP for Chorλ could be the basis for a future comparison of the compilations for choreographic programs (in terms of Chorλ) and multitier programs (in terms of Lambda 5).

To the best of our knowledge, no other work supports distributed choice types. The nearest feature is presented in [21], where choreographic conditionals for a first-order calculus can be conjunctions of local conditions at different processes. These conditions must be checked to be consistent by means of separate proofs given in a Hoare-like logic. Our syntax is more general, since conditions can be choreographies, and our EPP requires no such additional proofs. However, using a Hoare logic in [21] gives some interesting flexibility, in that agreement does not need to be encoded as distributed sum types. In the future, it could be interesting to integrate the two approaches such that agreement could be proved by using a logic and then made manifest to EPP through our distributed choice types.

7 Conclusion and Future Work

We have presented a new theory of compilation for higher-order functional choreographies, which introduces modularity and decentralisation.

Our development validates the design of Chorλ [6], but it also reveals that in the case without recursion it can be significantly simplified: reduction rules for out-of-order execution were not necessary until we had to deal with divergence. In particular, we have shown that the fragment of Chorλ without recursion can be modelled by simple semantics and still achieve the standard deadlock-freedom by design property. However, once recursion is added, a more sophisticated semantics allowing for out-of-order execution is required. This stems from the structure of a functional choreography being different than traditional imperative choreographies.

Our study fills a knowledge gap that is relevant for the future implementations and applications of choreographic languages. An ad-hoc distributed implementation of higher-order choreographies exists already in the Choral programming language [16]. However, Choral is a large object-oriented language that extends Java, meaning that it is not practical to formally study and prove the standard results expected of a choreographic language. We have been able to prove these results – correspondence between choreography and projected distributed implementation (Theorems 25 and 26) and deadlock-freedom (Corollary 27) – because Chorλ captures the essence of higher-order choreographic composition in a small language based on the λ-calculus. Our EPP is largely consistent with the implementation of the Choral compiler, but there are two key differences, both caused by Chorλ being based on the λ-calculus. First, since Choral is an object-oriented language, not every expression needs to return a value even if the result of the expression is located elsewhere as in **send**; therefore, Choral does not need a $\bot$ construct. Second, Choral does not have distributed choice types and instead restricts all conditions to be local (at one process). Thus, our distributed choice types could form the basis for an interesting extension of Choral.

Aside from Choral, existing choreographic programming languages either have no higher-order constructs (e.g., Scribble [30], a language based on multiparty session types [19]), or have the compilation of their higher-order constructs lack modularity and decentralisation (e.g., Pirouette [18]). Our results provide a foundation for adding mechanisms for higher-order composition to other choreographic and similar languages with modular compilation.

Future Work

Synchronous communication is widely adopted in theories of processes and is usually implemented in practice by using acknowledgements. A potential extension of Chorλ is adding support for asynchronous communication, which is usually achieved by adding message queues and choreographic terms to represent partially-executed communications [12, 7, 14, 24].

Another potential extension of Chorλ, our process language, and our theory of EPP would be to enable abstraction over process names, that is, extending the syntax such that values can be the names of processes to be acted upon. This could, for example, enable the modelling of choreographies with dynamic topologies, where processes discover whom they have to interact with at runtime.

References

1 Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2):8:1–8:78, 2012. doi:10.1145/2220365.2220367.

2 Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In Roberto Giacobazzi and Radhia Cousot, editors, *Procs. POPL*, pages 263–274. ACM, 2013. doi:10.1145/2429069.2429101.

3 Marco Carbone, Fabrizio Montesi, Carsten Schürmann, and Nobuko Yoshida. Multiparty session types as coherence proofs. *Acta Informatica*, 54(3):243–269, 2017. doi:10.1007/s00236-016-0285-y.

4 Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party session. *Log. Methods Comput. Sci.*, 8(1), 2012. doi:10.2168/LMCS-8(1:24)2012.

5 Alonzo Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):346–366, 1932. URL: http://www.jstor.org/stable/1968337.

6 Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. Functional choreographic programming. In Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu, editors, *Theoretical Aspects of Computing – ICTAC 2022 – 19th International Colloquium, Tbilisi, Georgia, September 27-29, 2022, Proceedings*, volume 13572 of *Lecture Notes in Computer Science*, pages 212–237. Springer, 2022. doi:10.1007/978-3-031-17715-6_15.

7 Luís Cruz-Filipe and Fabrizio Montesi. On asynchrony and choreographies. In Massimo Bartoletti, Laura Bocchi, Ludovic Henrio, and Sophia Knight, editors, *Proceedings 10th Interaction and Concurrency Experience, ICE@DisCoTec 2017, Neuchâtel, Switzerland, 21-22nd June 2017*, volume 261 of *EPTCS*, pages 76–90, 2017. doi:10.4204/EPTCS.261.8.

8 Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theor. Comput. Sci.*, 802:38–66, 2020. doi:10.1016/j.tcs.2019.07.005.

9 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Certifying choreography compilation. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing – ICTAC 2021 – 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings*, volume 12819 of *Lecture Notes in Computer Science*, pages 115–133. Springer, 2021. doi:10.1007/978-3-030-85315-0_8.

10 Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. Formalising a turing-complete choreographic language in coq. In Liron Cohen and Cezary Kaliszyk, editors, *12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference)*, volume 193 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ITP.2021.15.

11 Romain Demangeon and Kohei Honda. Nested protocols in session types. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 – Concurrency Theory – 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*, volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012. `doi:10.1007/978-3-642-32940-1_20`.

12 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In Fedor V. Fomin, Rusins Freivalds, Marta Z. Kwiatkowska, and David Peleg, editors, *Automata, Languages, and Programming – 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II*, volume 7966 of *Lecture Notes in Computer Science*, pages 174–186. Springer, 2013. `doi:10.1007/978-3-642-39212-2_18`.

13 Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Inf. Theory*, 22(6):644–654, 1976. `doi:10.1109/TIT.1976.1055638`.

14 Simon Fowler, Sam Lindley, and Philip Wadler. Mixing metaphors: Actors as channels and channels as actors. In Peter Müller, editor, *31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain*, volume 74 of *LIPIcs*, pages 11:1–11:28. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.ECOOP.2017.11`.

15 Saverio Giallorenzo, Ivan Lanese, and Daniel Russo. Chip: A choreographic integration process. In Hervé Panetto, Christophe Debruyne, Henderik A. Proper, Claudio Agostino Ardagna, Dumitru Roman, and Robert Meersman, editors, *Procs. OTM, part II*, volume 11230 of *Lecture Notes in Computer Science*, pages 22–40. Springer, 2018. `doi:10.1007/978-3-030-02671-4_2`.

16 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Object-oriented choreographic programming. *CoRR*, abs/2005.09520, 2020. URL: `https://arxiv.org/abs/2005.09520`.

17 Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. Multiparty Languages: The Choreographic and Multitier Cases. In *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 12-17, 2021, Aarhus, Denmark (Virtual Conference)*, LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2021. To appear. Pre-print available at `https://fabriziomontesi.com/files/gmprsw21.pdf`.

18 Andrew K. Hirsch and Deepak Garg. Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022. `doi:10.1145/3498684`.

19 Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *J. ACM*, 63(1):9, 2016. Also: POPL, pages 273–284, 2008. `doi:10.1145/2827695`.

20 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. `doi:10.1145/2873052`.

21 Sung-Shik Jongmans and Petra van den Bos. A predicate transformer for choreographies – computing preconditions in choreographic programming. In Ilya Sergey, editor, *Programming Languages and Systems – 31st European Symposium on Programming, ESOP 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings*, volume 13240 of *Lecture Notes in Computer Science*, pages 520–547. Springer, 2022. `doi:10.1007/978-3-030-99336-8_19`.

22 Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proc. of ASPLOS*, pages 517–530, 2016.

23 Fabrizio Montesi. *Choreographic Programming.* Ph.D. Thesis, IT University of Copenhagen, 2013. `http://www.fabriziomontesi.com/files/choreographic-programming.pdf`.

24 Fabrizio Montesi. *Introduction to Choreographies.* Cambridge University Press, 2023.

25 Tom Murphy VII, Karl Crary, Robert Harper, and Frank Pfenning. A symmetric modal lambda calculus for distributed computing. In *19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings*, pages 286–295. IEEE Computer Society, 2004. `doi:10.1109/LICS.2004.1319623`.

26 Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Commun. ACM*, 21(12):993–999, 1978. doi:10.1145/359657.359659.

27 Vasco Thudichum Vasconcelos, Simon J. Gay, and António Ravara. Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.*, 368(1-2):64–87, 2006. doi:10.1016/j.tcs.2006.06.028.

28 John Vollbrecht, James D. Carlson, Larry Blunk, Dr. Bernard D. Aboba, and Henrik Levkowetz. Extensible Authentication Protocol (EAP). RFC 3748, June 2004. doi:10.17487/RFC3748.

29 Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. A survey of multitier programming. *ACM Comput. Surv.*, 53(4):81:1–81:35, 2020. doi:10.1145/3397495.

30 Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In Martín Abadi and Alberto Lluch-Lafuente, editors, *Trustworthy Global Computing – 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers*, volume 8358 of *Lecture Notes in Computer Science*, pages 22–41. Springer, 2013. doi:10.1007/978-3-319-05119-2_3.

A Full definitions and proofs

▶ **Definition 29** (Free Variables). *Given a choreography M, the free variables of M, $\mathrm{fv}(M)$ are defined as:*

$$\begin{array}{ll}
\mathrm{fv}(N\ N') = \mathrm{fv}(N) \cup \mathrm{fv}(N') & \mathrm{fv}(\mathbf{select}_{\mathsf{q},\mathsf{p}}\ l\ M) = \mathrm{fv}(M) \\
\mathrm{fv}(x) = x & \mathrm{fv}(\lambda x : T.N) = \mathrm{fv}(N)\backslash\{x\} \\
\mathrm{fv}(()@\mathsf{p}) = \varnothing & \mathrm{fv}(\mathbf{com}_{\mathsf{q},\mathsf{p}}) = \varnothing \\
\mathrm{fv}(f(\vec{\mathsf{p}})) = \varnothing & \mathrm{fv}(\mathbf{Pair}\ V\ V') = \mathrm{fv}(V) \cup \mathrm{fv}(V') \\
\multicolumn{2}{l}{\mathrm{fv}(\mathbf{case}\ N\ \mathbf{of}\ \mathbf{Inl}\ x \Rightarrow M; \mathbf{Inr}\ y \Rightarrow M') = \mathrm{fv}(N) \cup (\mathrm{fv}(M)\backslash\{x\}) \cup (\mathrm{fv}(M')\backslash\{y\})} \\
\mathrm{fv}(\mathbf{fst}) = \mathrm{fv}(\mathbf{snd}) = \varnothing & \mathrm{fv}(\mathbf{Inl}\ V) = \mathrm{fv}(\mathbf{Inr}\ V) = \mathrm{fv}(V)
\end{array}$$

▶ **Definition 30** (Bound Variables). *Given a choreography M, the bound variables of M, $\mathrm{bv}(M)$ are defined as:*

$$\begin{array}{ll}
\mathrm{bv}(N\ N') = \mathrm{bv}(N) \cup \mathrm{bv}(N') & \mathrm{bv}(\mathbf{select}_{\mathsf{q},\mathsf{p}}\ l\ M) = \mathrm{bv}(M) \\
\mathrm{bv}(x) = \varnothing & \mathrm{bv}(\lambda x : T.N) = \mathrm{bv}(N) \cup \{x\} \\
\mathrm{bv}(()@\mathsf{p}) = \varnothing & \mathrm{bv}(\mathbf{com}_{\mathsf{q},\mathsf{p}}) = \varnothing \\
\mathrm{bv}(f(\vec{\mathsf{p}})) = \varnothing & \mathrm{fv}(\mathbf{Pair}\ V\ V') = \mathrm{bv}(V) \cup \mathrm{bv}(V') \\
\multicolumn{2}{l}{\mathrm{bv}(\mathbf{case}\ N\ \mathbf{of}\ \mathbf{Inl}\ x \Rightarrow M; \mathbf{Inr}\ y \Rightarrow M') = \mathrm{bv}(N) \cup \mathrm{bv}(M) \cup \{x\} \cup (\mathrm{bv}(M') \cup \{y\})} \\
\mathrm{bv}(\mathbf{fst}) = \mathrm{bv}(\mathbf{snd}) = \varnothing & \mathrm{bv}(\mathbf{Inl}\ V) = \mathrm{bv}(\mathbf{Inr}\ V) = \mathrm{bv}(V)
\end{array}$$

