


# Modular Compilation for Higher-order Functional Choreographies

Luís Cruz-Filipe ✉ 

Department of Mathematics and Computer Science, University of Southern Denmark, Denmark

Eva Graversen ✉ 

Department of Mathematics and Computer Science, University of Southern Denmark, Denmark

Lovro Lugović ✉ 

Department of Mathematics and Computer Science, University of Southern Denmark, Denmark

Fabrizio Montesi ✉ 

Department of Mathematics and Computer Science, University of Southern Denmark, Denmark

Marco Peressotti ✉ 

Department of Mathematics and Computer Science, University of Southern Denmark, 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

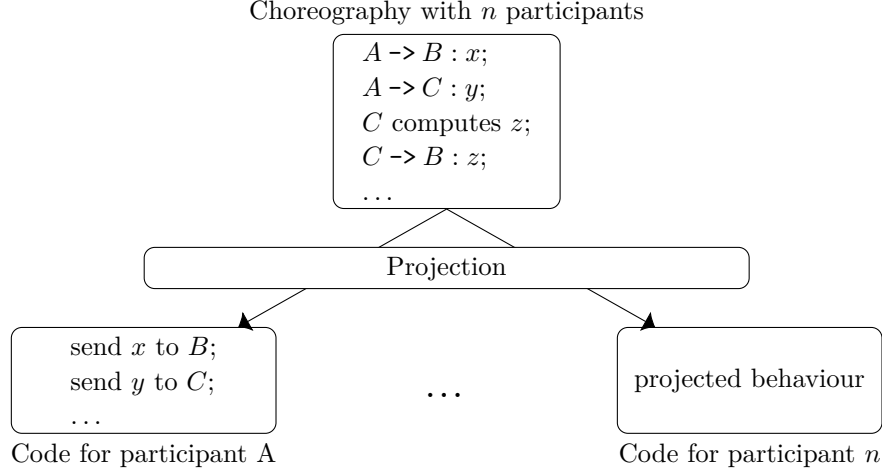
**Keywords and phrases** Choreographies, Concurrency,  $\lambda$ -calculus, Type Systems

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

## 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].



■ **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]).

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  $\rightarrow$  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  $\lambda$ -calculus is extended with choreographic primitives for communications, yielding a simple yet expressive model called Choral, 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.

## 73 The problem

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

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

## 89 This work

90 We define a notion of EPP for  $\text{Chor}\lambda$ , capitalising on the design of its type system and  
91 semantics.

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

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

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

114 A unique feature of  $\text{Chor}\lambda$  is that conditionals can use whole choreographies as conditions,  
115 and in particular ones that return distributed data structures—data structures that compose  
116 data residing at different participants. For the first time, our EPP leverages this feature  
117 to offer a new method for capturing *knowledge of choice*—distributed agreement regarding  
118 choices between alternative choreographic behaviours [4]. Specifically, we can statically

119 guarantee that two (or more) participants will agree on the instantiation of a sum type  
 120 (representing alternative choices) solely by performing independent local checks. When this is  
 121 used in a conditional, it means that all participants are guaranteed to make the same choice  
 122 at runtime. This gives a simpler alternative to existing verification methods for distributed  
 123 choices [21]. We call types used in this way *distributed choice types*.

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

## 132 Contribution

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

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

## 150 Structure

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

## 158 2 Background

159 In this section, we recap the theory of the choreographic  $\lambda$ -calculus (Chor $\lambda$ ) without recursion,  
 160 from [6]. Chor $\lambda$  extends the simply typed  $\lambda$ -calculus [5] with primitives that make distribution  
 161 and communication syntactically manifest.

## 162 System model

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

## 168 Syntax

169 The syntax of Chor $\lambda$  is given by the following grammar

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

174 where  $M$  is a choreography,  $V$  is a value,  $T$  is a type,  $x$  is a variable,  $l$  is a label,  $\mathbf{p}$  is a  
 175 process name, and  $\rho$  is a set of process names.

176 Terms are located at processes, to reflect distribution. For example, the value  $()@_{\mathbf{A}}$  reads  
 177 “the unit value at  $\mathbf{A}$ ”. Types are annotated with process names, as well. In the typing rules  
 178 of Chor $\lambda$  (shown later), term  $()@_{\mathbf{A}}$  has the type  $()@_{\mathbf{A}}$ , read “the unit type at  $\mathbf{A}$ ”. In our  
 179 examples, for simplicity, we assume the presence of primitives for integer values and an  
 180 integer type  $\text{Int}@_{\mathbf{p}}$  (“an integer at  $\mathbf{p}$ ”)—the formal treatment of these are straightforward  
 181 and similar to that of units.

182 Abstraction  $\lambda x : T.M$ , variable  $x$  and application  $MM$  are as in the standard (simply  
 183 typed)  $\lambda$ -calculus. Sums and products are constructed, respectively, by using **Inl**/**Inr** and  
 184 **Pair**. They are deconstructed in the usual way, respectively with **case** and **fst**/**snd**. The  
 185 constructors can take only values as arguments, but this does not restrict expressivity (cf.  
 186 [6]).

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

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

201 A key feature of Chor $\lambda$  is distributed data structures. For example, **Pair**  $()@_{\mathbf{p}} \ ()@_{\mathbf{q}}$  is  
 202 a distributed pair where the first element resides at  $\mathbf{p}$  and the second at  $\mathbf{q}$ . Types record  
 203 the distribution of values across processes: if  $\mathbf{p}$  occurs in the type given to  $V$  then part of  $V$   
 204 will be located at  $\mathbf{p}$ . A function may involve more processes than those listed in the types  
 205 of its input and output, so the type of abstractions  $T \rightarrow_{\rho} T'$  has the extra ingredient  $\rho$ ,  
 206 which denotes the processes that may participate in the computation of the function besides  
 207 those occurring in  $T$  or  $T'$ . We simply write  $T \rightarrow T'$  in place of  $T \rightarrow_{\emptyset} T'$ . For example,

208 if Alice wants to communicate an integer to Bob directly (without intermediaries), she can  
 209 use a choreography of type  $\text{Int}@Alice \rightarrow \text{Int}@Bob$ ; however, if the communication might go  
 210 through a proxy, then she can use a choreography of type  $\text{Int}@Alice \rightarrow_{\{\text{Proxy}\}} \text{Int}@Bob$ . The  
 211 information given by  $\rho$  gives control on what processes may participate in choreographies  
 212 taken as arguments. As we show in Section 3, this information is essential to achieve a  
 213 modular EPP.

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

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

221  $\lambda fun : \text{Int}@S \rightarrow_{\emptyset} \text{Int}@S. \lambda val : \text{Int}@C. \mathbf{com}_{S,C} (fun (\mathbf{com}_{C,S} val))$  ◀

## 224 Typing

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

$$\begin{array}{c}
 \frac{\text{pn}(T) = \{p\} \quad \{p, q\} \subseteq \Theta}{\Theta; \Gamma \vdash \mathbf{com}_{p,q} : T \rightarrow_{\emptyset} T[p := q]} [\text{TCom}] \\
 \frac{\Theta; \Gamma \vdash N : T \rightarrow_{\rho} T' \quad \Theta; \Gamma \vdash M : T}{\Theta; \Gamma \vdash N M : T'} [\text{TApp}] \\
 \frac{\Theta'; \Gamma, x : T \vdash M : T' \quad \rho \cup \text{pn}(T) \cup \text{pn}(T') = \Theta' \subseteq \Theta}{\Theta; \Gamma \vdash \lambda x : T. M : T \rightarrow_{\rho} T'} [\text{TAbs}]
 \end{array}$$

232 A communication is typed as a function from any type  $T$  located entirely at the sender  $p$  to  
 233 the same type moved to the receiver, as long as both process names are in  $\Theta$ . Application and  
 234 abstraction are typed similarly to simply-typed  $\lambda$ -calculus, extended with  $\rho$  and  $\Theta$  (whose  
 235 consistency is checked in rule TABS). Note that  $\rho$  and  $\Theta$  in rule TABS are not necessarily  
 236 minimal, and it is possible to type, e.g.,  $\{p, q\}; \emptyset \vdash \lambda x : \text{Int}@p. x : \text{Int}@p \rightarrow_{\{q\}} \text{Int}@p$ . A  
 237 minimal  $\rho$  would consist of those processes that appear either in  $M$  or in the types of the  
 238 free variables of  $M$  according to  $\Gamma$ .

239 ▶ **Example 2.** Let  $h$  be the function  $\lambda x : \text{Int}@Alice. \mathbf{com}_{\text{Proxy}, \text{Bob}} (\mathbf{com}_{\text{Alice}, \text{Proxy}} x)$ , which  
 240 communicates an integer from Alice to Bob by passing through an intermediary Proxy. Then,  
 241  $\{Alice, Bob, Proxy\}; \emptyset \vdash h : \text{Int}@Alice \rightarrow_{\{\text{Proxy}\}} \text{Int}@Bob$ . For any term  $M$ , the composition  
 242  $h M$  is well-typed if  $M$  has type  $\text{Int}@Alice$ , denoting that the evaluation of  $M$  will yield  
 243 an integer at Alice. By contrast,  $h 5@Bob$  is ill-typed because of wrong data locality (the  
 244 argument is not at the process expected by  $h$ ). ◀

## 245 Semantics

246 Chor $\lambda$  comes with an operational semantics given in terms of labelled reductions. Reduction  
 247 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.

$$\frac{}{\lambda x : T. M \ V \xrightarrow{\emptyset} M[x := V]} [\text{APPABS}] \quad \frac{\text{fv}(V) = \emptyset}{\mathbf{com}_{q,p} \ V \xrightarrow{\{q,p\}} V[q \mapsto p]} [\text{COM}]$$

Rule APPABS is the standard application rule of call-by-value  $\lambda$ -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,  $\mathbf{com}_{\text{Alice}, \text{Bob}} 3@ \text{Alice} \xrightarrow{\{\text{Alice}, \text{Bob}\}} 3@ \text{Bob}$ . 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  $\text{Chor}\lambda$ ’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 $\text{Chor}\lambda$

In this section we develop a theory of EPP for finite  $\text{Chor}\lambda$ .

#### 3.1 Process Language

We write implementations of choreographies in a distributed  $\lambda$ -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

$$\begin{aligned} B &::= L \mid B \ B \mid \mathbf{case} \ B \ \mathbf{of} \ \mathbf{Inl} \ x \Rightarrow B; \ \mathbf{Inr} \ x \Rightarrow B \mid \oplus_p \ l \ B \\ &\quad \mid \&_p \{l_1 : B_1, \dots, l_n : B_n\} \\ L &::= x \mid \lambda x : T. B \mid \mathbf{Inl} \ L \mid \mathbf{Inr} \ L \mid \mathbf{fst} \mid \mathbf{snd} \mid \mathbf{Pair} \ L \ L \mid () \mid \mathbf{recv}_p \mid \mathbf{send}_p \mid \perp \\ T &::= T \rightarrow T \mid T + T \mid T \times T \mid () \mid \perp \end{aligned}$$

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

The terms from the  $\lambda$ -calculus are standard. Pairs and sums work as described for  $\text{Chor}\lambda$ , 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  $\&_p \{l_1 : B_1, \dots, l_n : B_n\}$ , which offers a number of different ways it can continue for another process  $p$  to choose from, and the *choice* term  $\oplus_p \ l \ B$ , which directs  $p$  to continue as the behaviour labelled  $l$ . Likewise, value communication is divided into a *send* to  $p$  action,  $\mathbf{send}_p$ , and a *receive* from  $p$  action,  $\mathbf{recv}_p$ .

We also add the no-op term mentioned in the introduction,  $\perp$ , and its type,  $\perp$ . A term  $\perp$  represents a terminated behaviour with no result. This term is used in the semantics of send and receive: locally,  $\mathbf{send}_p$  acts as a function that can take any input and returns  $\perp$ , and  $\mathbf{recv}_p$  a function that given  $\perp$  returns some value. More interestingly,  $\perp$  also plays an important role wrt modularity in our notion of EPP, which we will discuss later in our

$$\begin{array}{c}
\frac{\Sigma; \Gamma \vdash B : T}{\Sigma; \Gamma \vdash \oplus_p l B : T} [\text{NTCHOR}] \quad \frac{\Sigma; \Gamma \vdash B_i : T \text{ for } 1 \leq i \leq n}{\Sigma; \Gamma \vdash \&_p \{l_1 : B_1, \dots, l_n : B_n\} : T} [\text{NTOFF}] \\
\\
\frac{}{\Sigma; \Gamma \vdash \mathbf{send}_p : T \rightarrow \perp} [\text{NTSEND}] \quad \frac{}{\Sigma; \Gamma \vdash \mathbf{recv}_p : \perp \rightarrow T} [\text{NTRECV}] \\
\\
\frac{}{\Sigma; \Gamma \vdash \perp : \perp} [\text{NTBOTM}] \quad \frac{\Sigma; \Gamma \vdash B : \perp \quad \Sigma; \Gamma \vdash B' : \perp}{\Sigma; \Gamma \vdash B B' : \perp} [\text{NTAPP2}]
\end{array}$$

■ **Figure 2** Typing rules for behaviours (selected rules).

288 presentation of projection. All types but  $\perp$  are standard (as in  $\text{Chor}\lambda$ , but without process  
289 name annotations).

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

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

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

296 ▶ **Example 4.** Consider the choreography  $\mathbf{com}_{B,C}(\mathbf{com}_{A,B}()@A)$ . A correct implementation  
297 is the network  $A[\mathbf{send}_B()] \mid B[\mathbf{send}_C(\mathbf{recv}_A \perp)] \mid C[\mathbf{recv}_B \perp]$ . ◀

## 298 Typing

299 Behaviours are typed with judgements of the form  $\Gamma \vdash B : T$ . The typing rules are the local  
300 counterparts of those in  $\text{Chor}\lambda$ , obtained by removing  $\Theta$  and process names in types. We  
301 add the  $\perp$  type for terms that can result in  $\perp$ . Figure 2 displays representative typing rules  
302 to deal with  $\perp$  and communications.

## 303 Semantics

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

306 Labels for network transitions have the form  $\tau_P$ , where  $P$  ranges over sets of one or two  
307 process names. Rule  $\text{NPRO}$  annotates an internal transition by a process with its name, and  
308 rule  $\text{NPAR}$  lifts transitions in parallel compositions.

309 The transition axioms for send and receive are typical of process calculi with early  
310 semantics. Send and receive transitions are matched in rule  $\text{NCOM}$  to perform a communica-  
311 tion. The label  $\tau_{p,q}$  denotes an internal move ( $\tau$ ) and manifests the names of processes that  
312 contribute to performing it ( $p$  and  $q$ ). We treat the subscript  $p, q$  as an unordered set that  
313 consists of the two process names.

314 The  $P$ -annotations in labels enable the formulation of the next lemma, which we use in  
315 some of our proofs to focus on the processes involved in a transition. The proof of this result  
316 and others for the full  $\text{Chor}\lambda$  language are provided in Appendix A.

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

318 Most of the other rules follow the same intuition and are otherwise standard. The  
319 exception is rule  $\text{NBOTM}$ , which garbage collects  $\perp$  terms. We discuss the role of this rule in  
320 Example 9, after having presented our notion of EPP.



$$\begin{array}{c}
\frac{\text{fv}(L) = \emptyset}{\text{send}_p L \xrightarrow{\text{send}_p L} \perp} [\text{NSEND}] \quad \frac{}{\text{recv}_p \perp \xrightarrow{\text{recv}_p L} L} [\text{NRECV}] \\
\\
\frac{B_1 \xrightarrow{\text{send}_q L} B'_1 \quad B_2 \xrightarrow{\text{recv}_p L} B'_2}{p[B_1] \mid q[B_2] \xrightarrow{\tau_{p,q}} p[B'_1] \mid q[B'_2]} [\text{NCOM}] \\
\\
\frac{}{\oplus_p l B \xrightarrow{\oplus_p l} B} [\text{NCHO}] \quad \frac{}{\&_p \{\ell_1 : B_1, \dots, \ell_n : B_n\} \xrightarrow{\&_p \ell_i} B_i} [\text{NOFF}] \\
\\
\frac{B_1 \xrightarrow{\oplus_q \ell} B'_1 \quad B_2 \xrightarrow{\&_p \ell} B'_2}{p[B_1] \mid q[B_2] \xrightarrow{\tau_{p,q}} p[B'_1] \mid q[B'_2]} [\text{NSEL}] \\
\\
\frac{}{(\lambda x : T.B) L \xrightarrow{\tau} B[x := L]} [\text{NABSAAPP}] \quad \frac{}{\perp \perp \xrightarrow{\tau} \perp} [\text{NBOTM}] \\
\\
\frac{B \xrightarrow{\tau} B'}{p[B] \xrightarrow{\tau_p} p[B']} [\text{NPRO}] \quad \frac{\mathcal{N} \xrightarrow{\tau_p} \mathcal{N}''}{\mathcal{N} \mid \mathcal{N}' \xrightarrow{\tau_p} \mathcal{N}'' \mid \mathcal{N}'} [\text{NPAR}]
\end{array}$$

■ **Figure 3** Network semantics (representative rules).

## 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  $\Theta$ ,  $\Gamma$ , and  $T$ . The definition of EPP formally depends on this typing derivation, but to keep notation simple we write just  $\llbracket M \rrbracket_p$  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  $\text{type}(M)$ .

Projection translates each choreographic term to a corresponding local behaviour. For example, a communication term  $\text{com}_{p,q}$  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  $\perp$  in our theory of EPP.

► **Example 6.** Let  $M = \lambda x : \text{Int}@p.M'$  for some  $M'$ , and consider the issue of defining its projection on a process  $q$  different than  $p$ ,  $\llbracket M \rrbracket_q$ . 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  $(\lambda x)$  of  $M$  in the implementation of  $q$ .

For example, for  $M = \lambda x : \text{Int}@p.2@q$ , we could design EPP such that  $\llbracket M \rrbracket_q = 2$ . This works when  $M$  is used in an application as  $(\lambda x : \text{Int}@p.2@q) 1@p$ , where  $\llbracket M \rrbracket_q = 2$  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  $\lambda x : \text{Int}@p$ , in general it could still participate in the context that produces the value that  $x$  is going to be replaced with. For example, let  $M'' = (\lambda x : \text{Int}@p.\text{com}_{q,p} 2@q) (\text{com}_{q,p} 1@q)$ , 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  $\llbracket M'' \rrbracket_q = (\text{send}_p 2) (\text{send}_p 1)$ . 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  $(\lambda x : \perp. \text{send}_p 2) (\text{send}_p 1)$ . Our process language is carefully designed to make terms like this normalise gracefully: after executing  $\text{send}_p 1$  the righthandside is  $\perp$ , 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  $\lambda x : \text{Int}@p.1@p$ , then  $\llbracket M \rrbracket_q$  should clearly be  $\perp$ . The alternative,  $\lambda x : \perp. \perp$ , would break modularity of EPP because the structure of  $\llbracket M \rrbracket_q$  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  $\perp$  and its companion rule NBTM in Example 9.  $\blacktriangleleft$

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  $\text{pn}(\text{type}(M))$  for this set of process names.

The second ingredient concerns knowledge of choice. When projecting a conditional **case**  $M$  of **Inl**  $x \Rightarrow M'$ ; **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

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

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

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

Using merging, its projection on process  $q$  is  $\llbracket C \rrbracket_q = \&_p\{\text{left} : 0, \text{right} : 1\}$ .  $\blacktriangleleft$

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

Intuitively, projecting a choreography on a process that is not involved in it returns a  $\perp$ . 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  $M$ , 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

### Choreographies

$$\begin{aligned}
\llbracket M \ N \rrbracket_p &= \begin{cases} \llbracket M \rrbracket_p \ \llbracket N \rrbracket_p & \text{if } p \in \text{pn}(\text{type}(M)) \text{ or } p \in \text{pn}(M) \cap \text{pn}(N) \\ \llbracket M \rrbracket_p & \text{if } \llbracket N \rrbracket_p = \perp \\ \llbracket N \rrbracket_p & \text{otherwise} \end{cases} \\
\llbracket \lambda x : T. M \rrbracket_p &= \begin{cases} \lambda x : \llbracket T \rrbracket_p . \llbracket M \rrbracket_p & \text{if } p \in \text{pn}(\text{type}(\lambda x : T. M)) \\ \perp & \text{otherwise} \end{cases} \\
\llbracket \text{case } M \text{ of } \text{Inl } x \Rightarrow N; \text{Inr } x' \Rightarrow N' \rrbracket_p &= \\
&\begin{cases} \text{case } \llbracket M \rrbracket_p \text{ of } \text{Inl } x \Rightarrow \llbracket N \rrbracket_p; \text{Inr } x' \Rightarrow \llbracket N' \rrbracket_p & \text{if } p \in \text{pn}(\text{type}(M)) \\ \llbracket M \rrbracket_p & \text{if } \llbracket N \rrbracket_p = \llbracket N' \rrbracket_p = \perp \\ \llbracket N \rrbracket_p \sqcup \llbracket N' \rrbracket_p & \text{if } \llbracket M \rrbracket_p = \perp \\ (\lambda x'' : \perp . \llbracket N \rrbracket_p \sqcup \llbracket N' \rrbracket_p) \ \llbracket M \rrbracket_p & \text{otherwise, for some} \\ & x'' \notin \text{fv}(N) \cup \text{fv}(N') \end{cases} \\
\llbracket \text{Inl } V \rrbracket_p &= \begin{cases} \text{Inl } \llbracket V \rrbracket_p & \text{if } p \in \text{pn}(\text{type}(\text{Inl } V)) \\ \perp & \text{otherwise} \end{cases} \quad \llbracket \text{fst} \rrbracket_p = \begin{cases} \text{fst} & \text{if } p \in \text{pn}(\text{type}(\text{fst})) \\ \perp & \text{otherwise} \end{cases} \\
\llbracket \text{select}_{q,q'} \ l \ M \rrbracket_p &= \begin{cases} \oplus_{q'} \ l \ \llbracket M \rrbracket_p & \text{if } p = q \neq q' \\ \&_q \{l : \llbracket M \rrbracket_p\} & \text{if } p = q' \neq q \\ \llbracket M \rrbracket_p & \text{otherwise} \end{cases} \\
\llbracket \text{com}_{q,q'} \rrbracket_p &= \begin{cases} \lambda x : \llbracket T \rrbracket_p . x & \text{if } p = q = q' \text{ and } \text{type}(\text{com}_{q,q'}) = T \rightarrow_{\emptyset} T' \\ \text{send}_{q'} & \text{if } p = q \neq q' \\ \text{recv}_q & \text{if } p = q' \neq q \\ \perp & \text{otherwise} \end{cases} \\
\llbracket () @ q \rrbracket_p &= \begin{cases} () & \text{if } q = p \\ \perp & \text{otherwise} \end{cases} \quad \llbracket x \rrbracket_p = \begin{cases} x & \text{if } p \in \text{pn}(\text{type}(x)) \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

### Types

$$\begin{aligned}
\llbracket () @ q \rrbracket_p &= \begin{cases} () & \text{if } q = p \\ \perp & \text{otherwise} \end{cases} \quad \llbracket T \times T' \rrbracket_p = \begin{cases} \llbracket T \rrbracket_p \times \llbracket T' \rrbracket_p & \text{if } p \in \text{pn}(T \times T') \\ \perp & \text{otherwise} \end{cases} \\
\llbracket T \rightarrow_{\rho} T' \rrbracket_p &= \begin{cases} \llbracket T \rrbracket_p \rightarrow \llbracket T' \rrbracket_p & \text{if } p \in \rho \cup \text{pn}(T) \cup \text{pn}(T') \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

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

the condition  $M$  without knowing the final choice, e.g., if  $M = \mathbf{com}_{p,q} M'$ . 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  $\perp$ ).

The projection of abstraction illustrates the necessity of the  $\rho$  annotation on abstraction types. For example, consider an application of a communication via a proxy  $(\lambda x : \text{Int}@p \rightarrow_{\{r\}} \text{Int}@q.x \text{ } 3@p) (\lambda y : \text{Int}@p.\mathbf{com}_{r,q} \mathbf{com}_{p,r} y)$ . Without the annotation  $\{r\}$  in subterm  $(\lambda x : \text{Int}@p \rightarrow_{\{r\}} \text{Int}@q.x \text{ } 3@p)$ , the projection of this subterm on  $r$  would just be  $\perp$ , 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  $\perp$  at any process not used in  $T$ .