▶ **Definition 31** (Process names of a type). *The process names of a type T, $\mathrm{pn}(T)$, are defined as follows.*

$$\begin{array}{rr}
\mathrm{pn}(t@\vec{\mathsf{R}}) = \vec{\mathsf{R}} & \mathrm{pn}(T \rightarrow_{\rho} T') = \mathrm{pn}(T) \cup \mathrm{pn}(T') \cup \rho \\
\mathrm{pn}(()@\mathsf{R}) = \{\mathsf{R}\} & \mathrm{pn}(T + T') = \mathrm{pn}(T \times T') = \mathrm{pn}(T) \cup \mathrm{pn}(T')
\end{array}$$

▶ **Definition 32** (Process names of a choreography). *The process names of a choreography M, $\mathrm{pn}(M)$, are defined as follows.*

$$\begin{aligned}
\mathrm{pn}(M\ N) &= \mathrm{pn}(M) \cup \mathrm{pn}(N) \\
\mathrm{pn}(\mathbf{select}_{\mathsf{p},\mathsf{q},}\ l\ M) &= \{\mathsf{p},\mathsf{q}\} \cup \mathrm{pn}(M) \\
\mathrm{pn}(x) &= \varnothing \\
\mathrm{pn}(\mathbf{case}\ M\ \mathbf{of}\ \mathbf{Inl}\ x \Rightarrow N; \mathbf{Inr}\ y \Rightarrow N') &= \mathrm{pn}(M) \cup \mathrm{pn}(N) \cup \mathrm{pn}(N') \\
\mathrm{pn}(\lambda x : T.M) &= \mathrm{pn}(T) \cup \mathrm{pn}(M) \\
\mathrm{pn}(\mathbf{Inl}\ V) = (\mathrm{pn}\,\mathbf{Inr}\ V) &= \mathrm{pn}(V)
\end{aligned}$$

$$\frac{\Theta';\Sigma;\Gamma,x:T\vdash M:T' \quad \rho\cup \mathrm{pn}(T)\cup\mathrm{pn}(T')=\Theta'\subseteq\Theta}{\Theta;\Sigma;\Gamma\vdash \lambda x:T.M:T\to_\rho T'}\ [\text{TAbs}]$$

$$\frac{x:T\in\Gamma \quad \mathrm{pn}(T)\subseteq\Theta}{\Theta;\Sigma;\Gamma\vdash x:T}\ [\text{TVar}] \qquad \frac{\Theta;\Sigma;\Gamma\vdash N:T\to_\rho T' \quad \Theta;\Sigma;\Gamma\vdash M:T}{\Theta;\Sigma;\Gamma\vdash N\ M:T'}\ [\text{TApp}]$$

$$\frac{\Theta;\Sigma;\Gamma\vdash N:T_1+T_2 \quad \Theta;\Sigma;\Gamma,x:T_1\vdash M':T \quad \Theta;\Sigma;\Gamma,x':T_2\vdash M'':T}{\Theta;\Sigma;\Gamma\vdash \textbf{case}\ N\ \textbf{of Inl}\ x\Rightarrow M';\ \textbf{Inr}\ x'\Rightarrow M'':T}\ [\text{TCase}]$$

$$\frac{\Theta;\Sigma;\Gamma\vdash M:T \quad \mathsf{q},\mathsf{p}\in\Theta}{\Theta;\Sigma;\Gamma\vdash \textbf{select}_{\mathsf{q},\mathsf{p}}\ l\ M:T}\ [\text{TSel}]$$

$$\frac{f(\vec{\mathsf{p}}'):T\in\Gamma \quad \mathrm{pn}(T)\subseteq\vec{\mathsf{p}}'\subseteq\Theta \quad ||\vec{\mathsf{p}}||=||\vec{\mathsf{p}}'|| \quad \mathrm{distinct}(\vec{\mathsf{p}})}{\Theta;\Sigma;\Gamma\vdash f(\vec{\mathsf{p}}):T[\vec{\mathsf{p}}':=\vec{\mathsf{p}}]}\ [\text{TFun}]$$

$$\frac{\mathsf{p}\in\Theta}{\Theta;\Sigma;\Gamma\vdash ()@\mathsf{p}:()@\mathsf{p}}\ [\text{TUnit}] \qquad \frac{\mathsf{q},\mathsf{p}\in\Theta \quad \mathrm{pn}(T)=\mathsf{q}}{\Theta;\Sigma;\Gamma\vdash \textbf{com}_{\mathsf{q},\mathsf{p}}:T\to_\varnothing T[\mathsf{q}:=\mathsf{p}]}\ [\text{TCom}]$$

$$\frac{\Theta;\Sigma;\Gamma\vdash V:T \quad \Theta;\Sigma;\Gamma\vdash V':T'}{\Theta;\Sigma;\Gamma\vdash \textbf{Pair}\ V\ V':(T\times T')}\ [\text{TPair}]$$

$$\frac{\mathrm{pn}(T\times T')\subseteq\Theta}{\Theta;\Sigma;\Gamma\vdash \textbf{fst}:(T\times T')\to_\varnothing T}\ [\text{TProj1}] \qquad \frac{\mathrm{pn}(T\times T')\subseteq\Theta}{\Theta;\Sigma;\Gamma\vdash \textbf{snd}:(T\times T')\to_\varnothing T'}\ [\text{TProj2}]$$

$$\frac{\Theta;\Sigma;\Gamma\vdash V:T \quad \mathrm{pn}(T+T')\subseteq\Theta}{\Theta;\Sigma;\Gamma\vdash \textbf{Inl}\ V:(T+T')}\ [\text{TInl}] \qquad \frac{\Theta;\Sigma;\Gamma\vdash V:T' \quad \mathrm{pn}(T+T')\subseteq\Theta}{\Theta;\Sigma;\Gamma\vdash \textbf{Inr}\ V:(T+T')}\ [\text{TInR}]$$

$$\frac{\Theta;\Sigma;\Gamma\vdash M:t@\vec{\mathsf{p}} \quad t@\vec{\mathsf{p}}'=_\Sigma T \quad ||\vec{\mathsf{p}}||=||\vec{\mathsf{p}}'|| \quad \mathrm{distinct}(\vec{\mathsf{p}})}{\Theta;\Sigma;\Gamma\vdash M:T[\vec{\mathsf{p}}':=\vec{\mathsf{p}}]}\ [\text{TEq}]$$

$$\frac{\forall f(\vec{\mathsf{p}})\in\mathsf{dom}(D):\quad f(\vec{\mathsf{p}}):T\in\Gamma \quad \vec{\mathsf{p}};\Sigma;\Gamma\vdash D(f(\vec{\mathsf{p}})):T \quad \mathrm{distinct}(\vec{\mathsf{p}}) \quad \vec{\mathsf{p}}\subseteq\Theta}{\Theta;\Sigma;\Gamma\vdash D}\ [\text{TDefs}]$$

Figure 7 Full set of typing rules for Chorλ.

$$\mathrm{pn}(\textbf{Pair}\ V\ V')=\mathrm{pn}(V)\cup\mathrm{pn}(V')$$
$$\mathrm{pn}(\textbf{fst})=\mathrm{pn}(\textbf{snd})=\varnothing$$
$$\mathrm{pn}(\textbf{com}_{\mathsf{p},\mathsf{q},})=\{\mathsf{p},\mathsf{q}\}$$

▶ **Definition 33.** *We define the set of synchronising processes of a choreography M, $\mathrm{spn}(M)$, by recursion on the structure of M:*
$\mathrm{spn}(\textbf{com}_{\mathsf{S},\mathsf{R}})=\{S,R\}$, $\mathrm{spn}(\textbf{select}_{\mathsf{S},\mathsf{R}}\ l\ M)=\{S,R\}\cup\mathrm{spn}(M)$,
$\mathrm{spn}(f(\vec{R}))=\vec{R}$, *and homomorphically on all other cases.*

▶ **Definition 34** (Merging). *Given two behaviours B and B', $B\sqcup B'$ is defined as follows.*

$$B_1\ B_2\sqcup B_1'\ B_2'=(B_1\sqcup B_1')\ (B_2\sqcup B_2')$$
$$\textbf{case}\ B_1\ \textbf{of Inl}\ x\Rightarrow B_2;\ \textbf{Inr}\ y\Rightarrow B_3\sqcup\textbf{case}\ B_1'\ \textbf{of Inl}\ x\Rightarrow B_2';\ \textbf{Inr}\ y\Rightarrow B_3'=$$
$$\textbf{case}\ (B_1\sqcup B_1')\ \textbf{of Inl}\ x\Rightarrow(B_2\sqcup B_2');\ \textbf{Inr}\ y\Rightarrow(B_3\sqcup B_3')$$
$$\oplus_{\mathsf{p}}\ \ell\ B\sqcup\oplus_{\mathsf{p}}\ \ell\ B'=\oplus_{\mathsf{p}}\ \ell\ (B\sqcup B')$$
$$\&\{\ell_i:B_i\}_{i\in I}\sqcup\&\{\ell_j:B_j'\}_{j\in J}=\&\left(\{\ell_k:B_k\sqcup B_k'\}_{k\in I\cap J}\cup\{\ell_i:B_i\}_{i\in I\setminus J}\cup\{\ell_j:B_j'\}_{j\in J\setminus I}\right)$$
$$x\sqcup x=x \qquad \lambda x:T.B\sqcup\lambda x:T.B'=\lambda x:T.(B\sqcup B')$$

$$\frac{\mathrm{fv}(V) \cap \mathrm{bv}(M) = \varnothing}{\lambda x : T.M\ V \xrightarrow{\tau,\varnothing}_D M[x := V]}\ [\textsc{AppAbs}] \qquad \frac{M \xrightarrow{\ell,\mathsf{P}}_D M'}{\lambda x : T.M \xrightarrow{\lambda,\mathsf{P}}_D \lambda x : T.M'}\ [\textsc{InAbs}]$$

$$\frac{M \xrightarrow{\ell,\mathsf{P}}_D M' \quad \ell = \lambda \Rightarrow \mathsf{P} \cap \mathrm{pn}(N) = \varnothing}{M\ N \xrightarrow{\tau,\mathsf{P}}_D M'\ N}\ [\textsc{App1}]$$

$$\frac{N \xrightarrow{\tau,\mathsf{P}}_D N'}{V\ N \xrightarrow{\tau,\mathsf{P}}_D V\ N'}\ [\textsc{App2}] \qquad \frac{N \xrightarrow{\tau,\mathsf{P}}_D N' \quad \mathsf{P} \cap \mathrm{pn}(M) = \varnothing}{M\ N \xrightarrow{\tau,\mathsf{P}}_D M\ N'}\ [\textsc{App3}]$$

$$\frac{N \xrightarrow{\tau,\mathsf{P}}_D N'}{\mathbf{case}\ N\ \mathbf{of\ Inl}\ x \Rightarrow M;\ \mathbf{Inr}\ x' \Rightarrow M' \xrightarrow{\tau,\mathsf{P}}_D \mathbf{case}\ N'\ \mathbf{of\ Inl}\ x \Rightarrow M;\ \mathbf{Inr}\ x' \Rightarrow M'}\ [\textsc{Case}]$$

$$\frac{M_1 \xrightarrow{\ell,\mathsf{P}}_D M_1' \quad M_2 \xrightarrow{\ell,\mathsf{P}}_D M_2' \quad \mathsf{P} \cap \mathrm{pn}(N) = \varnothing}{\mathbf{case}\ N\ \mathbf{of\ Inl}\ x \Rightarrow M_1;\ \mathbf{Inr}\ x' \Rightarrow M_2 \xrightarrow{\ell,\mathsf{P}}_D \mathbf{case}\ N\ \mathbf{of\ Inl}\ x \Rightarrow M_1';\ \mathbf{Inr}\ x' \Rightarrow M_2'}\ [\textsc{InCase}]$$

$$\frac{}{\mathbf{case\ Inl}\ V\ \mathbf{of\ Inl}\ x \Rightarrow M;\ \mathbf{Inr}\ x' \Rightarrow M' \xrightarrow{\tau,\varnothing}_D M[x := V]}\ [\textsc{CaseL}]$$