► **Example 9.** Let  $M = (\mathbf{com}_{p,q} (\lambda x : \text{Int}@p.3@p)) (\mathbf{com}_{p,q} 5@p)$ , where a function and a value are both sent from  $p$  to  $q$  before being applied at  $q$ . The implementation of  $q$  is  $\llbracket M \rrbracket_q = (\mathbf{recv}_p \perp) (\mathbf{recv}_p \perp)$ , whose execution is straightforward. At  $p$ , however, we have that  $\llbracket M \rrbracket_p = (\mathbf{send}_q (\lambda x : \text{Int}.3)) (\mathbf{send}_q 5)$ , which after executing the two send actions becomes  $\perp \perp$ . After executing its two communications, the choreography  $M$  becomes  $M' = (\lambda x : \text{Int}@q.3@q) 5@q$ .  $M'$  is located entirely at  $q$ , and therefore  $\llbracket M' \rrbracket_p = \perp$ , which is different than the  $\perp \perp$  reached by  $\llbracket M \rrbracket_p$ . We therefore need a way to make the application  $\perp \perp$  become  $\perp$ . Rule NBOTM serves this purpose. The fact that this is not possible with two units is the key semantic difference between  $\perp$  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  $\llbracket \Gamma \rrbracket \vdash \llbracket M \rrbracket_p : \llbracket T \rrbracket_p$ , where  $\llbracket \Gamma \rrbracket$  are defined by applying EPP to all types occurring in  $\Gamma$ .

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

$$\begin{aligned} \llbracket M \rrbracket_C &= \lambda f : \perp. \lambda val : \text{Int}. \mathbf{recv}_S (\mathbf{send}_S val) \\ \llbracket M \rrbracket_S &= \lambda f : (\text{Int} \rightarrow \text{Int}). \lambda val : \perp. \mathbf{send}_C (f (\mathbf{recv}_C \perp)) \end{aligned}$$

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  $\llbracket - \rrbracket$  is called modular if  $\llbracket C[M] \rrbracket_p = \llbracket C[N] \rrbracket_p$  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  $p \notin \Theta$ .

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  $\lambda x. \perp : \perp$ . 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 reveal 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  $p \notin \Theta$  then  $\llbracket M \rrbracket_p = \perp$ .*

**Proof.** Follows from  $p \notin \Theta$  implying  $p \notin \text{pn}(T) \cup \text{pn}(M)$  and induction on the derivation of  $\llbracket M \rrbracket_p$ . ◀

► **Proposition 14.** *The EPP  $\llbracket - \rrbracket$  given in Definition 8 is modular.*

**Proof.** Follows from Lemma 13 and observing that the projection of any context always treats  $\perp$  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

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

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

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

We can use distributed booleans to codify distributed choices, in this case by having both  $\mathbf{p}$  and  $\mathbf{q}$  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  $\mathbf{p}$  and  $\mathbf{q}$ , both of which are guaranteed to make the same choice. First we have the projections

$$\llbracket M \rrbracket_{\mathbf{p}} = \lambda x : (()) \times \perp + (()) \times \perp. \text{case } x \text{ of } \mathbf{Inl } y \Rightarrow \text{send}_{\mathbf{q}} 3; \mathbf{Inr } y \Rightarrow \perp$$

and

$$\llbracket M \rrbracket_{\mathbf{q}} = \lambda x : (\perp \times (())) + (\perp \times ()). \text{case } x \text{ of } \mathbf{Inl } y \Rightarrow \text{recv}_{\mathbf{p}} \perp; \mathbf{Inr } y \Rightarrow 5$$

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  $\mathbf{Inl } (\mathbf{Pair } (())@ \mathbf{p} (())@ \mathbf{q})$  or  $\mathbf{Inr } (\mathbf{Pair } (())@ \mathbf{p} (())@ \mathbf{q})$ . From the projection we get  $\llbracket \mathbf{Inl } (\mathbf{Pair } (())@ \mathbf{p} (())@ \mathbf{q}) \rrbracket_{\mathbf{p}} = \mathbf{Inl } (\mathbf{Pair } (()) \perp)$  and  $\llbracket \mathbf{Inl } (\mathbf{Pair } (())@ \mathbf{p} (())@ \mathbf{q}) \rrbracket_{\mathbf{q}} = \mathbf{Inl } (\mathbf{Pair } \perp (()))$ , and similar for the  $\mathbf{Inr}$  case. We therefore know that  $\text{Chor}\lambda$ ’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  $(())@ \mathbf{p} + (())@ \mathbf{p} \times (())@ \mathbf{q} + (())@ \mathbf{q}$ , 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  $B \sqsupseteq B'$ , 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  $p$ ,  $\mathcal{N}(p) \sqsupseteq \mathcal{N}'(p)$ . 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,

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

and its projection  $B$  on  $q$ ,  $B = \llbracket C \rrbracket_q = \&_p\{\text{left} : 0, \text{right} : 1\}$ .

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

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.,  $\Pi_p \mathcal{N}[P_p] \equiv \Pi_p \mathcal{N}'[P'_p]$  iff  $P_p \equiv P'_p$  for all  $p$ s) and  $\mathcal{N} \sqsupseteq \mathcal{N}'$  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  $\rightarrow^*$  denotes a sequence of transitions with any labels, and  $\rightarrow^+$  a nonempty such sequence.

► **Theorem 18 (Completeness).** Given a closed choreography  $M$ , if  $M \xrightarrow{P} M'$ ,  $\Theta; \Gamma \vdash M : T$ , and  $\llbracket M \rrbracket$  is defined, then there exist networks  $\mathcal{N}$  and  $M''$  such that:  $\llbracket M \rrbracket \rightarrow^+ \mathcal{N}$ ;  $M' \rightarrow^* M''$ ; and  $\mathcal{N} \sqsupseteq \llbracket M'' \rrbracket$ .

► **Theorem 19 (Soundness).** Given a closed choreography  $M$ , if  $\Theta; \Gamma \vdash M : T$  and  $\llbracket M \rrbracket \rightarrow^* \mathcal{N}$  for some network  $\mathcal{N}$ , then there exist a choreography  $M'$ , and a network  $\mathcal{N}'$  such that:  $M \rightarrow^* M'$ ;  $\mathcal{N} \rightarrow^* \mathcal{N}'$ ; and  $\mathcal{N}' \sqsupseteq \llbracket M' \rrbracket$ .

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  $\llbracket M \rrbracket \rightarrow^* \mathcal{N}$  for some network  $\mathcal{N}$ , either there exists  $p$  and  $\mathcal{N}'$  such that  $\mathcal{N} \xrightarrow{p} \mathcal{N}'$  or  $\mathcal{N} = \prod_{p \in \text{pn}(M)} p[L_p]$ .

## 4 Recursion

So far we have worked with a recursion-free subset of  $\text{Chor}\lambda$ . In this section, we extend our development to the full language presented of  $\text{Chor}\lambda$ , 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  $\text{Chor}\lambda$  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  $M ::= \dots \mid f(\vec{p})$ . A function call  $f(\vec{p})$  invokes  $f$  by instantiating its parameters with the process names  $\vec{p}$ , 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  $\text{Chor}\lambda$  with recursion are extended to the form  $\ell, P$ , where the new ingredient  $\ell$  can be either  $\tau$  or  $\lambda$ . The need for  $\ell$  is explained later.

$$\frac{D(f(\vec{p})) = M}{f(\vec{p}) \xrightarrow{\tau, \emptyset}_D M[\vec{p}' \mapsto \vec{p}]} [\text{DEF}]$$

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

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

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

► **Example 21 (Remote Map).** With recursive functions, we can write more complex choreographies that call themselves and each other. Let `remoteFunction(C, S)` be defined as the choreography in Example 1. We use it to define a function `remoteMap(C, 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) = λfun : Int@S → Int@S. λlist : [Int]@C.
  case list of
  Inl x ⇒ selectC, S stop ()@C;
  Inr x ⇒ selectC, S again
    cons(C) (remoteFunction(C, S) fun (fst x)) (remoteMap(C, S) fun (snd x))
```

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

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

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

538  $\text{modPow}(R) : \text{Int}@R \rightarrow \text{Int}@R \rightarrow \text{Int}@R \rightarrow \text{Int}@R$

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

```
541 diffieHellman(P, Q) =
542   λpsk : Int@P. λqsk : Int@Q. λpsg : Int@P.
543   λqsg : Int@Q. λpsp : Int@P. λqsp : Int@Q.
544   pair (modPow(P) psg (comQ,P (modPow(Q) qsg qsk qsp)) psp)
545       (modPow(Q) qsg (comP,Q (modPow(P) psg psk psp)) qsp)
```

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

552  $\text{Int}@P \rightarrow \text{Int}@Q \rightarrow \text{Int}@P \rightarrow \text{Int}@Q \rightarrow \text{Int}@P \rightarrow \text{Int}@Q \rightarrow \text{Int}@P \times \text{Int}@Q$

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

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

```
557 makeSecureChannels(P, Q) = λkey : Int@P × Int@Q.
558   Pair (λval : String@P. (dec(Q) (snd key) (comP,Q (enc(P) (fst key) val))))
559       (λval : String@Q. (dec(P) (fst key) (comQ,P (enc(Q) (snd key) val))))
```

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

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

564  $(\text{Int}@P \times \text{Int}@Q) \rightarrow ((\text{String}@P \rightarrow \text{String}@Q) \times (\text{String}@Q \rightarrow \text{String}@P))$

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

## 566 Process language

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

## 573 Endpoint Projection

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

$$575 \quad \llbracket f(\vec{p}) \rrbracket_p = \begin{cases} f_i(p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n) & \text{if } \vec{p} = p_1, \dots, p_{i-1}, p, p_{i+1}, \dots, p_n \\ \perp & \text{otherwise} \end{cases}$$



$$\begin{aligned}
576 \quad \llbracket t@p \rrbracket_p &= \begin{cases} t_i & \text{if } \vec{p} = p_1, \dots, p_{i-1}, p, p_{i+1}, \dots, p_n \\ \perp & \text{otherwise} \end{cases} \\
577 \quad \llbracket D \rrbracket &= \{f_i(p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n) \mapsto \llbracket M \rrbracket_{p_i} \mid D(f(p_1, \dots, p_n)) = M\} \\
578
\end{aligned}$$

579 Each named function gets projected to a different named function for each process in its  
580 list of parameters, with the projected environment now treating each of these as separate  
581 functions parametric on the remaining involved processes. These parameters are needed to  
582 implement interactions. Each process can enter a named function independently. Thus, for  
583 example, if  $D(f(p, q)) = M$  we get  $\llbracket D \rrbracket (f_1(q)) = \llbracket M \rrbracket_p$  and  $\llbracket D \rrbracket (f_2(p)) = \llbracket M \rrbracket_q$ .

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.  $\Sigma$  is otherwise projected similarly to  $D$ . For example, if  $t@p, q = T \in \Sigma$  then  $t_1 = \llbracket T \rrbracket_p \in \llbracket \Sigma \rrbracket$  and  $t_2 = \llbracket T \rrbracket_q \in \llbracket \Sigma \rrbracket$ .

$$\llbracket \Sigma \rrbracket = \{t_i = \llbracket T \rrbracket_{p_i} \mid t@p_1, \dots, p_n = T \in \Sigma\}$$

584 ▶ **Example 23** (Projecting Example 21). Projecting the choreography in Example 21 yields  
585 `remoteMap1` (for the client) and `remoteMap2` (for the server) below. The bodies of `remoteFunction1`  
586 and `remoteFunction2` are the terms in Example 11.

```

587 remoteMap1(S) = λfun : ⊥. λlist : [Int].
588   case list of
589     Inl x ⇒ ⊕S stop ();
590     Inr x ⇒ ⊕S again
591       cons1 (remoteFunction1(S) ⊥ (fst x)) (remoteMap1(S) ⊥ (snd x))
592 remoteMap2(C) = λfun : Int → Int. λlist : ⊥.
593   &C{stop : ⊥, again : (remoteFunction2(C) fun ⊥) (remoteMap2(C) fun ⊥)} ◀
594
595

```

596 ▶ **Example 24** (Projecting Example 22). Projecting our choreographies `diffieHellman(P, Q)`  
597 and `makeSecureChannels(P, Q)` for process P yields the following behaviours.