$$\frac{}{\mathbf{case\ Inr}\ V\ \mathbf{of\ Inl}\ x \Rightarrow M;\ \mathbf{Inr}\ x' \Rightarrow M' \xrightarrow{\tau,\varnothing}_D M'[x' := V]}\ [\textsc{CaseR}]$$

$$\frac{}{\mathbf{fst\ Pair}\ V\ V' \xrightarrow{\tau,\varnothing}_D V}\ [\textsc{Proj1}] \qquad \frac{}{\mathbf{snd\ Pair}\ V\ V' \xrightarrow{\tau,\varnothing}_D V'}\ [\textsc{Proj2}]$$

$$\frac{D(f(\vec{\mathsf{p}'})) = M}{f(\vec{\mathsf{p}}) \xrightarrow{\tau,\varnothing}_D M[\vec{\mathsf{p}'} := \vec{\mathsf{p}}]}\ [\textsc{Def}]$$

$$\frac{\mathrm{fv}(V) = \varnothing}{\mathbf{com}_{\mathsf{q},\mathsf{p}}\ V \xrightarrow{\tau,\{\mathsf{q},\mathsf{p}\}}_D V[\mathsf{q} := \mathsf{p}]}\ [\textsc{Com}] \qquad \frac{}{\mathbf{select}_{\mathsf{q},\mathsf{p}}\ l\ M \xrightarrow{\tau,\{\mathsf{q},\mathsf{p}\}}_D M}\ [\textsc{Sel}]$$

$$\frac{M \xrightarrow{\ell,\mathsf{P}}_D M' \quad \mathsf{P} \cap \{\mathsf{q},\mathsf{p}\} = \varnothing}{\mathbf{select}_{\mathsf{q},\mathsf{p}}\ \ell\ M \xrightarrow{\ell,\mathsf{P}}_D \mathbf{select}_{\mathsf{q},\mathsf{p}}\ \ell\ M'}\ [\textsc{InSel}] \qquad \frac{M \rightsquigarrow^* N \quad N \xrightarrow{\tau,\mathsf{P}} N'}{M \xrightarrow{\tau,\mathsf{P}}_D M'}\ [\textsc{Str}]$$

Figure 8 Semantics of Chorλ.

$$\begin{gathered}
\mathbf{fst} \sqcup \mathbf{fst} = \mathbf{fst} \qquad \mathbf{snd} \sqcup \mathbf{snd} = \mathbf{snd} \\
\mathbf{Inl}\ L \sqcup \mathbf{Inl}\ L' = \mathbf{Inl}\ (L \sqcup L') \qquad \mathbf{Inr}\ L \sqcup \mathbf{Inr}\ L' = \mathbf{Inr}\ (L \sqcup L') \\
\mathbf{Pair}\ L_1\ L_2 \sqcup \mathbf{Pair}\ L_1'\ L_2' = \mathbf{Pair}\ (L_1 \sqcup L_1')\ (L_2 \sqcup L_2') \qquad f \sqcup f = f \\
\mathbf{recv}_{\mathsf{p}} \sqcup \mathbf{recv}_{\mathsf{p}} = \mathbf{recv}_{\mathsf{p}} \qquad \mathbf{send}_{\mathsf{p}} \sqcup \mathbf{send}_{\mathsf{p}} = \mathbf{send}_{\mathsf{p}} \qquad \bot \sqcup \bot = \bot
\end{gathered}$$

▶ **Definition 35** (Context). *We define a context* $C[]$ *in* Chorλ *as follows:*

$$\begin{aligned}
C[] ::= \ & [] \mid M\ C[] \mid C[]\ M \mid \mathbf{select}_{\mathsf{p},\mathsf{p}}\ l\ C[] \mid \mathbf{case}\ C[]\ \mathbf{of\ Inl}\ x \Rightarrow M;\ \mathbf{Inr}\ x \Rightarrow M \\
& \mid \mathbf{case}\ M\ \mathbf{of\ Inl}\ x \Rightarrow C[];\ \mathbf{Inr}\ x \Rightarrow M \mid \mathbf{case}\ M\ \mathbf{of\ Inl}\ x \Rightarrow M;\ \mathbf{Inr}\ x' \Rightarrow C[] \\
& \mid \lambda x : T.C[]
\end{aligned}$$

$$\frac{x \notin \mathrm{fv}(M')}{((\lambda x : T.M)\ N)\ M' \rightsquigarrow (\lambda x : T.(M\ M'))\ N)}\ [\text{R-AbsR}]$$

$$\frac{x \notin \mathrm{fv}(M') \quad \mathrm{spn}(M') \cap \mathrm{pn}(N) = \varnothing}{M'\ ((\lambda x : T.M)\ N) \rightsquigarrow (\lambda x : T.(M'\ M))\ N)}\ [\text{R-AbsL}]$$

$$\frac{x, x' \notin \mathrm{fv}(M)}{\begin{array}{l}(\textbf{case}\ N\ \textbf{of}\ \textbf{Inl}\ x \Rightarrow M_1; \textbf{Inr}\ x' \Rightarrow M_2)\ M \rightsquigarrow \\ \qquad \textbf{case}\ N\ \textbf{of}\ \textbf{Inl}\ x \Rightarrow (M_1\ M); \textbf{Inr}\ x' \Rightarrow (M_2\ M)\end{array}}\ [\text{R-CaseR}]$$

$$\frac{x, x' \notin \mathrm{fv}(M) \quad \mathrm{spn}(M) \cap \mathrm{pn}(N) = \varnothing}{\begin{array}{l}M\ (\textbf{case}\ N\ \textbf{of}\ \textbf{Inl}\ x \Rightarrow M_1; \textbf{Inr}\ x' \Rightarrow M_2) \rightsquigarrow \\ \qquad \textbf{case}\ N\ \textbf{of}\ \textbf{Inl}\ x \Rightarrow (M\ M_1); \textbf{Inr}\ x' \Rightarrow (M\ M_2)\end{array}}\ [\text{R-CaseL}]$$

$$\frac{}{(\textbf{select}_{\mathsf{q},\mathsf{p}}\ l\ N)\ M \rightsquigarrow \textbf{select}_{\mathsf{q},\mathsf{p}}\ l\ (N\ M)}\ [\text{R-SelR}]$$

$$\frac{\mathrm{spn}(M) \cap \mathrm{pn}(N) = \varnothing}{M\ (\textbf{select}_{\mathsf{q},\mathsf{p}}\ l\ N) \rightsquigarrow \textbf{select}_{\mathsf{q},\mathsf{p}}\ l\ (M\ N)}\ [\text{R-SelL}]$$

$$\frac{y \text{ fresh for } M}{\lambda x : T.M \rightsquigarrow \lambda y : T.M[x := y]}\ [\text{R-alph}]$$

Figure 9 Rewriting of Chorλ.

$$\frac{\mathrm{fv}(L)=\varnothing}{\textbf{send}_{\mathsf{p}}\ L \xrightarrow{\textbf{send}_{\mathsf{p}}\ L}_{\mathbb{D}} \bot}\ [\textsc{NSend}] \qquad \frac{}{\textbf{recv}_{\mathsf{p}}\ \bot \xrightarrow{\textbf{recv}_{\mathsf{p}}\ L}_{\mathbb{D}} L}\ [\textsc{NRecv}]$$

$$\frac{B \xrightarrow{\textbf{send}_{\mathsf{q}}\ L}_{\mathbb{D}(\mathsf{q})} B_1' \quad B_2 \xrightarrow{\textbf{recv}_{\mathsf{p}}\ L}_{\mathbb{D}(\mathsf{p})} B_2'}{\mathsf{q}[B_1] \mid \mathsf{p}[B_2] \xrightarrow{\tau_{\mathsf{q},\mathsf{p}}}_{\mathbb{D}} \mathsf{q}[B_1'] \mid \mathsf{p}[B_2']}\ [\textsc{NCom}]$$

$$\frac{}{\oplus_{\mathsf{p}}\ l\ B \xrightarrow{\oplus_{\mathsf{p}}\ l}_{\mathbb{D}} B}\ [\textsc{NCho}] \qquad \frac{}{\&_{\mathsf{p}}\{\ell_1 : B_1, \ldots, \ell_n : B_n\} \xrightarrow{\&_{\mathsf{p}}\ell_i}_{\mathbb{D}} B_i}\ [\textsc{NOff}]$$

$$\frac{B_i \xrightarrow{\mu}_{\mathbb{D}} B_i' \text{ for } 1 \leqslant i \leqslant n \quad \mu \in \{\tau, \lambda\}}{\&_{\mathsf{p}}\{\ell_1 : B_1, \ldots, \ell_n : B_n\} \xrightarrow{\mu}_{\mathbb{D}} \&_{\mathsf{p}}\{\ell_1 : B_1', \ldots, \ell_n : B_n'\}}\ [\textsc{NOff2}]$$

$$\frac{B \xrightarrow{\mu}_{\mathbb{D}} B' \quad \mu \in \{\tau, \lambda\}}{\oplus_{\mathsf{p}}\ l\ B \xrightarrow{\mu}_{\mathbb{D}} \oplus_{\mathsf{p}}\ l\ B'}\ [\textsc{NCho2}] \qquad \frac{B_1 \xrightarrow{\oplus_{\mathsf{p}}\ \ell}_{\mathbb{D}(\mathsf{q})} B_1' \quad B_2 \xrightarrow{\&_{\mathsf{q}}\ \ell}_{\mathbb{D}(\mathsf{p})} B_2'}{\mathsf{q}[B_1] \mid \mathsf{p}[B_2] \xrightarrow{\tau_{\mathsf{q},\mathsf{p}}}_{\mathbb{D}} \mathsf{q}[B_1'] \mid \mathsf{p}[B_2']}\ [\textsc{NSel}]$$

$$\frac{}{(\lambda x : T.B)\ L \xrightarrow{\tau}_{\mathbb{D}} B[x := L]}\ [\textsc{NAbsApp}] \qquad \frac{B \xrightarrow{\mu}_{\mathbb{D}} B' \quad \mu \in \{\tau, \lambda\}}{\lambda x : T.B \xrightarrow{\lambda}_{D} \lambda x : T.B'}\ [\textsc{NInAbs}]$$

$$\frac{B \xrightarrow{\mu}_{\mathbb{D}} B'' \quad \text{if } \mu = \lambda \text{ then } \mu' = \tau \text{ else } \mu' = \mu}{B\ B' \xrightarrow{\mu'}_{\mathbb{D}} B''\ B'}\ [\textsc{NApp1}]$$

$$\frac{B \xrightarrow{\mu}_{\mathbb{D}} B'}{L\ B \xrightarrow{\mu}_{\mathbb{D}} L\ B'}\ [\textsc{NApp2}] \qquad \frac{B' \xrightarrow{\tau}_{\mathbb{D}} B''}{B\ B' \xrightarrow{\tau}_{\mathbb{D}} B\ B''}\ [\textsc{NApp3}]$$

$$\frac{B \xrightarrow{\mu}_{\mathbb{D}} B'''}{\textbf{case}\ B\ \textbf{of Inl}\ x \Rightarrow B';\ \textbf{Inr}\ x' \Rightarrow B'' \xrightarrow{\mu}_{\mathbb{D}} \textbf{case}\ B'''\ \textbf{of Inl}\ x \Rightarrow B';\ \textbf{Inr}\ x' \Rightarrow B''}\ [\textsc{NCase}]$$

$$\frac{B_1 \xrightarrow{\mu}_{\mathbb{D}} B_1' \quad B_2 \xrightarrow{\mu}_{\mathbb{D}} B_2' \quad \mu \in \{\lambda, \tau\}}{\textbf{case}\ B\ \textbf{of Inl}\ x \Rightarrow B_1;\ \textbf{Inr}\ x' \Rightarrow B_2 \xrightarrow{\mu}_{\mathbb{D}} \textbf{case}\ B\ \textbf{of Inl}\ x \Rightarrow B_1';\ \textbf{Inr}\ x' \Rightarrow B_2'}\ [\textsc{NCase2}]$$

$$\frac{}{\textbf{case Inl}\ L\ \textbf{of Inl}\ x \Rightarrow B;\ \textbf{Inr}\ x' \Rightarrow B' \xrightarrow{\tau}_{\mathbb{D}} B[x := L]}\ [\textsc{NCaseL}]$$

$$\frac{}{\textbf{case Inr}\ L\ \textbf{of Inl}\ x \Rightarrow B;\ \textbf{Inr}\ x' \Rightarrow B' \xrightarrow{\tau}_{\mathbb{D}} B'[x' := L]}\ [\textsc{NCaseR}]$$

$$\frac{}{\textbf{fst Pair}\ L\ L' \xrightarrow{\tau}_{\mathbb{D}} L}\ [\textsc{NProj1}] \qquad \frac{}{\textbf{snd Pair}\ L\ L' \xrightarrow{\tau}_{\mathbb{D}} L'}\ [\textsc{NProj2}]$$

$$\frac{B \xrightarrow{\tau}_{\mathbb{D}(\mathsf{p})} B'}{\mathsf{p}[B] \xrightarrow{\tau_{\mathsf{p}}}_{\mathbb{D}} \mathsf{p}[B']}\ [\textsc{NPro}] \qquad \frac{\mathcal{N} \xrightarrow{\tau_{\mathsf{p}}}_{\mathbb{D}} \mathcal{N}''}{\mathcal{N} \mid \mathcal{N}' \xrightarrow{\tau_{\mathsf{p}}}_{\mathbb{D}} \mathcal{N}'' \mid \mathcal{N}'}\ [\textsc{NPar}]$$

$$\frac{D(f(\vec{\mathsf{p}'})) = B}{f(\vec{\mathsf{p}}) \xrightarrow{\tau}_{\mathbb{D}} B[\vec{\mathsf{p}'} := \vec{\mathsf{p}}]}\ [\textsc{NFun}] \qquad \frac{B \rightsquigarrow^* B'' \quad B'' \xrightarrow{\mu} B'}{B \xrightarrow{\mu}_{\mathbb{D}} B'}\ [\textsc{NStr}]$$

Figure 10 Semantics of networks.

$$\frac{}{((\lambda x.B)\ B')\ B'' \rightsquigarrow (\lambda x.B\ B'')\ B'}\ [\text{LR-AbsR}]$$

$$\frac{\mathrm{pn}(B'') = \varnothing}{B''\ ((\lambda x.B)\ B') \rightsquigarrow (\lambda x.B''\ B)\ B'}\ [\text{LR-AbsL}]$$

$$\frac{}{\begin{array}{l}(\textsf{case}\ B\ \textsf{of Inl}\ x \Rightarrow B_1;\ \textsf{Inr}\ x \Rightarrow B_2)\ B' \rightsquigarrow \\ \qquad\qquad \textsf{case}\ B\ \textsf{of Inl}\ x \Rightarrow (B_1\ B');\ \textsf{Inr}\ x \Rightarrow (B_2\ B')\end{array}}\ [\text{LR-CaseR}]$$

$$\frac{\mathrm{pn}(B') = \varnothing}{\begin{array}{l}B'\ (\textsf{case}\ B\ \textsf{of Inl}\ x \Rightarrow B_1;\ \textsf{Inr}\ x \Rightarrow B_2) \rightsquigarrow \\ \qquad\qquad \textsf{case}\ B\ \textsf{of Inl}\ x \Rightarrow (B'\ B_1);\ \textsf{Inr}\ x \Rightarrow (B'\ B_2)\end{array}}\ [\text{LR-CaseL}]$$

$$\frac{\mathrm{pn}(B) = \varnothing}{B\ (\&_{\mathsf{p}}\{l_1 : B_1, \ldots, l_n : B_n\}) \rightsquigarrow \&_{\mathsf{p}}\{l_1 : B\ B_1, \ldots, l_n : B\ B_n\}}\ [\text{LR-OffL}]$$

$$\frac{}{(\&_{\mathsf{p}}\{l_1 : B_1, \ldots, l_n : B_n\})\ B \rightsquigarrow \&_{\mathsf{p}}\{l_1 : B_1\ B, \ldots, l_n : B_n\ B\}}\ [\text{LR-OffR}]$$

$$\frac{\mathrm{pn}(B') = \varnothing}{B'\ (\oplus_{\mathsf{p}}\ l\ B) \rightsquigarrow \oplus_{\mathsf{p}}\ l\ (B'\ B)}\ [\text{LR-ChoL}] \qquad \frac{}{(\oplus_{\mathsf{p}}\ l\ B)\ B' \rightsquigarrow \oplus_{\mathsf{p}}\ l\ (B\ B')}\ [\text{LR-ChoR}]$$

$$\frac{}{\bot\ \bot \rightsquigarrow \bot}\ [\text{LR-BotM}] \qquad \frac{y \text{ fresh for } B)}{\lambda x : T.B \rightsquigarrow \lambda y : T.B[x := y]}\ [\text{LR-Alph}]$$

Figure 11 Rewriting of processes.

$$\frac{\Sigma; \Gamma \vdash B : T}{\Sigma; \Gamma \vdash \oplus_{\mathsf{p}}\ \ell\ B : T}\ [\text{NTChor}] \qquad \frac{\Sigma; \Gamma \vdash B_i : T \text{ for } 1 \leqslant i \leqslant n}{\Sigma; \Gamma \vdash \&_{\mathsf{p}}\{\ell_1 : B_1, \ldots \ell_n : B_n\} : T}\ [\text{NTOff}]$$

$$\frac{}{\Sigma; \Gamma \vdash \textsf{send}_{\mathsf{p}} : T \to \bot}\ [\text{NTSend}] \qquad \frac{}{\Sigma; \Gamma \vdash \textsf{recv}_{\mathsf{p}} : \bot \to T}\ [\text{NTRecv}]$$

$$\frac{\Sigma; \Gamma, x : T \vdash B : T'}{\Sigma; \Gamma \vdash \lambda x : T.B : T \to T'}\ [\text{NTAbs}] \qquad \frac{x : T \in \Gamma}{\Sigma; \Gamma \vdash x : T}\ [\text{NTVar}]$$

$$\frac{\Sigma; \Gamma \vdash B : T \to T' \quad \Sigma; \Gamma \vdash B : T}{\Sigma; \Gamma \vdash B\ B' : T'}\ [\text{NTApp}] \qquad \frac{\Sigma; \Gamma \vdash B : \bot \quad \Sigma; \Gamma \vdash B' : \bot}{\Sigma; \Gamma \vdash B\ B' : \bot}\ [\text{NTApp2}]$$

$$\frac{\Sigma; \Gamma \vdash B : T_1 + T_2 \quad \Sigma; \Gamma, x : T_1 \vdash B' : T \quad \Sigma; \Gamma, x' : T_2 \vdash B'' : T}{\Sigma; \Gamma \vdash \textsf{case}\ B\ \textsf{of Inl}\ x \Rightarrow B';\ \textsf{Inr}\ x' \Rightarrow B'' : T}\ [\text{NTCase}]$$

$$\frac{f : T \in \Gamma}{\Sigma; \Gamma \vdash f : T}\ [\text{NTDef}] \qquad \frac{}{\Sigma; \Gamma \vdash () : ()}\ [\text{NTUnit}] \qquad \frac{}{\Sigma; \Gamma \vdash \bot : \bot}\ [\text{NTBotm}]$$

$$\frac{}{\Sigma; \Gamma \vdash \textsf{Pair} : T \to T' \to (T \times T')}\ [\text{NTPair}]$$

$$\frac{}{\Sigma; \Gamma \vdash \textsf{fst} : (T \times T') \to T}\ [\text{NTProj1}] \qquad \frac{}{\Sigma; \Gamma \vdash \textsf{snd} : (T \times T') \to T'}\ [\text{NTProj2}]$$

$$\frac{\Sigma; \Gamma \vdash B : T' \quad \{T = T', T' = T\} \cap \Sigma \neq \varnothing}{\Sigma; \Gamma \vdash B : T}\ [\text{NTEq}]$$

$$\frac{\forall f \in \mathsf{dom}(\mathbb{D}) \quad f : T \in \Gamma \quad \Sigma; \Gamma \vdash \mathbb{D}(f) : T}{\Sigma; \Gamma \vdash \mathbb{D}}\ [\text{NTDefs}]$$

Figure 12 Typing rules for behaviours.

Choreographies:

$$[\![M\ N]\!]_{\mathsf{p}} = \begin{cases} [\![M]\!]_{\mathsf{p}}\ [\![N]\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(M)) \text{ or } \mathsf{p} \in \mathrm{pn}(M) \cap \mathrm{pn}(N) \\ \bot & \text{if } [\![M]\!]_{\mathsf{p}} = [\![N]\!]_{\mathsf{p}} = \bot \\ [\![M]\!]_{\mathsf{p}} & \text{if } [\![N]\!]_{\mathsf{p}} = \bot \\ [\![N]\!]_{\mathsf{p}} & \text{otherwise} \end{cases}$$

$$[\![\lambda x : T.M]\!]_{\mathsf{p}} = \begin{cases} \lambda x.\ [\![M]\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(\lambda x : T.M)) \\ \bot & \text{otherwise} \end{cases}$$

$$\left[\!\!\left[\textbf{case } M \textbf{ of Inl } x \Rightarrow N; \textbf{Inr } x' \Rightarrow N'\right]\!\!\right]_{\mathsf{p}} =$$

$$\begin{cases} \textbf{case } [\![M]\!]_{\mathsf{p}} \textbf{ of Inl } x \Rightarrow [\![N]\!]_{\mathsf{p}}; \textbf{Inr } x' \Rightarrow [\![N']\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(M)) \\ [\![M]\!]_{\mathsf{p}} & \text{if } [\![N]\!]_{\mathsf{p}} = [\![N']\!]_{\mathsf{p}} = \bot \\ [\![N]\!]_{\mathsf{p}} \sqcup [\![N']\!]_{\mathsf{p}} & \text{if } [\![M]\!]_{\mathsf{p}} = \bot \\ (\lambda x'' : \bot.\ [\![N]\!]_{\mathsf{p}} \sqcup [\![N']\!]_{\mathsf{p}})\ [\![M]\!]_{\mathsf{p}} & \text{otherwise, for some} \\ & \quad x'' \notin \mathrm{fv}(N) \cup \mathrm{fv}(N') \end{cases}$$

$$\left[\!\!\left[\textbf{select}_{\mathsf{q},\mathsf{q}'}\ \ell\ M\right]\!\!\right]_{\mathsf{p}} = \begin{cases} \oplus_{\mathsf{q}'}\ \ell\ [\![M]\!]_{\mathsf{p}} & \text{if } \mathsf{p} = \mathsf{q} \neq \mathsf{q}' \\ \&_{\mathsf{q}}\{\ell : [\![M]\!]_{\mathsf{p}}\} & \text{if } \mathsf{p} = \mathsf{q}' \neq \mathsf{q} \\ [\![M]\!]_{\mathsf{p}} & \text{otherwise} \end{cases}$$

$$\left[\!\!\left[\textbf{com}_{\mathsf{q},\mathsf{q}'}\right]\!\!\right]_{\mathsf{p}} = \begin{cases} \lambda x.x & \text{if } \mathsf{p} = \mathsf{q} = \mathsf{q}' \\ \textbf{send}_{\mathsf{q}'} & \text{if } \mathsf{p} = \mathsf{q} \neq \mathsf{q}' \\ \textbf{recv}_{\mathsf{q}} & \text{if } \mathsf{p} = \mathsf{q}' \neq \mathsf{q} \\ \bot & \text{otherwise} \end{cases}$$

$$[\![()@\mathsf{q}]\!]_{\mathsf{p}} = \begin{cases} () & \text{if } \mathsf{q} = \mathsf{p} \\ \bot & \text{otherwise} \end{cases} \qquad [\![x]\!]_{\mathsf{p}} = \begin{cases} x & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(x)) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![f(\vec{\mathsf{p}})]\!]_{\mathsf{p}} = \begin{cases} f_i(\mathsf{p}_1, \ldots, \mathsf{p}_{i-1}, \mathsf{p}_{i+1}, \ldots, \mathsf{p}_n) & \text{if } \vec{\mathsf{p}} = \mathsf{p}_1, \ldots, \mathsf{p}_{i-1}, \mathsf{p}, \mathsf{p}_{i+1}, \ldots, \mathsf{p}_n \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\textbf{Pair}\ V\ V']\!]_{\mathsf{p}} = \begin{cases} \textbf{Pair}\ [\![V]\!]_{\mathsf{p}}\ [\![V']\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(V) \times \mathrm{type}(V')) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\textbf{fst}]\!]_{\mathsf{p}} = \begin{cases} \textbf{fst} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(\textbf{fst})) \\ \bot & \text{otherwise} \end{cases} \qquad [\![\textbf{snd}]\!]_{\mathsf{p}} = \begin{cases} \textbf{snd} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(\textbf{snd})) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\textbf{Inl}\ V]\!]_{\mathsf{p}} = \begin{cases} \textbf{Inl}\ [\![V]\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(\textbf{Inl}\ V)) \\ \bot & \text{otherwise} \end{cases} \qquad [\![\textbf{Inr}\ V]\!]_{\mathsf{p}} = \begin{cases} \textbf{Inr}\ [\![V]\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(\textbf{Inr}\ V)) \\ \bot & \text{otherwise} \end{cases}$$