```

598 ⌊D(diffieHellman(P, Q))⌋1(Q) = λpsk : Int. λqsk : ⊥. λpsg : Int. λqsg : ⊥. λpsp : Int. λqsp : ⊥.
599   pair (modPow1 psg (recvQ ⊥)) psp
600     (sendQ (modPow1 psg psk psp))
601
602 ⌊D(makeSecureChannels(P, Q))⌋1(Q) = λkey : Int × ⊥.
603   Pair (λval : String. ((snd key) (sendQ (encrypt1 (fst key) val))))
604     (λval : ⊥. (decrypt1 (fst key) (recvQ (snd key))))
605
606

```

607 Note the way function calls such as `modPow(P)` in the choreography get projected to  
608 `modPow1` on P, since they are treated as degenerate choreographies (they have only one  
609 process) and P is the first and only process involved. Conversely, `modPow(Q)` on P gets  
610 projected as  $\perp$  since it is located entirely at a different process.

## 611 4.2 Out-of-order execution

612 In the presence of recursion, getting a correspondence between a process and choreographic  
613 language becomes much more challenging. In our results for `Chorλ` without recursion, we  
614 relied on the fact that a choreography would eventually reduce to a value. This is no longer  
615 true as choreographies can now diverge, and worse they can diverge at one process without  
616 diverging at another. Let, for example,  $M = (\lambda x : \text{Int}@p. \text{fst } (\text{Pair } 5@q \ x)) \ f(p)$ . Assume  
617 that  $D(f(p_1)) = M'$ , where  $M'$  diverges. Then the reduction rules that we have seen so

$$\begin{array}{c}
\frac{x \notin \text{fv}(M')}{((\lambda x : T.M) N) M' \rightsquigarrow (\lambda x : T.(M M')) N} \text{[R-ABS R]} \\
\\
\frac{x, x' \notin \text{fv}(M) \quad \text{spn}(M) \cap \text{pn}(N) = \emptyset}{M (\text{case } N \text{ of } \text{Inl } x \Rightarrow M_1; \text{Inr } x' \Rightarrow M_2) \rightsquigarrow \text{case } N \text{ of } \text{Inl } x \Rightarrow (M M_1); \text{Inr } x' \Rightarrow (M M_2)} \text{[R-CASEL]} \\
\\
\frac{\text{spn}(M) \cap \text{pn}(N) = \emptyset}{M (\text{select}_{q,p} l N) \rightsquigarrow \text{select}_{q,p} l (M N)} \text{[R-SELL]} \\
\\
\frac{y \text{ fresh for } M}{\lambda x : T.M \rightsquigarrow \lambda y : T.M[x := y]} \text{[R-ALPH]}
\end{array}$$

■ **Figure 5** Rewriting of Chor $\lambda$  (representative rules).

far would not allow  $x$  to be instantiated. However,  $\llbracket f(\mathbf{p}) \rrbracket_q = \perp$ , so  $\llbracket M \rrbracket_q$  can reduce to 5. Therefore, we need a way to let  $M$  copy the reduction of **fst** (**Pair** 5@ $q$   $x$ ) to 5@ $q$ . In [6], we included corresponding reduction rules for Chor $\lambda$  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.

$$\frac{M \xrightarrow{\ell, \mathbf{P}}_D M' \quad \lambda x : T.M \xrightarrow{\lambda, \mathbf{P}}_D \lambda x : T.M'}{M N \xrightarrow{\ell, \mathbf{R}}_D M' N \quad \ell = \lambda \Rightarrow \mathbf{P} \cap \text{pn}(N) = \emptyset} \text{[INABS]} \quad \frac{M N \xrightarrow{\tau, \mathbf{P}}_D M' N}{\lambda x : T.M \xrightarrow{\lambda, \mathbf{P}}_D \lambda x : T.M'} \text{[APP1]}$$

Rule APP1 use the  $\ell$ -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  $(\lambda x : \text{Int}@p.\text{com}_{q,p} 4@q) (\text{com}_{q,p} 5@q)$ , 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  $M = (\lambda x : \text{Int}@q.\text{fst} (\text{Pair } 5@q x)) f(\mathbf{p}, \mathbf{q})$  to reduce to  $M' = (\lambda x : \text{Int}@p_2.5@q) f(\mathbf{p}, \mathbf{q})$ . Thus, the projections of  $M$  on  $\mathbf{p}$  and  $\mathbf{q}$  must be able to reduce to the projections of  $M'$ . For  $\mathbf{p}$  this is easy, since  $\llbracket M \rrbracket_p = \llbracket M' \rrbracket_p = \perp f_1(\mathbf{q})$ . For  $\mathbf{q}$ , however, we need  $\llbracket M \rrbracket_q = (\lambda x : \text{Int}.\text{fst} (\text{Pair } \perp x)) f_2(\mathbf{p})$  to reduce to  $\llbracket M' \rrbracket_q = (\lambda x : \text{Int}.\perp) f_2(\mathbf{p})$ , 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 (**send**, **recv**,  $\oplus$ ,  $\&$ ) from inside an abstraction. The reduction labels for the process language are thus simpler ( $\tau$  or  $\lambda$ ), since we do not need to track process names involved in actions.

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

$$\begin{array}{c}
\frac{\text{pn}(B) = \emptyset}{B \ (\&_{\mathbf{p}}\{l_1 : B_1, \dots, l_n : B_n\}) \rightsquigarrow \&_{\mathbf{p}}\{l_1 : B \ B_1, \dots, l_n : B \ B_n\}} \text{[LR-OFFL]} \\
\\
\frac{\text{pn}(B') = \emptyset}{B' \ (\oplus_{\mathbf{p}} l \ B) \rightsquigarrow \oplus_{\mathbf{p}} l \ (B' \ B)} \text{[LR-CHOL]} \quad \frac{}{\perp \ \perp \rightsquigarrow \perp} \text{[LR-BOTM]}
\end{array}$$

■ **Figure 6** Rewriting of behaviours (representative rules).

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

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

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  $\llbracket M \rrbracket_{\mathbf{p}} = ((\lambda x : \text{Int}.\lambda y : \perp.3) f_1()) \ \perp$  to be rewritten to  $(\lambda x : \text{Int}.\lambda y : \perp.3 \ \perp) f_1()$ , which can reduce to  $(\lambda x : \text{Int}.3) f_1()$ .

### 4.3 Properties

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

► **Theorem 25** (Completeness). *Given a closed choreography  $M$ , if  $M \xrightarrow{\tau, \mathbf{P}}_D M'$  and  $\Theta; \Sigma; \Gamma \vdash M : T$  and  $\llbracket M \rrbracket$  is defined, then there exist networks  $\mathcal{N}$  and  $M''$  such that:  $\llbracket M \rrbracket \rightarrow_{\llbracket D \rrbracket}^+ \mathcal{N}$ ;  $M' \rightarrow^* M''$ ; and  $\mathcal{N} \sqsubseteq \llbracket M'' \rrbracket$ .*

► **Theorem 26** (Soundness). *Given a closed choreography  $M$ , if  $\Theta; \Gamma \vdash M : T$  and  $\llbracket M \rrbracket \rightarrow^* \mathcal{N}$  for some network  $\mathcal{N}$ , then there exist a choreography  $M'$ , and a network  $\mathcal{N}'$  such that:  $M \xrightarrow{*}_D M'$ ;  $\mathcal{N} \rightarrow^* \mathcal{N}'$ ; and  $\mathcal{N}' \sqsubseteq \llbracket M' \rrbracket$ .*

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  $\llbracket M \rrbracket \rightarrow_{\llbracket D \rrbracket}^* \mathcal{N}$  for some network  $\mathcal{N}$ , either there exists  $\mathbf{P}$  and  $\mathcal{N}'$  such that  $\mathcal{N} \xrightarrow{\tau, \mathbf{P}}_{\llbracket D \rrbracket} \mathcal{N}'$  or  $\mathcal{N} = \prod_{\mathbf{p} \in \text{pn}(M)} \mathbf{p}[L_{\mathbf{p}}]$ .*

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

668 ▶ **Proposition 28.** *The EPP  $\llbracket - \rrbracket$  given in Definition 8 and extended with the equations in*  
 669 *Section 4.1 is modular.*

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

## 673 5 EAP

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

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

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

```

685  $\text{eap}(\mathbf{P}, \mathbf{A}, \mathbf{S}) = \lambda \text{methods} : [\text{AuthMethod}]@(\mathbf{P}, \mathbf{A}, \mathbf{S}).$ 
686    $\text{eapAuth}(\mathbf{P}, \mathbf{A}, \mathbf{S}) (\text{eapIdentity}(\mathbf{P}, \mathbf{A}, \mathbf{S}) \text{"Auth request"}@S) \text{methods}$ 
687
688
689  $\text{eapAuth}(\mathbf{P}, \mathbf{A}, \mathbf{S}) = \lambda id : \text{String}@S. \lambda \text{methods} : [\text{AuthMethod}]@(\mathbf{P}, \mathbf{A}, \mathbf{S}).$ 
690   if  $\text{empty}(\mathbf{P}, \mathbf{A}, \mathbf{S}) \text{methods}$  then
691      $\text{eapFailure}(\mathbf{P}, \mathbf{A}, \mathbf{S}) \text{"Try again later"}@S$ 
692   else
693     if  $(\text{fst } \text{methods}) id$  then
694        $\text{select}_{S, P} \text{ok } (\text{select}_{S, A} \text{ok } (\text{eapSuccess}(\mathbf{P}, \mathbf{A}, \mathbf{S}) \text{"Welcome"}@S))$ 
695     else
696        $\text{select}_{S, P} \text{ko } (\text{select}_{S, A} \text{ko } (\text{eapAuth}(\mathbf{P}, \mathbf{A}, \mathbf{S}) id (\text{snd } \text{methods})))$ 
697

```

698 For the sake of simplicity, we have left out the definitions of a couple of helper choreo-  
 699 graphs that are referenced in the example:

```

700  $\text{eapIdentity}(\mathbf{P}, \mathbf{A}, \mathbf{S}) : \text{String}@S \rightarrow_{\{\mathbf{P}, \mathbf{A}\}} \text{String}@S$ 
701    $\text{empty}(\mathbf{P}, \mathbf{A}, \mathbf{S}) : [\text{AuthMethod}]@(\mathbf{P}, \mathbf{A}, \mathbf{S}) \rightarrow \text{Bool}@(\mathbf{P}, \mathbf{A}, \mathbf{S})$ 
702  $\text{eapSuccess}(\mathbf{P}, \mathbf{A}, \mathbf{S}) : \text{String}@S \rightarrow (\text{String}@P \times \text{String}@A)$ 
703  $\text{eapFailure}(\mathbf{P}, \mathbf{A}, \mathbf{S}) : \text{String}@S \rightarrow (\text{String}@P \times \text{String}@A)$ 
704

```

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

709 EAP is parametric on a list of choreographies called *methods*. We use the notation for lists  
 710 in  $[\text{AuthMethod}]@(\mathbf{P}, \mathbf{A}, \mathbf{S})$  as described in Example 21, as well as the **if**  $M$  **then**  $M'$  **else**  $M''$   
 711 construct which is just syntactic sugar for the previously described **case**  $M$  **of**  $\text{Inl } x \Rightarrow$   
 712  $M'; \text{Inr } x \Rightarrow M''$ . 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 processes. 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.