Types:

$$[\![T \to_{\rho} T']\!]_{\mathsf{p}} = \begin{cases} [\![T]\!]_{\mathsf{p}} \to [\![T']\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \rho \cup \mathrm{pn}(T) \cup \mathrm{pn}(T') \\ \bot & \text{otherwise} \end{cases} \qquad [\![()@\mathsf{q}]\!]_{\mathsf{p}} = \begin{cases} () & \text{if } \mathsf{q} = \mathsf{p} \\ \bot & \text{otherwise} \end{cases}$$

$$[\![T \times T']\!]_{\mathsf{p}} = \begin{cases} [\![T]\!]_{\mathsf{p}} \times [\![T']\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(T \times T') \\ \bot & \text{otherwise} \end{cases} \qquad [\![T + T']\!]_{\mathsf{p}} = \begin{cases} [\![T]\!]_{\mathsf{p}} + [\![T']\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(T + T') \\ \bot & \text{otherwise} \end{cases}$$

$$[\![t@\vec{\mathsf{p}}]\!]_{\mathsf{p}} = \begin{cases} t_i & \text{if } \vec{\mathsf{p}} = \mathsf{p}_1, \ldots, \mathsf{p}_{i-1}, \mathsf{p}, \mathsf{p}_{i+1}, \ldots, \mathsf{p}_n \\ \bot & \text{otherwise} \end{cases}$$

Definitions:

$$[\![D]\!] = \{f_i(\mathsf{p}_1, \ldots, \mathsf{p}_{i-1}, \mathsf{p}_{i+1}, \ldots, \mathsf{p}_n) \mapsto [\![D(f(\mathsf{p}_1, \ldots, \mathsf{p}_n))]\!]_{\mathsf{p}_i} \mid f(\mathsf{p}_1, \ldots, \mathsf{p}_n) \in \mathsf{dom}(D)\}$$

Figure 13 Projecting Chorλ onto a process.

A.1 Proof of Theorem 25

Proof of Lemma 5. Straightforward from the network semantics. ◀

▶ **Lemma 36.** *Given a value V, if $\Theta; \Sigma; \Gamma \vdash V : T$ and $\llbracket V \rrbracket$ is defined then for any process p in $\mathrm{pn}(V)$, $\llbracket V \rrbracket_{\mathsf{p}} = L$.*

Proof. Straightforward from the projection rules. ◀

▶ **Lemma 37.** *Given a type T, for any process $\mathsf{p} \notin \mathrm{pn}(T)$, $\llbracket T \rrbracket_{\mathsf{p}} = \bot$.*

Proof. Straightforward from induction on T. ◀

▶ **Lemma 38.** *Given a value V, for any process $\mathsf{p} \notin \mathrm{pn}(\mathrm{type}(V))$, if $\llbracket V \rrbracket_{\mathsf{p}}$ is defined then $\llbracket V \rrbracket_{\mathsf{p}} = \bot$.*

Proof. Follows from Lemmas 36 and 37 and the projection rules. ◀

▶ **Lemma 39.** *If $M \rightsquigarrow M'$ and $M \xrightarrow{\tau,\mathsf{P}}_D M''$ and $\llbracket M \rrbracket$ is defined then $M' \xrightarrow{\tau,\mathsf{P}}_D M'''$ such that $M'' \rightsquigarrow^* M'''$*

Proof. Follows from case analysis on $M \rightsquigarrow M'$. ◀

▶ **Lemma 40.** *If $M \rightsquigarrow M'$ then for any process p, $\llbracket M \rrbracket_{\mathsf{p}} \rightsquigarrow \cup \xrightarrow{\tau}^* B$ such that $B \equiv \llbracket M' \rrbracket_{\mathsf{p}}$*

Proof. Follows from case analysis on $M \rightsquigarrow M'$. ◀

Proof of Theorem 25. We prove this by structural induction on $M \xrightarrow{\tau,\mathsf{P}}_D M'$.

- Assume $M = \lambda x : T.N\ V$ and $M' = N[x := V]$. Then for any process $\mathsf{p} \in \mathrm{pn}(\mathrm{type}(\lambda x : T.N))$, we have $\llbracket M \rrbracket_{\mathsf{p}} = (\lambda x : \llbracket T \rrbracket_{\mathsf{p}} . \llbracket N \rrbracket_{\mathsf{p}})\ \llbracket V \rrbracket_{\mathsf{p}}$ and $\llbracket M' \rrbracket_{\mathsf{p}} = \llbracket N \rrbracket_{\mathsf{p}}[x := \llbracket V \rrbracket_{\mathsf{p}}]$, and for any $\mathsf{p}' \notin \mathrm{pn}(\mathrm{type}(\lambda x : T.N))$, we have $\mathsf{p}' \notin \mathrm{pn}(\mathrm{type}(V))$ and therefore $\llbracket M \rrbracket_{\mathsf{p}'} = \llbracket M' \rrbracket_{\mathsf{p}'} = \bot$. We therefore get $\mathsf{p}[\llbracket M \rrbracket_{\mathsf{p}}] \xrightarrow{\tau}_{\llbracket D \rrbracket} \llbracket M' \rrbracket_{\mathsf{p}}$ for all $\mathsf{p} \in \mathrm{pn}(\mathrm{type}(\lambda x : T.N)$ and define $\mathcal{N} = \prod_{\mathsf{p} \in \mathrm{pn}(\mathrm{type}(\lambda x : T.N))} \mathsf{p}[\llbracket M' \rrbracket_{\mathsf{p}}] \mid \prod_{\mathsf{p}' \notin \mathrm{pn}(\mathrm{type}((\lambda x : T.N))} \mathsf{p}'[\bot]$ and the result follows.
- Assume $M = N\ M''$, $M' = N'\ M''$, and $N \xrightarrow{\tau,\mathsf{P}}_D N'$. Then for any process $\mathsf{p} \in \mathrm{pn}(\mathrm{type}(N))$, $\llbracket M \rrbracket_{\mathsf{p}} = \llbracket N \rrbracket_{\mathsf{p}}\ \llbracket M'' \rrbracket_{\mathsf{p}}$ and $\llbracket M' \rrbracket_{\mathsf{p}} = \llbracket N' \rrbracket_{\mathsf{p}}\ \llbracket M'' \rrbracket_{\mathsf{p}}$. For any process p' such that $\llbracket N \rrbracket_{\mathsf{p}'} = \llbracket M'' \rrbracket_{\mathsf{p}'} = \bot$, by induction we have $\llbracket N' \rrbracket_{\mathsf{p}'} = \bot$, and therefore $\llbracket M \rrbracket_{\mathsf{p}'} = \llbracket M' \rrbracket_{\mathsf{p}'} = \bot$. For any other process p'' such that $\llbracket N \rrbracket_{\mathsf{p}''} = \bot$, by induction we get $\llbracket N' \rrbracket_{\mathsf{p}''} = \bot$ and therefore $\llbracket M \rrbracket_{\mathsf{p}''} = \llbracket M' \rrbracket_{\mathsf{p}''} = \llbracket M'' \rrbracket_{\mathsf{p}''}$. For any other process p''' such that $\llbracket M'' \rrbracket_{\mathsf{p}'''} = \bot$, we get $\llbracket M \rrbracket_{\mathsf{p}'''} = \llbracket N \rrbracket_{\mathsf{p}'''}$ and $\llbracket M' \rrbracket_{\mathsf{p}'''} = \llbracket N' \rrbracket_{\mathsf{p}'''}$. And by induction $\llbracket N \rrbracket \rightarrow^*_{\llbracket D \rrbracket} \mathcal{N}_N$ and $N' \rightarrow^*_{\llbracket D \rrbracket} N''$ for $\mathcal{N}_N \sqsupseteq \llbracket N'' \rrbracket$. For any process p we therefore get $\llbracket N \rrbracket_{\mathsf{p}} \xrightarrow{\mu_0}_{\llbracket D \rrbracket} \xrightarrow{\mu_1}_{\llbracket D \rrbracket} \dots B_{\mathsf{p}}$ for $B_{\mathsf{p}} \sqsupseteq \llbracket N'' \rrbracket_{\mathsf{p}}$ for some sequences of transitions $\xrightarrow{\mu_0}_{\llbracket D \rrbracket} \xrightarrow{\mu_1}_{\llbracket D \rrbracket} \dots$, and from the network semantics we get

$$\llbracket M \rrbracket \rightarrow^* \begin{array}{l} \prod_{\mathsf{p} \in \mathrm{pn}(\mathrm{type}(N)) \cup (\mathrm{pn}(N) \cap \mathrm{pn}(M''))} \mathsf{p}[B_{\mathsf{p}}\ \llbracket M'' \rrbracket_{\mathsf{p}}] \mid \prod_{\llbracket N \rrbracket_{\mathsf{p}'} = \llbracket M'' \rrbracket_{\mathsf{p}'} = \bot} \mathsf{p}'[\bot] \\ \mid \prod_{\llbracket M \rrbracket_{\mathsf{p}''} = \llbracket M'' \rrbracket_{\mathsf{p}''}} \mathsf{p}''[\llbracket M'' \rrbracket_{\mathsf{p}''}] \mid \prod_{\llbracket M \rrbracket_{\mathsf{p}'''} = \llbracket N \rrbracket_{\mathsf{p}'''}} \mathsf{p}''[B_{\mathsf{p}''}] \end{array} = \mathcal{N}$$

  and $M' \rightarrow^* N''\ M$. And since $\llbracket N \rrbracket \rightarrow^*_{\llbracket D \rrbracket} \mathcal{N}'$ and $\llbracket N' \rrbracket \rightarrow^*_{\llbracket D \rrbracket} \mathcal{N}'_N$, we know these sequences of transitions can synchronise when necessary, and if $\llbracket N \rrbracket_{\mathsf{p}''''} \neq \llbracket N' \rrbracket_{\mathsf{p}''''} = \bot$ then we can do the extra application to get rid of this unit.

- Assume $M = V\ N$, $M' = V\ N'$, and $N \xrightarrow{\tau,\mathsf{P}}_D N'$. Then for any process $\mathsf{p} \in \mathrm{pn}(\mathrm{type}(V))$, $[\![M]\!]_{\mathsf{p}} = [\![V]\!]_{\mathsf{p}}\ [\![N]\!]_{\mathsf{p}}$ and $[\![M']\!]_{\mathsf{p}} = [\![V]\!]_{\mathsf{p}}\ [\![N']\!]_{\mathsf{p}}$. Since V is a value, for any process $\mathsf{p}' \notin \mathrm{pn}(\mathrm{type}(V))$, we have $[\![V]\!]_{\mathsf{p}'} = \bot$ and so for any process p' such that $[\![V]\!]_{\mathsf{p}'} = [\![N]\!]_{\mathsf{p}'} = \bot$, by induction we get $[\![N']\!]_{\mathsf{p}'} = \bot$ and therefore $[\![M]\!]_{\mathsf{p}'} = [\![M']\!]_{\mathsf{p}'} = \bot$. For any other process p'' such that $[\![V]\!]_{\mathsf{p}''} = \bot$, we have $[\![M]\!]_{\mathsf{p}''} = [\![N]\!]_{\mathsf{p}''}$ and $[\![M']\!]_{\mathsf{p}''} = [\![N']\!]_{\mathsf{p}''}$. By induction, $[\![N]\!] \rightarrow^*_{[\![D]\!]} \mathcal{N}_N$ and $N' \rightarrow^*_{[\![D]\!]} N''$ for $\mathcal{N}_N \sqsupseteq [\![N'']\!]$. For any process p we therefore get $[\![N]\!]_{\mathsf{p}} \xrightarrow{\mu_0}_{[\![D]\!](\mathsf{p})} \xrightarrow{\mu_1}_{[\![D]\!](\mathsf{p})} \dots B_{\mathsf{p}}$ for $B_{\mathsf{p}} \sqsupseteq [\![N'']\!]_{\mathsf{p}}$ for some sequences of transitions $\xrightarrow{\mu_0}_{[\![D]\!](\mathsf{p})} \xrightarrow{\mu_1}_{[\![D]\!](\mathsf{p})} \dots$ and from the network semantics we get

$$[\![M]\!] \rightarrow^* \prod_{\mathsf{p} \in \mathrm{pn}(\mathrm{type}(N))} \mathsf{p}[[\![V]\!]_{\mathsf{p}}\ B_{\mathsf{p}}] \mid \prod_{\mathsf{p}' \notin \mathrm{pn}(\mathrm{type}(N))} \mathsf{p}'[B_{\mathsf{p}'}] = \mathcal{N}$$

and

$$M' \rightarrow^* V\ N''$$

and the result follows.
- Assume $M = M''\ N$, $M' = M''\ N'$, $N \xrightarrow{\tau,\mathsf{P}} N'$, and $\mathrm{pn}(M) \cap \mathsf{P} = \varnothing$. Then for any $\mathsf{p} \in \mathsf{P}$, $\mathrm{pn}([\![M'']\!]_{\mathsf{p}}) \cap \mathsf{P} = \varnothing$ and the result follows from induction and using rule NApp3.
- Assume $M = \textsf{case}\ N\ \textsf{of Inl}\ x \Rightarrow N';\ \textsf{Inr}\ x' \Rightarrow N''$, $M' = \textsf{case}\ M''\ \textsf{of Inl}\ x \Rightarrow N';\ \textsf{Inr}\ x \Rightarrow N''$, and $N \xrightarrow{\tau,\mathsf{P}}_D M''$. Then for any process p such that $\mathsf{p} \in \mathrm{pn}(\mathrm{type}(N))$, we have projections $[\![M]\!]_{\mathsf{p}} = \textsf{case}\ [\![N]\!]_{\mathsf{p}}\ \textsf{of Inl}\ x \Rightarrow [\![N']\!]_{\mathsf{p}};\ \textsf{Inr}\ x' \Rightarrow [\![N'']\!]_{\mathsf{p}}$ and $[\![M']\!]_{\mathsf{p}} = \textsf{case}\ [\![M'']\!]_{\mathsf{p}}\ \textsf{of Inl}\ x \Rightarrow [\![N']\!]_{\mathsf{p}};\ \textsf{Inr}\ x' \Rightarrow [\![N'']\!]_{\mathsf{p}}$. For any other process p' such that $[\![N]\!]_{\mathsf{p}'} = [\![N']\!]_{\mathsf{p}'} = [\![N'']\!]_{\mathsf{p}'} = \bot$, by induction we get $[\![M'']\!]_{\mathsf{p}'} = \bot$, and therefore $[\![M]\!]_{\mathsf{p}'} = [\![M']\!]_{\mathsf{p}'} = \bot$. For any other process p'' such that $[\![N]\!]_{\mathsf{p}''} = \bot$, we get $[\![M]\!]_{\mathsf{p}''} = [\![M']\!]_{\mathsf{p}''} = [\![N']\!]_{\mathsf{p}''} \sqcup [\![N'']\!]_{\mathsf{p}''}$. For any other processes p''' such that $[\![N']\!]_{\mathsf{p}'''} = [\![N'']\!]_{\mathsf{p}'''} = \bot$, we have $[\![M]\!]_{\mathsf{p}'''} = [\![N]\!]_{\mathsf{p}'''}$ and $[\![M']\!]_{\mathsf{p}'''} = [\![M'']\!]_{\mathsf{p}'''}$. For any other process p'''', we have $[\![M]\!]_{\mathsf{p}''''} = (\lambda x : \bot.[\![N']\!]_{\mathsf{p}''''} \sqcup [\![N'']\!]_{\mathsf{p}''''})\ [\![N]\!]_{\mathsf{p}''''}$ and $[\![M']\!]_{\mathsf{p}''''} = (\lambda x.[\![N']\!]_{\mathsf{p}''''} \sqcup [\![N'']\!]_{\mathsf{p}''''})\ [\![M'']\!]_{\mathsf{p}''''}$ for $x \notin \mathrm{fv}(N') \cup \mathrm{fv}(N'')$. The rest follows by simple induction similar to the second case.
- Assume $M = \textsf{case}\ N\ \textsf{of Inl}\ x \Rightarrow N_1;\ \textsf{Inr}\ x' \Rightarrow N_2$, $M' = \textsf{case}\ N\ \textsf{of Inl}\ x \Rightarrow N_1';\ \textsf{Inr}\ x' \Rightarrow N_2'$, $N_1 \xrightarrow{\tau,\mathsf{P}}_D N_1'$, $N_1 \xrightarrow{\tau,\mathsf{P}}_D N_2$, and $\mathsf{P} \cap \mathrm{pn}(N) = \varnothing$. Then for any process p such that $\mathsf{p} \in \mathrm{pn}(\mathrm{type}(N))$, we have $[\![M]\!]_{\mathsf{p}} = \textsf{case}\ [\![N]\!]_{\mathsf{p}}\ \textsf{of Inl}\ x \Rightarrow [\![N']\!]_{\mathsf{p}};\ \textsf{Inr}\ x' \Rightarrow [\![N'']\!]_{\mathsf{p}}$ For any other process p' such that $[\![N]\!]_{\mathsf{p}'} = [\![N_1]\!]_{\mathsf{p}'} = [\![N_2]\!]_{\mathsf{p}'} = \bot$, by induction we get $[\![N_1']\!]_{\mathsf{p}'} = [\![N_2']\!]_{\mathsf{p}'} = \bot$, and therefore $[\![M]\!]_{\mathsf{p}'} = [\![M']\!]_{\mathsf{p}'} = \bot$. For any other process p'' such that $[\![N]\!]_{\mathsf{p}''} = \bot$, we get $[\![M]\!]_{\mathsf{p}''} = [\![N_1]\!]_{\mathsf{p}''} \sqcup [\![N_2]\!]_{\mathsf{p}''}$. For any other processes p''' such that $[\![N_1]\!]_{\mathsf{p}'''} = [\![N_2]\!]_{\mathsf{p}'''} = \bot$, we have $[\![M]\!]_{\mathsf{p}'''} = [\![N]\!]_{\mathsf{p}'''}$. For any other process p'''', we have $[\![M]\!]_{\mathsf{p}''''} = (\lambda x : \bot.[\![N_1]\!]_{\mathsf{p}''''} \sqcup [\![N_2]\!]_{\mathsf{p}''''})\ [\![N]\!]_{\mathsf{p}''''}$. If $[\![N_1']\!]_{\mathsf{p}} \sqcup [\![N_2']\!]_{\mathsf{p}}$ is defined for all p then the result follows from induction. Otherwise we have M_1 and M_2 such that $N_1' \xrightarrow{\tau,\mathsf{P}}_D M_1$ and $N_2' \rightarrow \tau, \mathsf{P}_D M_2$ and $[\![M_1]\!]_{\mathsf{p}} \sqcup [\![M_2]\!]_{\mathsf{p}}$ for all p, and the result follows from induction on these transitions.
- Assume $M = \textsf{case Inl}\ V\ \textsf{of Inl}\ x \Rightarrow N;\ \textsf{Inr}\ x' \Rightarrow N'$ and $M' = N[x := V]$. Then for any process $\mathsf{p} \in \mathrm{pn}(\mathrm{type}(\textsf{Inl}\ V))$, we have $[\![M]\!]_{\mathsf{p}} = \textsf{case Inl}\ [\![V]\!]_{\mathsf{p}}\ \textsf{of Inl}\ x \Rightarrow [\![N]\!]_{\mathsf{p}};\ \textsf{Inr}\ x' \Rightarrow [\![N']\!]_{\mathsf{p}}$ and $[\![M']\!]_{\mathsf{p}} = [\![N[x := [\![V]\!]_{\mathsf{p}}]]\!]_{\mathsf{p}}$. By Lemma 38, $[\![N[x := [\![V]\!]_{\mathsf{p}}]]\!]_{\mathsf{p}} = [\![N]\!]_{\mathsf{p}}[x := [\![V]\!]_{\mathsf{p}}]$. For any other process $\mathsf{p}' \notin \mathrm{pn}(\mathrm{type}(\textsf{Inl}\ V))$, $[\![\textsf{Inl}\ V]\!]_{\mathsf{p}'} = \bot$, and therefore $[\![M]\!]_{\mathsf{p}'} = [\![N]\!]_{\mathsf{p}'} \sqcup [\![N']\!]_{\mathsf{p}'} \sqsupset [\![N]\!]_{\mathsf{p}'} = [\![M']\!]_{\mathsf{p}'}$. The result follows.
- Assume $M = \textsf{case Inr}\ V\ \textsf{of Inl}\ x \Rightarrow N;\ \textsf{Inr}\ x' \Rightarrow N'$ and $M' = N'[x' := V]$. This case is similar to the previous.
- Assume $M = \textsf{case}\ N\ \textsf{of Inl}\ x \Rightarrow N_1;\ \textsf{Inr}\ x' \Rightarrow N_2$, $M' = \textsf{case}\ N\ \textsf{of Inl}\ x \Rightarrow N_1';\ \textsf{Inr}\ x' \Rightarrow N_2'$, $N_1 \xrightarrow{\mathsf{P}}_D N_1'$, $N_2 \xrightarrow{\mathsf{P}} N_2'$, and $\mathsf{P} \cap \mathrm{pn}(N) = \varnothing$. This case is similar to case four.