```

eap1(A, S) = λmethods : [AuthMethod].
  eapAuth1(A, S) (eapIdentity1(A, S) ⊥) methods

eap2(P, S) = λmethods : [AuthMethod].
  eapAuth2(P, S) (eapIdentity2(P, S) ⊥) methods

eap3(P, A) = λmethods : [AuthMethod].
  eapAuth3(P, A) (eapIdentity3(P, 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.

```

eapAuth1(A, S) = λid : ⊥. λmethods : [AuthMethod].
  case empty1(A, S) methods of
    Inl _ ⇒ eapFailure1(A, S) ⊥
    Inr _ ⇒ &S{ok : eapSuccess1(A, S) ⊥
              ko : eapAuth1(A, S) ⊥ (snd methods)}

eapAuth2(P, S) = λid : ⊥. λmethods : [AuthMethod].
  case empty2(P, S) methods of
    Inl _ ⇒ eapFailure2(P, S) ⊥
    Inr _ ⇒ &S{ok : (eapSuccess2(P, S) ⊥)
              ko : (eapAuth2(P, S) ⊥ (snd methods))}

eapAuth3(P, A) = λid : String. λmethods : [AuthMethod].
  case empty3(P, A) methods of
    Inl _ ⇒ eapFailure3(P, A) "Try again later"
    Inr _ ⇒ case (fst methods) id of
      Inl _ ⇒ ⊕P ok (⊕A ok (eapSuccess3(P, A) "Welcome"))
      Inr _ ⇒ ⊕P ko (⊕A ko (eapAuth3(P, A) id (snd methods)))

```

Note that the implementation of the check `empty(P, A, 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.

## 766 6 Related Work

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

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

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

794 In contrast to [11] and [18],  $\text{Chor}\lambda$  presents no “hidden” barriers: coordination among  
795 processes is left to the programmer of the choreography, and EPP inserts no hidden syn-  
796 chronisations. Our EPP thus generates more concurrent and faithful implementations. It  
797 is also the first modular EPP for functional choreographic programming: changing the  
798 behaviours of some processes in a choreography requires re-running EPP only for those  
799 processes. This is important for the application of choreographic programming to DevOps  
800 (continuous integration and deployment), library management, and modularity in general.

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

813 multiter programs (in terms of Lambda 5).

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

## 823 **7 Conclusion and Future Work**

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

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

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

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

## 854 **Future Work**

855 Synchronous communication is widely adopted in theories of processes and is usually imple-  
856 mented in practice by using acknowledgements. A potential extension of Chor $\lambda$  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  $\text{Chor}\lambda$ , 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 Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.* **34**(2), 8:1–8:78 (2012). <https://doi.org/10.1145/2220365.2220367>, <https://doi.org/10.1145/2220365.2220367>
- 2 Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: Giacobazzi, R., Cousot, R. (eds.) *Procs. POPL*. pp. 263–274. ACM (2013). <https://doi.org/10.1145/2429069.2429101>
- 3 Carbone, M., Montesi, F., Schürmann, C., Yoshida, N.: Multiparty session types as coherence proofs. *Acta Informatica* **54**(3), 243–269 (2017). <https://doi.org/10.1007/s00236-016-0285-y>, <https://doi.org/10.1007/s00236-016-0285-y>
- 4 Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multi-party session. *Log. Methods Comput. Sci.* **8**(1) (2012). [https://doi.org/10.2168/LMCS-8\(1:24\)2012](https://doi.org/10.2168/LMCS-8(1:24)2012), [https://doi.org/10.2168/LMCS-8\(1:24\)2012](https://doi.org/10.2168/LMCS-8(1:24)2012)
- 5 Church, A.: A set of postulates for the foundation of logic. *Annals of Mathematics* **33**(2), 346–366 (1932), <http://www.jstor.org/stable/1968337>
- 6 Cruz-Filipe, L., Graversen, E., Lugović, L., Montesi, F., Peressotti, M.: Functional choreographic programming. In: Seidl, H., Liu, Z., Pasareanu, C.S. (eds.) *Theoretical Aspects of Computing - ICTAC 2022 - 19th International Colloquium*, Tbilisi, Georgia, September 27–29, 2022, *Proceedings. Lecture Notes in Computer Science*, vol. 13572, pp. 212–237. Springer (2022). [https://doi.org/10.1007/978-3-031-17715-6\\_15](https://doi.org/10.1007/978-3-031-17715-6_15), [https://doi.org/10.1007/978-3-031-17715-6\\_15](https://doi.org/10.1007/978-3-031-17715-6_15)
- 7 Cruz-Filipe, L., Montesi, F.: On asynchrony and choreographies. In: Bartoletti, M., Bocchi, L., Henrio, L., Knight, S. (eds.) *Proceedings 10th Interaction and Concurrency Experience, ICE@DisCoTec 2017*, Neuchâtel, Switzerland, 21–22nd June 2017. *EPTCS*, vol. 261, pp. 76–90 (2017). <https://doi.org/10.4204/EPTCS.261.8>, <https://doi.org/10.4204/EPTCS.261.8>
- 8 Cruz-Filipe, L., Montesi, F.: A core model for choreographic programming. *Theor. Comput. Sci.* **802**, 38–66 (2020). <https://doi.org/10.1016/j.tcs.2019.07.005>, <https://doi.org/10.1016/j.tcs.2019.07.005>
- 9 Cruz-Filipe, L., Montesi, F., Peressotti, M.: Certifying choreography compilation. In: Cerone, A., Ölveczky, P.C. (eds.) *Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium*, Virtual Event, Nur-Sultan, Kazakhstan, September 8–10, 2021, *Proceedings. Lecture Notes in Computer Science*, vol. 12819, pp. 115–133. Springer (2021). [https://doi.org/10.1007/978-3-030-85315-0\\_8](https://doi.org/10.1007/978-3-030-85315-0_8)
- 10 Cruz-Filipe, L., Montesi, F., Peressotti, M.: Formalising a turing-complete choreographic language in coq. In: Cohen, L., Kaliszyk, C. (eds.) *12th International Conference on Interactive Theorem Proving, ITP 2021*, June 29 to July 1, 2021, Rome, Italy (Virtual Conference). *LIPIcs*, vol. 193, pp. 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). <https://doi.org/10.4230/LIPIcs.ITP.2021.15>
- 11 Demangeon, R., Honda, K.: Nested protocols in session types. In: Koutny, M., Ulidowski, I. (eds.) *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012*, Newcastle upon Tyne, UK, September 4–7, 2012. *Proceedings. Lecture Notes in Computer Science*, vol. 7454, pp. 272–286. Springer (2012). [https://doi.org/10.1007/978-3-642-32940-1\\_20](https://doi.org/10.1007/978-3-642-32940-1_20), [https://doi.org/10.1007/978-3-642-32940-1\\_20](https://doi.org/10.1007/978-3-642-32940-1_20)



- 907 **12** Deniérou, P., Yoshida, N.: Multiparty compatibility in communicating automata: Characterisation and synthesis of global session types. In: Fomin, F.V., Freivalds, R., Kwiatkowska, M.Z., Peleg, D. (eds.) Automata, Languages, and Programming - 40th International Colloquium, ICALP 2013, Riga, Latvia, July 8-12, 2013, Proceedings, Part II. Lecture Notes in Computer Science, vol. 7966, pp. 174–186. Springer (2013). [https://doi.org/10.1007/978-3-642-39212-2\\_18](https://doi.org/10.1007/978-3-642-39212-2_18), [https://doi.org/10.1007/978-3-642-39212-2\\_18](https://doi.org/10.1007/978-3-642-39212-2_18)
- 908  
909  
910  
911  
912  
913 **13** Diffie, W., Hellman, M.E.: New directions in cryptography. *IEEE Trans. Inf. Theory* **22**(6), 644–654 (1976). <https://doi.org/10.1109/TIT.1976.1055638>, <https://doi.org/10.1109/TIT.1976.1055638>
- 914  
915  
916 **14** Fowler, S., Lindley, S., Wadler, P.: Mixing metaphors: Actors as channels and channels as actors. In: Müller, P. (ed.) 31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain. *LIPIcs*, vol. 74, pp. 11:1–11:28. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2017). <https://doi.org/10.4230/LIPIcs.ECOOP.2017.11>, <https://doi.org/10.4230/LIPIcs.ECOOP.2017.11>
- 917  
918  
919  
920  
921 **15** Giallorenzo, S., Lanese, I., Russo, D.: Chip: A choreographic integration process. In: Panetto, H., Debruyne, C., Proper, H.A., Ardagna, C.A., Roman, D., Meersman, R. (eds.) *Procs. OTM*, part II. Lecture Notes in Computer Science, vol. 11230, pp. 22–40. Springer (2018). [https://doi.org/10.1007/978-3-030-02671-4\\_2](https://doi.org/10.1007/978-3-030-02671-4_2)
- 922  
923  
924  
925 **16** Giallorenzo, S., Montesi, F., Peressotti, M.: Object-oriented choreographic programming. *CoRR abs/2005.09520* (2020), <https://arxiv.org/abs/2005.09520>
- 926  
927 **17** Giallorenzo, S., Montesi, F., Peressotti, M., Richter, D., Salvaneschi, G., Weisenburger, P.: 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>
- 928  
929  
930  
931  
932 **18** Hirsch, A.K., Garg, D.: Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.* **6**(POPL), 1–27 (2022). <https://doi.org/10.1145/3498684>, <https://doi.org/10.1145/3498684>
- 933  
934  
935 **19** Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. *J. ACM* **63**(1), 9 (2016). <https://doi.org/10.1145/2827695>, also: *POPL*, pages 273–284, 2008
- 936  
937 **20** Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniérou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. *ACM Comput. Surv.* **49**(1), 3:1–3:36 (2016). <https://doi.org/10.1145/2873052>
- 938  
939  
940  
941 **21** Jongmans, S., van den Bos, P.: A predicate transformer for choreographies - computing preconditions in choreographic programming. In: Sergey, I. (ed.) *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*. Lecture Notes in Computer Science, vol. 13240, pp. 520–547. Springer (2022). [https://doi.org/10.1007/978-3-030-99336-8\\_19](https://doi.org/10.1007/978-3-030-99336-8_19), [https://doi.org/10.1007/978-3-030-99336-8\\_19](https://doi.org/10.1007/978-3-030-99336-8_19)
- 942  
943  
944  
945  
946  
947  
948 **22** Leesatapornwongsa, T., Lukman, J.F., Lu, S., Gunawi, H.S.: TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In: *Proc. of ASPLOS*. pp. 517–530 (2016)
- 949  
950  
951 **23** Montesi, F.: *Choreographic Programming*. Ph.D. Thesis, IT University of Copenhagen (2013), <http://www.fabriziomontesi.com/files/choreographic-programming.pdf>
- 952  
953 **24** Montesi, F.: *Introduction to Choreographies*. Cambridge University Press (2023)
- 954  
955 **25** Murphy VII, T., Crary, K., Harper, R., Pfenning, F.: 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. pp. 286–295. IEEE Computer Society (2004). <https://doi.org/10.1109/LICS.2004.1319623>, <https://doi.org/10.1109/LICS.2004.1319623>
- 956  
957

- 26 Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. *Commun. ACM* **21**(12), 993–999 (1978). <https://doi.org/10.1145/359657.359659>
- 27 Vasconcelos, V.T., Gay, S.J., Ravara, A.: Type checking a multithreaded functional language with session types. *Theor. Comput. Sci.* **368**(1-2), 64–87 (2006). <https://doi.org/10.1016/j.tcs.2006.06.028>, <https://doi.org/10.1016/j.tcs.2006.06.028>
- 28 Vollbrecht, J., Carlson, J.D., Blunk, L., Aboba, D.B.D., Levkowetz, H.: Extensible Authentication Protocol (EAP). RFC 3748 (Jun 2004). <https://doi.org/10.17487/RFC3748>, <https://rfc-editor.org/rfc/rfc3748.txt>
- 29 Weisenburger, P., Wirth, J., Salvaneschi, G.: A survey of multitier programming. *ACM Comput. Surv.* **53**(4), 81:1–81:35 (2020). <https://doi.org/10.1145/3397495>, <https://doi.org/10.1145/3397495>
- 30 Yoshida, N., Hu, R., Neykova, R., Ng, N.: The scribble protocol language. In: Abadi, M., Lluch-Lafuente, A. (eds.) *Trustworthy Global Computing - 8th International Symposium, TGC 2013, Buenos Aires, Argentina, August 30-31, 2013, Revised Selected Papers. Lecture Notes in Computer Science*, vol. 8358, pp. 22–41. Springer (2013). [https://doi.org/10.1007/978-3-319-05119-2\\_3](https://doi.org/10.1007/978-3-319-05119-2_3), [https://doi.org/10.1007/978-3-319-05119-2\\_3](https://doi.org/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  $M$ ,  $\text{fv}(M)$  are defined as:*

$$\begin{aligned}
 \text{fv}(N \ N') &= \text{fv}(N) \cup \text{fv}(N') & \text{fv}(\text{select}_{q,p} \ l \ M) &= \text{fv}(M) \\
 \text{fv}(x) &= x & \text{fv}(\lambda x : T.N) &= \text{fv}(N) \setminus \{x\} \\
 \text{fv}(()@p) &= \emptyset & \text{fv}(\text{com}_{q,p}) &= \emptyset \\
 \text{fv}(f(\vec{p})) &= \emptyset & \text{fv}(\text{Pair } V \ V') &= \text{fv}(V) \cup \text{fv}(V') \\
 \text{fv}(\text{case } N \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } y \Rightarrow M') &= \text{fv}(N) \cup (\text{fv}(M) \setminus \{x\}) \cup (\text{fv}(M') \setminus \{y\}) \\
 \text{fv}(\text{fst}) &= \text{fv}(\text{snd}) = \emptyset & \text{fv}(\text{Inl } V) &= \text{fv}(\text{Inr } V) = \text{fv}(V)
 \end{aligned}$$