- Assume $M = \mathbf{com}_{\mathsf{q},\mathsf{p}} V$ and $M' = V[\mathsf{q} := \mathsf{p}]$ and $\mathrm{fv}(V) = \varnothing$. Then if $\mathsf{q} \neq \mathsf{p}$, $[\![M]\!]_{\mathsf{p}} = \mathbf{recv}_{\mathsf{q}}\ \bot$, $[\![M']\!]_{\mathsf{p}} = [\![V[\mathsf{q} := \mathsf{p}]]\!]_{\mathsf{p}} = [\![V]\!]_{\mathsf{p}}[\mathsf{q} := \mathsf{p}]$ since $\mathrm{pn}(\mathrm{type}(V)) = \mathsf{q}$, $[\![M]\!]_{\mathsf{q}} = \mathbf{send}_{\mathsf{p}}\ [\![V]\!]_{\mathsf{q}}$, $[\![M']\!]_{\mathsf{q}} = \bot$, and for any $\mathsf{p}' \notin \{\mathsf{q},\mathsf{p}\}$, $[\![M]\!]_{\mathsf{p}'} = [\![M']\!]_{\mathsf{p}'} = \bot$. We therefore get $[\![M]\!]_{\mathsf{p}} \xrightarrow{\mathbf{recv}_{\mathsf{q}}[\![V]\!]_{\mathsf{q}}[\mathsf{q}:=\mathsf{p}]}_{[\![D]\!]} [\![M']\!]_{\mathsf{p}}$, $[\![M]\!]_{\mathsf{q}} \xrightarrow{\mathbf{send}_{\mathsf{p}}[\![V]\!]_{\mathsf{q}}}_{[\![D]\!]} [\![M']\!]_{\mathsf{q}}$, and $[\![M]\!]_{\mathsf{p}'} = [\![M']\!]_{\mathsf{p}'}$. We define $\mathcal{N} = \mathcal{N}' = [\![M']\!]$ and the result follows. If $\mathsf{q} = \mathsf{p}$, then $[\![M]\!]_{\mathsf{p}} = (\lambda x.x)\ [\![V]\!]_{\mathsf{p}}$ and $[\![M']\!]_{\mathsf{p}} = [\![V]\!]_{\mathsf{p}}$ and $\mathcal{N} = \mathcal{N}' = [\![M']\!]$ and the result follows.
- Assume $M = \mathbf{select}_{\mathsf{q},\mathsf{p}}\ l\ M'$. Then $[\![M]\!]_{\mathsf{q}} = \oplus_{\mathsf{p}}\ l\ [\![M']\!]_{\mathsf{q}}$, $[\![M]\!]_{\mathsf{p}} = \&\{l : [\![M']\!]_{\mathsf{p}}\}$, and for any $\mathsf{p}' \notin \{\mathsf{q},\mathsf{p}\}$, $[\![M]\!]_{\mathsf{p}'} = [\![M']\!]_{\mathsf{p}'}$. We therefore get $[\![M]\!] \xrightarrow{\tau_{\mathsf{p},\mathsf{q}}}_{[\![D]\!]} [\![M]\!] \backslash \{\mathsf{p},\mathsf{q}\} \mid \mathsf{p}[[\![M']\!]_{\mathsf{p}}] \mid \mathsf{q}[[\![M']\!]_{\mathsf{q}}]$ and the result follows.
- Assume $M = \mathbf{select}_{\mathsf{q},\mathsf{p}}\ l\ N$, $M' = \mathbf{select}_{\mathsf{q},\mathsf{p}}\ l\ N'$, $N \xrightarrow{\tau,\mathsf{P}}_{D} N'$, and $\mathsf{P} \cap \{\mathsf{q},\mathsf{p}\} = \varnothing$. Then $[\![M]\!]_{\mathsf{q}} = \oplus_{\mathsf{p}}\ l\ [\![N]\!]_{\mathsf{q}}$, $[\![M']\!]_{\mathsf{q}} = \oplus_{\mathsf{p}}\ l\ [\![N']\!]_{\mathsf{q}}$, $[\![M]\!]_{\mathsf{p}} = \&\{l : [\![N]\!]_{\mathsf{p}}\}$, $[\![M']\!]_{\mathsf{p}} = \&\{l : [\![N']\!]_{\mathsf{p}}\}$, and for any $\mathsf{p}' \notin \{\mathsf{q},\mathsf{p}\}$, $[\![M]\!]_{\mathsf{p}'} = [\![N]\!]_{\mathsf{p}'}$ and $[\![M']\!]_{\mathsf{p}'} = [\![N']\!]_{\mathsf{p}'}$. The result follows from induction and using rules NOFF2 and NCHO2.
- Assume $M = \mathbf{fst}\ \mathbf{Pair}\ V\ V'$ and $M' = V$. Then for any process $\mathsf{p} \in \mathrm{pn}(\mathrm{type}(\mathbf{Pair}\ M'\ V'))$, $[\![M]\!]_{\mathsf{p}} = \mathbf{fst}\ \mathbf{Pair}\ [\![M']\!]_{\mathsf{p}}\ [\![V']\!]_{\mathsf{p}}$ and for any other process $\mathsf{p}' \notin \mathrm{pn}(\mathrm{type}(\mathbf{Pair}\ M'\ V')$, we have $[\![M]\!]_{\mathsf{p}'} = \bot$ and $[\![M']\!]_{\mathsf{p}'} = \bot$. We define $\mathcal{N} = \mathcal{N}' = [\![M']\!]$ and the result follows.
- Assume $M = \mathbf{snd}\ \mathbf{Pair}\ V\ V'$ and $M' = V'$. Then the case is similar to the previous.
- Assume $M = f(\vec{\mathsf{p}})$ and $M' = D(f(\vec{\mathsf{p}'}))[\vec{\mathsf{p}'} := \vec{\mathsf{p}}]$. Then the result follows from the definition of $[\![D]\!]$.
- Assume there exists N such that $M \rightsquigarrow N$ and $N \xrightarrow{\tau,\mathsf{P}}_{D} M'$. Then the result follows from induction and Lemma 40. ◀

A.2 Proof of Theorem 26

▶ **Definition 41.** *Given a network* $\mathcal{N} = \prod_{\mathsf{p}\in\rho} \mathsf{p}[B_{\mathsf{p}}]$, *we have* $\mathcal{N}\backslash\rho' = \prod_{\mathsf{p}\in(\rho\backslash\rho')} \mathsf{p}[B_{\mathsf{p}}]$

▶ **Lemma 42.** *For any process* p *and network* $\mathcal{N}$, *if* $\mathcal{N} \xrightarrow{\tau_{\mathsf{P}}} \mathcal{N}'$ *and* $\mathsf{p} \notin \mathsf{P}$ *then* $\mathcal{N}(\mathsf{p}) = \mathcal{N}'(\mathsf{p})$.

Proof. Straightforward from the network semantics. ◀

▶ **Lemma 43.** *For any set of processes* P *and network* $\mathcal{N}$, *if* $\mathcal{N} \xrightarrow{\tau_{\mathsf{P}'}} \mathcal{N}'$ *and* $\mathsf{P} \cap \mathsf{P}' = \varnothing$ *then* $\mathcal{N}\backslash\mathsf{P} \xrightarrow{\tau_{\mathsf{P}'}} \mathcal{N}'\backslash\mathsf{P}$.

Proof. Straightforward from the network semantics. ◀

▶ **Lemma 44.** *If* $[\![M]\!]_{\mathsf{p}} \rightsquigarrow B$ *then there exists* M' *such that* $M \rightsquigarrow M'$ *and* $B \equiv [\![M']\!]_{\mathsf{p}}$

Proof. Follows from case analysis on $[\![M]\!]_{\mathsf{p}} \rightsquigarrow B$ keeping in mind that $[\![M]\!]_{\mathsf{p}}$ cannot be $\bot\ \bot$. ◀

Proof of Theorem 26. If $[\![M]\!] \to^{*}_{[\![D]\!]} \mathcal{N}$ uses rule NSTR then this follows from Lemma 44. Otherwise we prove this by structural induction on M.

- Assume $M = N_1\ N_2$. Then for any process $\mathsf{p} \in \mathrm{pn}(\mathrm{type}(N_1)) \cup (\mathrm{pn}(N_1) \cap \mathrm{pn}(N_2)$, $[\![M]\!]_{\mathsf{p}} = [\![N_1]\!]_{\mathsf{p}}\ [\![N_2]\!]_{\mathsf{p}}$, for any process p' such that $[\![N_1]\!]_{\mathsf{p}'} = [\![N_2]\!]_{\mathsf{p}'} = \bot$, we have $[\![M]\!]_{\mathsf{p}'} = \bot$. For any other process p'' such that $[\![N_1]\!]_{\mathsf{p}''} = \bot$, $[\![M]\!]_{\mathsf{p}''} = [\![N_2]\!]_{\mathsf{p}''}$. For any other process p''' such that $[\![N_2]\!]_{=} \bot$, we get $[\![M]\!]_{\mathsf{p}'''} = [\![N_1]\!]_{\mathsf{p}'''}$. We then have 2 cases.
  - Assume $N_2 = V$. Then $[\![N_2]\!]_{\mathsf{p}} = L$ by Lemma 36, and for any p' such that $\mathsf{p}' \notin \mathrm{pn}(\mathrm{type}(N_2)) \subseteq \mathrm{pn}(\mathrm{type}(N_1))$, by Lemma 38, $[\![N_2]\!]_{\mathsf{p}'} = \bot$ and therefore $[\![M]\!]_{\mathsf{p}'} = [\![N_1]\!]_{\mathsf{p}'}$, and we have 5 cases.

  * Assume $N_1 = \lambda x : T.N_3$. Then for any process $\mathsf{p} \in \mathrm{pn}(\mathrm{type}(N_1))$, $[\![N_1]\!]_{\mathsf{p}} = \lambda x : [\![T]\!]_{\mathsf{p}} .[\![N_3]\!]_{\mathsf{p}}$. And for any process $\mathsf{p}' \notin \mathrm{pn}(\mathrm{type}(N_1))$, $[\![N_1]\!]_{\mathsf{p}} = \bot$. We have two cases, using either rule NAbsApp or rules NInAbs and NApp1.
    If we use rule NAbsApp, then there exists p'' such that $\mathsf{P} = \mathsf{p}''$ and $\mathsf{p}'' \in \mathrm{pn}(\mathrm{type}(N_1))$. We then get $[\![M]\!] \xrightarrow{\tau,\mathsf{P}}_{[\![D]\!]} \mathcal{M} = [\![M]\!]\backslash\{\mathsf{p}''\} \mid \mathsf{p}''[[\![N_3]\!]_{\mathsf{p}''}[x := [\![N_2]\!]_{\mathsf{p}''}]]$. Since $\mathcal{M} \to^* [\![N_3[x := N_2]]\!]$ and the remaining transitions in $[\![M]\!] \to^*_{[\![D]\!]} \mathcal{N}$ take place in N_3, the result follows from using rule NAbsApp in every process in $\mathrm{pn}(\mathrm{type}(N_1))$ and induction.
    If we use rules NInAbs and NApp1 then there exists p'' such that $\mathsf{P} = \mathsf{p}''$ and $[\![N_3]\!]_{\mathsf{p}''} \xrightarrow{\mu} B$ and

$$[\![M]\!] \xrightarrow{\mu}_{[\![D]\!]} [\![M]\!]\backslash\{\mathsf{p}''\}) \mid \mathsf{p}''[\lambda x.B\ [\![N_2]\!]_{\mathsf{p}''}] \to^*_{[\![D]\!]} \mathcal{N}$$

    By induction, $N_3 \to^*_D N_3'$ and $([\![N_3]\!]\backslash\{\mathsf{p}''\} \mid \mathsf{p}''[B] \to_{\mathbb{D}} \mathcal{N}''$ such that $[\![N_3]\!] \sqsupseteq \mathcal{N}''$, and we define $M' = \lambda x : T.N_3'\ N_2$ and

$$\mathcal{N}' = (\mathcal{N}\backslash \mathrm{pn}(\mathrm{type}(N_1))) \mid \prod_{\mathsf{p}\in\mathrm{pn}(\mathrm{type}(N_3))} \mathsf{p}[(\lambda x.\mathcal{N}''(\mathsf{p}))\ [\![N_2]\!]_{\mathsf{p}''}]$$