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

$$\begin{aligned}
 \text{bv}(N \ N') &= \text{bv}(N) \cup \text{bv}(N') & \text{bv}(\text{select}_{q,p} \ l \ M) &= \text{bv}(M) \\
 \text{bv}(x) &= \emptyset & \text{bv}(\lambda x : T.N) &= \text{bv}(N) \cup \{x\} \\
 \text{bv}(()@p) &= \emptyset & \text{bv}(\text{com}_{q,p}) &= \emptyset \\
 \text{bv}(f(\vec{p})) &= \emptyset & \text{bv}(\text{Pair } V \ V') &= \text{bv}(V) \cup \text{bv}(V') \\
 \text{bv}(\text{case } N \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } y \Rightarrow M') &= \text{bv}(N) \cup \text{bv}(M) \cup \{x\} \cup (\text{bv}(M') \cup \{y\}) \\
 \text{bv}(\text{fst}) &= \text{bv}(\text{snd}) = \emptyset & \text{bv}(\text{Inl } V) &= \text{bv}(\text{Inr } V) = \text{bv}(V)
 \end{aligned}$$

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

$$\begin{aligned}
 \text{pn}(t@R) &= \vec{R} & \text{pn}(T \rightarrow_\rho T') &= \text{pn}(T) \cup \text{pn}(T') \cup \rho \\
 \text{pn}(()@R) &= \{R\} & \text{pn}(T + T') &= \text{pn}(T \times T') = \text{pn}(T) \cup \text{pn}(T')
 \end{aligned}$$

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

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

$$\begin{array}{c}
\frac{\Theta'; \Sigma; \Gamma, x : T \vdash M : T' \quad \rho \cup \text{pn}(T) \cup \text{pn}(T') = \Theta' \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \lambda x : T. M : T \rightarrow_{\rho} T'} [\text{TAbs}] \\
\frac{x : T \in \Gamma \quad \text{pn}(T) \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash x : T} [\text{TVar}] \quad \frac{\Theta; \Sigma; \Gamma \vdash N : T \rightarrow_{\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 \mathbf{case} N \mathbf{of} \mathbf{Inl} x \Rightarrow M'; \mathbf{Inr} x' \Rightarrow M'' : T} [\text{TCASE}] \\
\frac{\Theta; \Sigma; \Gamma \vdash M : T \quad \mathbf{q}, \mathbf{p} \in \Theta}{\Theta; \Sigma; \Gamma \vdash \mathbf{select}_{\mathbf{q}, \mathbf{p}} l M : T} [\text{TSEL}] \\
\frac{f(\vec{p}) : T \in \Gamma \quad \text{pn}(T) \subseteq \vec{p} \subseteq \Theta \quad \|\vec{p}\| = \|\vec{p}'\| \quad \text{distinct}(\vec{p})}{\Theta; \Sigma; \Gamma \vdash f(\vec{p}) : T[\vec{p}' := \vec{p}]} [\text{TFUN}] \\
\frac{\mathbf{p} \in \Theta}{\Theta; \Sigma; \Gamma \vdash ()@_{\mathbf{p}} : ()@_{\mathbf{p}}} [\text{TUNIT}] \quad \frac{\mathbf{q}, \mathbf{p} \in \Theta \quad \text{pn}(T) = \mathbf{q}}{\Theta; \Sigma; \Gamma \vdash \mathbf{com}_{\mathbf{q}, \mathbf{p}} : T \rightarrow_{\emptyset} T[\mathbf{q} := \mathbf{p}]} [\text{TCom}] \\
\frac{\Theta; \Sigma; \Gamma \vdash V : T \quad \Theta; \Sigma; \Gamma \vdash V' : T'}{\Theta; \Sigma; \Gamma \vdash \mathbf{Pair} V V' : (T \times T')} [\text{TPAIR}] \\
\frac{\text{pn}(T \times T') \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \mathbf{fst} : (T \times T') \rightarrow_{\emptyset} T} [\text{TPROJ1}] \quad \frac{\text{pn}(T \times T') \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \mathbf{snd} : (T \times T') \rightarrow_{\emptyset} T'} [\text{TPROJ2}] \\
\frac{\Theta; \Sigma; \Gamma \vdash V : T \quad \text{pn}(T + T') \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \mathbf{Inl} V : (T + T')} [\text{TINL}] \quad \frac{\Theta; \Sigma; \Gamma \vdash V : T' \quad \text{pn}(T + T') \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \mathbf{Inr} V : (T + T')} [\text{TINR}] \\
\frac{\Theta; \Sigma; \Gamma \vdash M : t@_{\vec{p}} \quad t@_{\vec{p}'} =_{\Sigma} T \quad \|\vec{p}\| = \|\vec{p}'\| \quad \text{distinct}(\vec{p})}{\Theta; \Sigma; \Gamma \vdash M : T[\vec{p}' := \vec{p}]} [\text{TEQ}] \\
\frac{\forall f(\vec{p}) \in \text{dom}(D) : \quad f(\vec{p}) : T \in \Gamma \quad \vec{p}; \Sigma; \Gamma \vdash D(f(\vec{p})) : T \quad \text{distinct}(\vec{p}) \quad \vec{p} \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash D} [\text{TDEFS}]
\end{array}$$

■ **Figure 7** Full set of typing rules for Chor $\lambda$ .

$$\begin{array}{l}
993 \quad \text{pn}(\mathbf{Inl} V) = (\text{pn} \mathbf{Inr} V) = \text{pn}(V) \\
994 \quad \text{pn}(\mathbf{Pair} V V') = \text{pn}(V) \cup \text{pn}(V') \\
995 \quad \text{pn}(\mathbf{fst}) = \text{pn}(\mathbf{snd}) = \emptyset \\
996 \quad \text{pn}(\mathbf{com}_{\mathbf{p}, \mathbf{q}}) = \{\mathbf{p}, \mathbf{q}\} \\
997
\end{array}$$

998 ▶ **Definition 33.** We define the set of synchronising processes of a choreography  $M$ ,  $\text{spn}(M)$ ,  
999 by recursion on the structure of  $M$ :

$$\begin{array}{l}
1000 \quad \text{spn}(\mathbf{com}_{S, R}) = \{S, R\}, \text{spn}(\mathbf{select}_{S, R} l M) = \{S, R\} \cup \text{spn}(M), \\
1001 \quad \text{spn}(f(\vec{R})) = \vec{R}, \text{ and homomorphically on all other cases.}
\end{array}$$

1002 ▶ **Definition 34 (Merging).** Given two behaviours  $B$  and  $B'$ ,  $B \sqcup B'$  is defined as follows.

$$\begin{array}{l}
1003 \quad B_1 B_2 \sqcup B'_1 B'_2 = (B_1 \sqcup B'_1) (B_2 \sqcup B'_2) \\
1004 \quad \mathbf{case} B_1 \mathbf{of} \mathbf{Inl} x \Rightarrow B_2; \mathbf{Inr} y \Rightarrow B_3 \sqcup \mathbf{case} B'_1 \mathbf{of} \mathbf{Inl} x \Rightarrow B'_2; \mathbf{Inr} y \Rightarrow B'_3 = \\
1005 \quad \mathbf{case} (B_1 \sqcup B'_1) \mathbf{of} \mathbf{Inl} x \Rightarrow (B_2 \sqcup B'_2); \mathbf{Inr} y \Rightarrow (B_3 \sqcup B'_3) \\
1006 \quad \oplus_{\mathbf{p}} \ell B \sqcup \oplus_{\mathbf{p}} \ell B' = \oplus_{\mathbf{p}} \ell (B \sqcup B') \\
1007 \quad \&\{\ell_i : B_i\}_{i \in I} \sqcup \&\{\ell_j : B'_j\}_{j \in J} = \&(\{\ell_k : B_k \sqcup B'_k\}_{k \in I \cap J} \cup \{\ell_i : B_i\}_{i \in I \setminus J} \cup \{\ell_j : B'_j\}_{j \in J \setminus I})
\end{array}$$

$$\begin{array}{c}
\frac{\text{fv}(V) \cap \text{bv}(M) = \emptyset}{\lambda x : T.M \ V \xrightarrow{\tau, \emptyset}_D M[x := V]} \text{[APPABS]} \quad \frac{M \xrightarrow{\ell, P}_D M'}{\lambda x : T.M \xrightarrow{\lambda, P}_D \lambda x : T.M'} \text{[INABS]} \\
\\
\frac{M \xrightarrow{\ell, P}_D M' \quad \ell = \lambda \Rightarrow P \cap \text{pn}(N) = \emptyset}{M \ N \xrightarrow{\tau, P}_D M' \ N} \text{[APP1]} \\
\\
\frac{N \xrightarrow{\tau, P}_D N'}{V \ N \xrightarrow{\tau, P}_D V \ N'} \text{[APP2]} \quad \frac{N \xrightarrow{\tau, P}_D N' \quad P \cap \text{pn}(M) = \emptyset}{M \ N \xrightarrow{\tau, P}_D M \ N'} \text{[APP3]} \\
\\
\frac{N \xrightarrow{\tau, P}_D N'}{\text{case } N \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } x' \Rightarrow M' \xrightarrow{\tau, P}_D \text{case } N' \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } x' \Rightarrow M'} \text{[CASE]} \\
\\
\frac{M_1 \xrightarrow{\ell, P}_D M'_1 \quad M_2 \xrightarrow{\ell, P}_D M'_2 \quad P \cap \text{pn}(N) = \emptyset}{\text{case } N \text{ of } \text{Inl } x \Rightarrow M_1; \text{Inr } x' \Rightarrow M_2 \xrightarrow{\ell, P}_D \text{case } N \text{ of } \text{Inl } x \Rightarrow M'_1; \text{Inr } x' \Rightarrow M'_2} \text{[INCASE]} \\
\\
\frac{}{\text{case Inl } V \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } x' \Rightarrow M' \xrightarrow{\tau, \emptyset}_D M[x := V]} \text{[CASEL]} \\
\\
\frac{}{\text{case Inr } V \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } x' \Rightarrow M' \xrightarrow{\tau, \emptyset}_D M'[x' := V]} \text{[CASER]} \\
\\
\frac{}{\text{fst Pair } V \ V' \xrightarrow{\tau, \emptyset}_D V} \text{[PROJ1]} \quad \frac{}{\text{snd Pair } V \ V' \xrightarrow{\tau, \emptyset}_D V'} \text{[PROJ2]} \\
\\
\frac{D(f(\vec{p}')) = M}{f(\vec{p}) \xrightarrow{\tau, \emptyset}_D M[\vec{p}' := \vec{p}]} \text{[DEF]} \\
\\
\frac{\text{fv}(V) = \emptyset}{\text{com}_{q,p} \ V \xrightarrow{\tau, \{q,p\}}_D V[q := p]} \text{[COM]} \quad \frac{}{\text{select}_{q,p} \ l \ M \xrightarrow{\tau, \{q,p\}}_D M} \text{[SEL]} \\
\\
\frac{M \xrightarrow{\ell, P}_D M' \quad P \cap \{q, p\} = \emptyset}{\text{select}_{q,p} \ \ell \ M \xrightarrow{\ell, P}_D \text{select}_{q,p} \ \ell \ M'} \text{[INSEL]} \quad \frac{M \rightsquigarrow^* N \quad N \xrightarrow{\tau, P}_D N'}{M \xrightarrow{\tau, P}_D M'} \text{[STR]}
\end{array}$$

■ **Figure 8** Semantics of Chor $\lambda$

$$\begin{array}{c}
\frac{x \notin \text{fv}(M')}{((\lambda x : T.M) N) M' \rightsquigarrow (\lambda x : T.(M M')) N} \text{[R-ABSR]} \\
\frac{x \notin \text{fv}(M') \quad \text{spn}(M') \cap \text{pn}(N) = \emptyset}{M' ((\lambda x : T.M) N) \rightsquigarrow (\lambda x : T.(M' M)) N} \text{[R-ABSL]} \\
\frac{x, x' \notin \text{fv}(M)}{(\text{case } N \text{ of } \text{Inl } x \Rightarrow M_1; \text{Inr } x' \Rightarrow M_2) M \rightsquigarrow \text{case } N \text{ of } \text{Inl } x \Rightarrow (M_1 M); \text{Inr } x' \Rightarrow (M_2 M)} \text{[R-CASER]} \\
\frac{x, x' \notin \text{fv}(M) \quad \text{spn}(M) \cap \text{pn}(N) = \emptyset}{M (\text{case } N \text{ of } \text{Inl } x \Rightarrow M_1; \text{Inr } x' \Rightarrow M_2) \rightsquigarrow \text{case } N \text{ of } \text{Inl } x \Rightarrow (M M_1); \text{Inr } x' \Rightarrow (M M_2)} \text{[R-CASEL]} \\
\frac{}{(\text{select}_{q,p} l N) M \rightsquigarrow \text{select}_{q,p} l (N M)} \text{[R-SELR]} \\
\frac{\text{spn}(M) \cap \text{pn}(N) = \emptyset}{M (\text{select}_{q,p} l N) \rightsquigarrow \text{select}_{q,p} l (M N)} \text{[R-SELL]} \\
\frac{y \text{ fresh for } M}{\lambda x : T.M \rightsquigarrow \lambda y : T.M[x := y]} \text{[R-ALPH]}
\end{array}$$

■ **Figure 9** Rewriting of Chorλ.

$$\begin{array}{l}
1008 \quad x \sqcup x = x \quad \lambda x : T.B \sqcup \lambda x : T.B' = \lambda x : T.(B \sqcup B') \\
1009 \quad \text{fst} \sqcup \text{fst} = \text{fst} \quad \text{snd} \sqcup \text{snd} = \text{snd} \\
1010 \quad \text{Inl } L \sqcup \text{Inl } L' = \text{Inl } (L \sqcup L') \quad \text{Inr } L \sqcup \text{Inr } L' = \text{Inr } (L \sqcup L') \\
1011 \quad \text{Pair } L_1 L_2 \sqcup \text{Pair } L'_1 L'_2 = \text{Pair } (L_1 \sqcup L'_1) (L_2 \sqcup L'_2) \quad f \sqcup f = f \\
1012 \quad \text{recv}_p \sqcup \text{recv}_p = \text{recv}_p \quad \text{send}_p \sqcup \text{send}_p = \text{send}_p \quad \perp \sqcup \perp = \perp
\end{array}$$

1013 ▶ **Definition 35** (Context). We define a context  $C[]$  in Chorλ as follows:

$$\begin{array}{l}
1014 \quad C[] ::= [] \mid M C[] \mid C[] M \mid \text{select}_{p,p} l C[] \mid \text{case } C[] \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } x \Rightarrow M \\
\quad \mid \text{case } M \text{ of } \text{Inl } x \Rightarrow C[]; \text{Inr } x \Rightarrow M \mid \text{case } M \text{ of } \text{Inl } x \Rightarrow M; \text{Inr } x' \Rightarrow C[] \\
\quad \mid \lambda x : T.C[]
\end{array}$$

$$\begin{array}{c}
\frac{\text{fv}(L) = \emptyset}{\text{send}_p L \xrightarrow{\text{send}_p L} \perp} [\text{NSEND}] \quad \frac{}{\text{recv}_p \perp \xrightarrow{\text{recv}_p L} L} [\text{NRECV}] \\
\frac{B \xrightarrow{\text{send}_q L} B'_1 \quad B_2 \xrightarrow{\text{recv}_p L} B'_2}{q[B_1] \mid p[B_2] \xrightarrow{\tau_{q,p}} q[B'_1] \mid p[B'_2]} [\text{NCOM}] \\
\frac{}{\oplus_p l B \xrightarrow{\oplus_p l} B} [\text{NCHO}] \quad \frac{}{\&_p \{\ell_1 : B_1, \dots, \ell_n : B_n\} \xrightarrow{\&_p \ell_i} B_i} [\text{NOFF}] \\
\frac{B_i \xrightarrow{\mu} B'_i \text{ for } 1 \leq i \leq n \quad \mu \in \{\tau, \lambda\}}{\&_p \{\ell_1 : B_1, \dots, \ell_n : B_n\} \xrightarrow{\mu} \&_p \{\ell_1 : B'_1, \dots, \ell_n : B'_n\}} [\text{NOFF2}] \\
\frac{B \xrightarrow{\mu} B' \quad \mu \in \{\tau, \lambda\}}{\oplus_p l B \xrightarrow{\mu} \oplus_p l B'} [\text{NCHO2}] \quad \frac{B_1 \xrightarrow{\oplus_p \ell} B'_1 \quad B_2 \xrightarrow{\&_q \ell} B'_2}{q[B_1] \mid p[B_2] \xrightarrow{\tau_{q,p}} q[B'_1] \mid p[B'_2]} [\text{NSEL}] \\
\frac{}{(\lambda x : T.B) L \xrightarrow{\tau} B[x := L]} [\text{NABSApP}] \quad \frac{B \xrightarrow{\mu} B' \quad \mu \in \{\tau, \lambda\}}{\lambda x : T.B \xrightarrow{\lambda} \lambda x : T.B'} [\text{NINABS}] \\
\frac{B \xrightarrow{\mu} B'' \quad \text{if } \mu = \lambda \text{ then } \mu' = \tau \text{ else } \mu' = \mu}{B B' \xrightarrow{\mu'} B'' B'} [\text{NAPP1}] \\
\frac{B \xrightarrow{\mu} B'}{L B \xrightarrow{\mu} L B'} [\text{NAPP2}] \quad \frac{B' \xrightarrow{\tau} B''}{B B' \xrightarrow{\tau} B B''} [\text{NAPP3}] \\
\frac{B \xrightarrow{\mu} B''}{\text{case } B \text{ of } \text{Inl } x \Rightarrow B'; \text{Inr } x' \Rightarrow B'' \xrightarrow{\mu} \text{case } B''' \text{ of } \text{Inl } x \Rightarrow B'; \text{Inr } x' \Rightarrow B''} [\text{NCASE}] \\
\frac{B_1 \xrightarrow{\mu} B'_1 \quad B_2 \xrightarrow{\mu} B'_2 \quad \mu \in \{\lambda, \tau\}}{\text{case } B \text{ of } \text{Inl } x \Rightarrow B_1; \text{Inr } x' \Rightarrow B_2 \xrightarrow{\mu} \text{case } B \text{ of } \text{Inl } x \Rightarrow B'_1; \text{Inr } x' \Rightarrow B'_2} [\text{NCASE2}] \\
\frac{}{\text{case Inl } L \text{ of } \text{Inl } x \Rightarrow B; \text{Inr } x' \Rightarrow B' \xrightarrow{\tau} B[x := L]} [\text{NCASEL}] \\
\frac{}{\text{case Inr } L \text{ of } \text{Inl } x \Rightarrow B; \text{Inr } x' \Rightarrow B' \xrightarrow{\tau} B'[x' := L]} [\text{NCASER}] \\
\frac{}{\text{fst Pair } L L' \xrightarrow{\tau} L} [\text{NPROJ1}] \quad \frac{}{\text{snd Pair } L L' \xrightarrow{\tau} L'} [\text{NPROJ2}] \\
\frac{B \xrightarrow{\tau} B'}{p[B] \xrightarrow{\tau_p} p[B']} [\text{NPRO}] \quad \frac{\mathcal{N} \xrightarrow{\tau_p} \mathcal{N}''}{\mathcal{N} \mid \mathcal{N}' \xrightarrow{\tau_p} \mathcal{N}'' \mid \mathcal{N}'} [\text{NPAR}] \\
\frac{D(f(\vec{p}')) = B}{f(\vec{p}) \xrightarrow{\tau} B[\vec{p}' := \vec{p}]} [\text{NFUN}] \quad \frac{B \rightsquigarrow^* B'' \quad B'' \xrightarrow{\mu} B'}{B \xrightarrow{\mu} B'} [\text{NSTR}]
\end{array}$$

■ **Figure 10** Semantics of networks.

$$\begin{array}{c}
\frac{}{((\lambda x.B) B') B'' \rightsquigarrow (\lambda x.B B'') B')} \text{[LR-ABSR]} \\
\frac{\text{pn}(B'') = \emptyset}{B'' ((\lambda x.B) B') \rightsquigarrow (\lambda x.B'' B) B'} \text{[LR-ABSL]} \\
\frac{}{(\text{case } B \text{ of } \text{Inl } x \Rightarrow B_1; \text{Inr } x \Rightarrow B_2) B' \rightsquigarrow \text{case } B \text{ of } \text{Inl } x \Rightarrow (B_1 B'); \text{Inr } x \Rightarrow (B_2 B')} \text{[LR-CASER]} \\
\frac{\text{pn}(B') = \emptyset}{B' (\text{case } B \text{ of } \text{Inl } x \Rightarrow B_1; \text{Inr } x \Rightarrow B_2) \rightsquigarrow \text{case } B \text{ of } \text{Inl } x \Rightarrow (B' B_1); \text{Inr } x \Rightarrow (B' B_2)} \text{[LR-CASEL]} \\
\frac{\text{pn}(B) = \emptyset}{B (\&_p\{l_1 : B_1, \dots, l_n : B_n\}) \rightsquigarrow \&_p\{l_1 : B B_1, \dots, l_n : B B_n\}} \text{[LR-OFFL]} \\
\frac{}{(\&_p\{l_1 : B_1, \dots, l_n : B_n\}) B \rightsquigarrow \&_p\{l_1 : B_1 B, \dots, l_n : B_n B\}} \text{[LR-OFFR]} \\
\frac{\text{pn}(B') = \emptyset}{B' (\oplus_p l B) \rightsquigarrow \oplus_p l (B' B)} \text{[LR-CHOL]} \quad \frac{}{(\oplus_p l B) B' \rightsquigarrow \oplus_p l (B B')} \text{[LR-CHOR]} \\
\frac{}{\perp \perp \rightsquigarrow \perp} \text{[LR-BOTM]} \quad \frac{y \text{ fresh for } B}{\lambda x : T.B \rightsquigarrow \lambda y : T.B[x := y]} \text{[LR-ALPH]}
\end{array}$$

■ **Figure 11** Rewriting of processes.