    and the result follows by using rules InAbs, App1, NInAbs, and NApp1 and induction.
  * Assume $N_1 = \mathbf{com}_{\mathsf{q},\mathsf{p}}$. Then if $\mathsf{q} \neq \mathsf{p}$, $[\![M]\!]_{\mathsf{q}} = \mathbf{send}_{\mathsf{p}}\ [\![N_2]\!]_{\mathsf{q}}$, $[\![M]\!]_{\mathsf{p}} = \mathbf{recv}_{\mathsf{p}}\ \bot$, and for $\mathsf{p}' \notin \{\mathsf{q},\mathsf{p}\}$, $[\![N_1]\!]_{\mathsf{p}'} = \bot = [\![M]\!]_{\mathsf{p}'}$, and therefore $\mathsf{P} = \mathsf{q},\mathsf{p}$, and if $\mathsf{q} = \mathsf{p}$ then $[\![N_1]\!]_{\mathsf{p}} = \lambda x.x$.
    If $\mathsf{P} = \mathsf{q},\mathsf{p}$ then $\mathcal{N} = [\![M]\!]\backslash\{\mathsf{q},\mathsf{p}\} \mid \mathsf{q}[\bot] \mid \mathsf{p}[[\![N_2]\!]_{\mathsf{q}}[\mathsf{q} := \mathsf{p}]]$. Because $[\![N_2]\!]_{\mathsf{p}} = \bot$ and $[\![N_2]\!]_{\mathsf{q}} = V$, $N_2 = V$. Therefore $M \xrightarrow{\mathsf{P}}_D V[\mathsf{q} := \mathsf{p}]$ and the result follows.
    If $\mathsf{P} = \mathsf{p}$ then $\mathsf{q} = \mathsf{p}$, $\mathcal{N} = [\![M]\!]\backslash\{\mathsf{p}\} \mid \mathsf{p}[[\![N_2]\!]_{\mathsf{p}}]$ and the rest is similar to above.
  * Assume $N_1 = \mathbf{fst}$. Then $N_2 = \mathbf{Pair}\ V\ V'$ and for any process $\mathsf{p} \in \mathrm{pn}(\mathrm{type}(\mathbf{Pair}\ V\ V'))$, $[\![M]\!]_{\mathsf{p}} = \mathbf{fst}\ \mathbf{Pair}\ [\![V]\!]_{\mathsf{p}}\ [\![V']\!]_{\mathsf{p}}$ and for any other process $\mathsf{p}' \notin \mathrm{pn}(\mathrm{type}(\mathbf{Pair}\ V\ V')$, by Lemma 38 we have $[\![M]\!]_{\mathsf{p}'} = [\![N_1]\!]_{\mathsf{p}'} = \bot$, and therefore $[\![M]\!]_{\mathsf{p}'} \not\to$.
    If $\mathsf{P} = \mathsf{p} \in \mathrm{pn}(\mathrm{type}(\mathbf{Pair}\ V\ V'))$ then $\mathcal{N} = [\![M]\!]\backslash\{\mathsf{p}\} \mid \mathsf{p}[[\![V]\!]_{\mathsf{p}}]$ and $M \xrightarrow{\mathsf{P}}_D V$. The result follows by use of rule NProj1 and Lemma 38.
  * Assume $N_1 = \mathbf{snd}$. This case is similar to the previous.
  * Otherwise, $N_1 \neq V$ and either $[\![M]\!] \xrightarrow{\tau_{\mathsf{p}}}_{[\![D]\!]} \mathcal{M} \to^*_{[\![D]\!]} \mathcal{N}$ or $[\![M]\!] \xrightarrow{\tau_{\mathsf{p},\mathsf{q}}}_{[\![D]\!]} \mathcal{M} \to^*_{[\![D]\!]} \mathcal{N}$.
    If $[\![M]\!] \xrightarrow{\tau_{\mathsf{p}}}_{[\![D]\!]} \mathcal{M} \to^*_{[\![D]\!]} \mathcal{N}$ then either $[\![N_1]\!]_{\mathsf{p}} \xrightarrow{\tau} B$ and $\mathsf{p} \in \mathrm{pn}(\mathrm{type}(N_1))$, $\mathcal{M} = [\![M]\!]\backslash\{\mathsf{p}\} \mid \mathsf{p}[B\ [\![N_2]\!]_{\mathsf{p}}]$. We therefore have $[\![N_1]\!] \xrightarrow{\tau_{\mathsf{p}}} [\![N_1]\!]\backslash\{\mathsf{p}\} \mid \mathsf{p}[B]$, and by induction, $N_1 \to^*_D N_1'$ such that $[\![N_1]\!]\backslash\{\mathsf{p}\} \mid \mathsf{p}[B] \to^* \mathcal{N}_1 \sqsupseteq [\![N_1']\!]$. Since all these transitions can be propagated past N_2 in the network and $[\![N_2]\!]_{\mathsf{p}'}$ in any process p' involved, we get the result for $M' = N_1'\ N_2$.
    If $[\![M]\!] \xrightarrow{\tau_{\mathsf{p},\mathsf{q}}}_{[\![D]\!]} \mathcal{M} \to^*_{[\![D]\!]} \mathcal{N}$ then the case is similar.
- If $N_2 \neq V$ then we have 2 cases.
  * If $[\![M]\!] \xrightarrow{\tau_{\mathsf{p}}}_{[\![D]\!]} \mathcal{M} \to^*_{[\![D]\!]} \mathcal{N}$ then either $[\![N_1]\!]_{\mathsf{p}} \xrightarrow{\tau} B$ or $[\![N_2]\!]_{\mathsf{p}} \xrightarrow{\tau} B$ and the case is similar to the previous.
  * If $[\![M]\!] \xrightarrow{\tau_{\mathsf{p},\mathsf{q}}}_{[\![D]\!]} \mathcal{M} \to^*_{[\![D]\!]} \mathcal{N}$ then there exists L such that either $[\![N_1]\!]_{\mathsf{q}} \xrightarrow{\mathbf{send}_{\mathsf{p}}\ L} B_{\mathsf{q}}$ or $[\![N_2]\!]_{\mathsf{q}} \xrightarrow{\mathbf{send}_{\mathsf{p}}\ L} B_{\mathsf{q}}$ and $[\![N_1]\!]_{\mathsf{p}} \xrightarrow{\mathbf{recv}_{\mathsf{q}}\ L[\mathsf{q}:=\mathsf{p}]} B_{\mathsf{p}}$ or $[\![N_2]\!]_{\mathsf{p}} \xrightarrow{\mathbf{recv}_{\mathsf{q}}\ L[\mathsf{q}:=\mathsf{p}]} B_{\mathsf{p}}$.
    If $[\![N_1]\!]_{\mathsf{q}} \xrightarrow{\mathbf{send}_{\mathsf{p}}\ L} B_{\mathsf{q}}$ then $[\![N_1]\!]_{\mathsf{q}} \neq L'$ and therefore $[\![N_1]\!]_{\mathsf{p}} \xrightarrow{\mathbf{recv}_{\mathsf{q}}\ L[\mathsf{q}:=\mathsf{p}]} B_{\mathsf{p}}$ and the

case is similar to the previous. If $[\![N_2]\!]_\mathsf{q} \xrightarrow{\mathsf{send}_\mathsf{p}\ L} B_\mathsf{q}$ then $[\![N_1]\!]_\mathsf{q} = L'$, and therefore $[\![N_2]\!]_\mathsf{p} \xrightarrow{\mathsf{recv}_\mathsf{q}\ L[\mathsf{q}:=\mathsf{p}]} B_\mathsf{p}$ and the case is similar to the previous.

- Assume $M = \textbf{case}\ N\ \textbf{of Inl}\ x \Rightarrow N';\ \textbf{Inr}\ x' \Rightarrow N''$. Then for any process $\mathsf{p} \in \mathrm{pn}(\mathrm{type}(N))$, $[\![M]\!]_\mathsf{p} = \textbf{case}\ [\![N]\!]_\mathsf{p}\ \textbf{of Inl}\ x \Rightarrow [\![N']\!]_\mathsf{p};\ \textbf{Inr}\ x' \Rightarrow [\![N'']\!]_\mathsf{p}$. And for any other process $\mathsf{p}' \notin \mathrm{pn}(\mathrm{type}(N))$, $[\![M]\!]_{\mathsf{p}'} = (\lambda x.[\![N']\!]_{\mathsf{p}'} \sqcup [\![N'']\!]_{\mathsf{p}'})\ [\![N]\!]_{\mathsf{p}'}$. We know that $[\![M]\!] \xrightarrow{\tau_\mathsf{P}}_{[\![D]\!]} \mathcal{M} \to^*_{[\![D]\!]} \mathcal{N}$ and we have three cases.
  - Assume $\mathsf{P} = \mathsf{p} \in \mathrm{pn}(\mathrm{type}(N))$. Then we have three cases.
    * Assume $N = \textbf{Inl}\ V$. Then $[\![N]\!]_\mathsf{p} = \textbf{Inl}\ [\![V]\!]_\mathsf{p}$ and $\mathcal{M} = [\![M]\!]\backslash\{\mathsf{p}\} \mid \mathsf{p}[[\![N'[x := [\![V]\!]_\mathsf{p}]]\!]_\mathsf{p}]$. We define $M'' = N'$ and the transitions used in $\mathcal{M} \to^*_{[\![D]\!]} \mathcal{N}$ can be used on M''. By induction, since $[\![N']\!]_{\mathsf{p}'} \sqsupseteq [\![N']\!]_{\mathsf{p}'} \sqcup [\![N'']\!]_{\mathsf{p}'}$ the result follows from using rules NAbsApp and NCaseL.
    * Assume $N = \textbf{Inr}\ V$. Then the case is similar to the previous.
    * Otherwise, we use either rule NCase or rule NCase2. If we use rule NCase, we have a transition $[\![N]\!]_\mathsf{p} \xrightarrow{\tau} B$ such that

      $$\mathcal{M} = [\![M]\!]\backslash\{\mathsf{p}\} \mid \mathsf{p}[\textbf{case}\ B\ \textbf{of Inl}\ x \Rightarrow [\![N']\!]_\mathsf{p};\ \textbf{Inr}\ x' \Rightarrow [\![N'']\!]_\mathsf{p}]$$

      and the result follows from induction similar to the last application case.
      If we use rule NCase2 then $[\![N']\!]_\mathsf{p} \xrightarrow{\tau}_\mathbb{D} B$ and $[\![N'']\!]_\mathsf{p} \xrightarrow{\tau}_\mathbb{D} B$. If $[\![N']\!]_\mathsf{p} \xrightarrow{\tau}_\mathbb{D} B$ then by induction, $N' \to^*_D N'''$ and $[\![N']\!]\backslash\{\mathsf{p}\} \mid \mathsf{p}[B] \to^*_\mathbb{D} \mathcal{N}''$ such that $\mathcal{N}'' \supseteq [\![N''']\!]$ and $N'' \to^*_D N''''$ and $[\![N'']\!]\backslash\{\mathsf{p}\} \mid \mathsf{p}[B] \to^*_\mathbb{D} \mathcal{N}'''$ such that $\mathcal{N}''' \supseteq [\![N'''']\!]$. Since N' and N'' are mergeable on other processes, the result follows from using rule InCase.
  - Assume $\mathsf{P} = \mathsf{p} \notin \mathrm{pn}(\mathrm{type}(N))$. Then we have three cases.
    * Assume $N = \textbf{Inl}\ V$. Then $[\![N]\!]_\mathsf{p} = \bot$ and $\mathcal{M} = [\![M]\!]\backslash\{\mathsf{p}\} \mid \mathsf{p}[[\![N']\!]_\mathsf{p} \sqcup [\![N'']\!]_\mathsf{p}]$. We define $M' = N'$ and the result follows.
    * Assume $N = \textbf{Inr}\ V$. Then the case is similar to the previous.
    * Otherwise, $[\![N]\!]_\mathsf{p} \neq L$ and we therefore have $[\![N]\!]_\mathsf{p} \xrightarrow{\tau} B$ and $\mathcal{M} = [\![M]\!]\backslash\{\mathsf{p}\} \mid \mathsf{p}[(\lambda x.[\![N']\!]_\mathsf{p} \sqcup [\![N'']\!]_\mathsf{p})\ B]$. We therefore have $[\![N]\!] \xrightarrow{\tau_\mathsf{p}} [\![N]\!]\backslash\{\mathsf{p}\} \mid \mathsf{p}[B]$, and by induction, $N \to_D N'''$ such that $[\![N]\!]\backslash\{\mathsf{p}\} \mid \mathsf{p}[B] \to^* \mathcal{N}'''$ for $\mathcal{N}''' \sqsupseteq [\![N''']\!]$. Since all these transitions can be propagated past N_2 in the network and the conditional or $(\lambda x.[\![N']\!]_{\mathsf{p}''} \sqcup [\![N'']\!]_{\mathsf{p}''})$ in any other process p' involved, we get the result for $M' = \textbf{case}\ N'''\ \textbf{of Inl}\ x \Rightarrow N';\ \textbf{Inr}\ x' \Rightarrow N''$.
  - Assume $\mathsf{P} = \mathsf{q}, \mathsf{p}$. Then the logic is similar to the third subcases of the previous two cases.
- Assume $M = \textbf{select}_{\mathsf{q},\mathsf{p}}\ \ell\ N$. This is similar to the $N_1 = \textbf{com}_{\mathsf{q},\mathsf{p}}$ case above.
- Assume $M = f(\mathsf{p}_1, \ldots, \mathsf{p}_n)$. Then

$$[\![M]\!] = \prod_{i=1}^{n} \mathsf{p}_i[f_i(\mathsf{p}_1, \ldots, \mathsf{p}_{i-1}, \mathsf{p}_{i+1}, \ldots, \mathsf{p}_n)] \mid \prod_{\mathsf{p} \notin \{\mathsf{p}_1, \ldots, \mathsf{p}_n\}} \mathsf{p}[\bot]$$

We therefore have some process p such that $\mathsf{P} = \mathsf{p}$ and $([\![M]\!]\backslash\mathsf{p}_i) \mid \mathsf{p}_i[[\![D]\!](f_i(\vec{\mathsf{p}'}))[\vec{\mathsf{p}'} := \mathsf{p}_1, \ldots, \mathsf{p}_{i-1}, \mathsf{p}_{i+1}, \ldots, \mathsf{p}_n]] \to^* \mathcal{N}$. We then define the required choreography $M'' = D(f(\mathsf{p'}_1, \ldots, \mathsf{p'}_n))[\mathsf{p'}_1, \ldots, \mathsf{p'}_n := \mathsf{p}_1, \ldots, \mathsf{p}_n]$ and network

$$\mathcal{N} = [\![M'']\!] = \prod_{i=1}^{n} \mathsf{p}_i[[\![D]\!](f_i(\vec{\mathsf{p}'}))[\vec{\mathsf{p}'} := \mathsf{p}_1, \ldots, \mathsf{p}_{i-1}, \mathsf{p}_{i+1}, \ldots, \mathsf{p}_n]] \mid \prod_{\mathsf{p} \notin \mathsf{p}_1, \ldots, \mathsf{p}_n} \mathsf{p}[\bot]$$

and the result follows from induction. ◀