$$\begin{array}{c}
\frac{\Sigma; \Gamma \vdash B : T}{\Sigma; \Gamma \vdash \oplus_p l B : T} \text{[NTCHOR]} \quad \frac{\Sigma; \Gamma \vdash B_i : T \text{ for } 1 \leq i \leq n}{\Sigma; \Gamma \vdash \&_p\{\ell_1 : B_1, \dots, \ell_n : B_n\} : T} \text{[NTOFF]} \\
\frac{}{\Sigma; \Gamma \vdash \text{send}_p : T \rightarrow \perp} \text{[NTSEND]} \quad \frac{}{\Sigma; \Gamma \vdash \text{recv}_p : \perp \rightarrow T} \text{[NTRECV]} \\
\frac{\Sigma; \Gamma, x : T \vdash B : T'}{\Sigma; \Gamma \vdash \lambda x : T.B : T \rightarrow T'} \text{[NTABS]} \quad \frac{x : T \in \Gamma}{\Sigma; \Gamma \vdash x : T} \text{[NTVAR]} \\
\frac{\Sigma; \Gamma \vdash B : T \rightarrow T' \quad \Sigma; \Gamma \vdash B : T}{\Sigma; \Gamma \vdash B B' : T'} \text{[NTAPP]} \quad \frac{\Sigma; \Gamma \vdash B : \perp \quad \Sigma; \Gamma \vdash B' : \perp}{\Sigma; \Gamma \vdash B B' : \perp} \text{[NTAPP2]} \\
\frac{\Sigma; \Gamma \vdash B : T_1 + T_2 \quad \Sigma; \Gamma, x : T_1 \vdash B' : T \quad \Sigma; \Gamma, x' : T_2 \vdash B'' : T}{\Sigma; \Gamma \vdash \text{case } B \text{ of } \text{Inl } x \Rightarrow B'; \text{Inr } x' \Rightarrow B'' : T} \text{[NTCASE]} \\
\frac{f : T \in \Gamma}{\Sigma; \Gamma \vdash f : T} \text{[NTDEF]} \quad \frac{}{\Sigma; \Gamma \vdash () : ()} \text{[NTUNIT]} \quad \frac{}{\Sigma; \Gamma \vdash \perp : \perp} \text{[NTBOTM]} \\
\frac{}{\Sigma; \Gamma \vdash \text{Pair} : T \rightarrow T' \rightarrow (T \times T')} \text{[NTPAIR]} \\
\frac{}{\Sigma; \Gamma \vdash \text{fst} : (T \times T') \rightarrow T} \text{[NTPROJ1]} \quad \frac{}{\Sigma; \Gamma \vdash \text{snd} : (T \times T') \rightarrow T'} \text{[NTPROJ2]} \\
\frac{\Sigma; \Gamma \vdash B : T' \quad \{T = T', T' = T\} \cap \Sigma \neq \emptyset}{\Sigma; \Gamma \vdash B : T} \text{[NTEQ]} \\
\frac{\forall f \in \text{dom}(\mathbb{D}) \quad f : T \in \Gamma \quad \Sigma; \Gamma \vdash \mathbb{D}(f) : T}{\Sigma; \Gamma \vdash \mathbb{D}} \text{[NTDEFS]}
\end{array}$$

■ **Figure 12** Typing rules for behaviours.

Choreographies:

$$\begin{aligned}
\llbracket M \ N \rrbracket_p &= \begin{cases} \llbracket M \rrbracket_p \ \llbracket N \rrbracket_p & \text{if } p \in \text{pn}(\text{type}(M)) \text{ or } p \in \text{pn}(M) \cap \text{pn}(N) \\ \perp & \text{if } \llbracket M \rrbracket_p = \llbracket N \rrbracket_p = \perp \\ \llbracket M \rrbracket_p & \text{if } \llbracket N \rrbracket_p = \perp \\ \llbracket N \rrbracket_p & \text{otherwise} \end{cases} \\
\llbracket \lambda x : T.M \rrbracket_p &= \begin{cases} \lambda x. \llbracket M \rrbracket_p & \text{if } p \in \text{pn}(\text{type}(\lambda x : T.M)) \\ \perp & \text{otherwise} \end{cases} \\
\llbracket \text{case } M \text{ of } \text{Inl } x \Rightarrow N; \text{Inr } x' \Rightarrow N' \rrbracket_p &= \\
&\begin{cases} \text{case } \llbracket M \rrbracket_p \text{ of } \text{Inl } x \Rightarrow \llbracket N \rrbracket_p; \text{Inr } x' \Rightarrow \llbracket N' \rrbracket_p & \text{if } p \in \text{pn}(\text{type}(M)) \\ \llbracket M \rrbracket_p & \text{if } \llbracket N \rrbracket_p = \llbracket N' \rrbracket_p = \perp \\ \llbracket N \rrbracket_p \sqcup \llbracket N' \rrbracket_p & \text{if } \llbracket M \rrbracket_p = \perp \\ (\lambda x'' : \perp. \llbracket N \rrbracket_p \sqcup \llbracket N' \rrbracket_p) \ \llbracket M \rrbracket_p & \text{otherwise, for some } x'' \notin \text{fv}(N) \cup \text{fv}(N') \end{cases} \\
\llbracket \text{select}_{q,q'} \ell \ M \rrbracket_p &= \begin{cases} \oplus_{q'} \ell \ \llbracket M \rrbracket_p & \text{if } p = q \neq q' \\ \&_q \{ \ell : \llbracket M \rrbracket_p \} & \text{if } p = q' \neq q \\ \llbracket M \rrbracket_p & \text{otherwise} \end{cases} \\
\llbracket \text{com}_{q,q'} \rrbracket_p &= \begin{cases} \lambda x.x & \text{if } p = q = q' \\ \text{send}_{q'} & \text{if } p = q \neq q' \\ \text{recv}_q & \text{if } p = q' \neq q \\ \perp & \text{otherwise} \end{cases} \\
\llbracket () @ q \rrbracket_p &= \begin{cases} () & \text{if } q = p \\ \perp & \text{otherwise} \end{cases} \quad \llbracket x \rrbracket_p = \begin{cases} x & \text{if } p \in \text{pn}(\text{type}(x)) \\ \perp & \text{otherwise} \end{cases} \\
\llbracket f(\vec{p}) \rrbracket_p &= \begin{cases} f_i(p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n) & \text{if } \vec{p} = p_1, \dots, p_{i-1}, p, p_{i+1}, \dots, p_n \\ \perp & \text{otherwise} \end{cases} \\
\llbracket \text{Pair } V \ V' \rrbracket_p &= \begin{cases} \text{Pair } \llbracket V \rrbracket_p \ \llbracket V' \rrbracket_p & \text{if } p \in \text{pn}(\text{type}(V) \times \text{type}(V')) \\ \perp & \text{otherwise} \end{cases} \\
\llbracket \text{fst} \rrbracket_p &= \begin{cases} \text{fst} & \text{if } p \in \text{pn}(\text{type}(\text{fst})) \\ \perp & \text{otherwise} \end{cases} \quad \llbracket \text{snd} \rrbracket_p = \begin{cases} \text{snd} & \text{if } p \in \text{pn}(\text{type}(\text{snd})) \\ \perp & \text{otherwise} \end{cases} \\
\llbracket \text{Inl } V \rrbracket_p &= \begin{cases} \llbracket V \rrbracket_p & \text{if } p \in \text{pn}(\text{type}(\text{Inl } V)) \\ \perp & \text{otherwise} \end{cases} \quad \llbracket \text{Inr } V \rrbracket_p = \begin{cases} \llbracket V \rrbracket_p & \text{if } p \in \text{pn}(\text{type}(\text{Inr } V)) \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

Types:

$$\begin{aligned}
\llbracket T \rightarrow_\rho T' \rrbracket_p &= \begin{cases} \llbracket T \rrbracket_p \rightarrow \llbracket T' \rrbracket_p & \text{if } p \in \rho \cup \text{pn}(T) \cup \text{pn}(T') \\ \perp & \text{otherwise} \end{cases} \quad \llbracket () @ q \rrbracket_p = \begin{cases} () & \text{if } q = p \\ \perp & \text{otherwise} \end{cases} \\
\llbracket T \times T' \rrbracket_p &= \begin{cases} \llbracket T \rrbracket_p \times \llbracket T' \rrbracket_p & \text{if } p \in \text{pn}(T \times T') \\ \perp & \text{otherwise} \end{cases} \quad \llbracket T + T' \rrbracket_p = \begin{cases} \llbracket T \rrbracket_p + \llbracket T' \rrbracket_p & \text{if } p \in \text{pn}(T + T') \\ \perp & \text{otherwise} \end{cases} \\
\llbracket t @ \vec{p} \rrbracket_p &= \begin{cases} t_i & \text{if } \vec{p} = p_1, \dots, p_{i-1}, p, p_{i+1}, \dots, p_n \\ \perp & \text{otherwise} \end{cases}
\end{aligned}$$

Definitions:

$$\llbracket D \rrbracket = \{ f_i(p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n) \mapsto \llbracket D(f(p_1, \dots, p_n)) \rrbracket_{p_i} \mid f(p_1, \dots, p_n) \in \text{dom}(D) \}$$

■ **Figure 13** Projecting Chor $\lambda$  onto a process



## 1015 A.1 Proof of Theorem 25

1016 **Proof of Lemma 5.** Straightforward from the network semantics.  $\blacktriangleleft$

1017  $\blacktriangleright$  **Lemma 36.** *Given a value  $V$ , if  $\Theta; \Sigma; \Gamma \vdash V : T$  and  $\llbracket V \rrbracket$  is defined then for any process  $p$*   
 1018 *in  $\text{pn}(V)$ ,  $\llbracket V \rrbracket_p = L$ .*

1019 **Proof.** Straightforward from the projection rules.  $\blacktriangleleft$

1020  $\blacktriangleright$  **Lemma 37.** *Given a type  $T$ , for any process  $p \notin \text{pn}(T)$ ,  $\llbracket T \rrbracket_p = \perp$ .*

1021 **Proof.** Straightforward from induction on  $T$ .  $\blacktriangleleft$

1022  $\blacktriangleright$  **Lemma 38.** *Given a value  $V$ , for any process  $p \notin \text{pn}(\text{type}(V))$ , if  $\llbracket V \rrbracket_p$  is defined then*  
 1023  *$\llbracket V \rrbracket_p = \perp$ .*

1024 **Proof.** Follows from Lemmas 36 and 37 and the projection rules.  $\blacktriangleleft$

1025  $\blacktriangleright$  **Lemma 39.** *If  $M \rightsquigarrow M'$  and  $M \xrightarrow{\tau, P}_D M''$  and  $\llbracket M \rrbracket$  is defined then  $M' \xrightarrow{\tau, P}_D M'''$  such*  
 1026 *that  $M'' \rightsquigarrow^* M'''$*

1027 **Proof.** Follows from case analysis on  $M \rightsquigarrow M'$ .  $\blacktriangleleft$

1028  $\blacktriangleright$  **Lemma 40.** *If  $M \rightsquigarrow M'$  then for any process  $p$ ,  $\llbracket M \rrbracket_p \rightsquigarrow \cup \xrightarrow{\tau}^* B$  such that  $B \equiv \llbracket M' \rrbracket_p$*

1029 **Proof.** Follows from case analysis on  $M \rightsquigarrow M'$ .  $\blacktriangleleft$

1030 **Proof of Theorem 25.** We prove this by structural induction on  $M \xrightarrow{\tau, P}_D M'$ .

1031  $\blacksquare$  Assume  $M = \lambda x : T. N \ V$  and  $M' = N[x := V]$ . Then for any process  $p \in \text{pn}(\text{type}(\lambda x : T. N))$ , we have  $\llbracket M \rrbracket_p = (\lambda x : \llbracket T \rrbracket_p. \llbracket N \rrbracket_p) \llbracket V \rrbracket_p$  and  $\llbracket M' \rrbracket_p = \llbracket N \rrbracket_p[x := \llbracket V \rrbracket_p]$ , and for  
 1032 any  $p' \notin \text{pn}(\text{type}(\lambda x : T. N))$ , we have  $p' \notin \text{pn}(\text{type}(V))$  and therefore  $\llbracket M \rrbracket_{p'} = \llbracket M' \rrbracket_{p'} =$   
 1033  $\perp$ . We therefore get  $p[\llbracket M \rrbracket_p] \xrightarrow{\tau}_{[D]} \llbracket M' \rrbracket_p$  for all  $p \in \text{pn}(\text{type}(\lambda x : T. N))$  and define  
 1034  $\mathcal{N} = \prod_{p \in \text{pn}(\text{type}(\lambda x : T. N))} p[\llbracket M' \rrbracket_p] \mid \prod_{p' \notin \text{pn}(\text{type}(\lambda x : T. N))} p'[\perp]$  and the result follows.  
 1035

1036  $\blacksquare$  Assume  $M = N \ M''$ ,  $M' = N' \ M''$ , and  $N \xrightarrow{\tau, P}_D N'$ . Then for any process  $p \in$   
 1037  $\text{pn}(\text{type}(N))$ ,  $\llbracket M \rrbracket_p = \llbracket N \rrbracket_p \llbracket M'' \rrbracket_p$  and  $\llbracket M' \rrbracket_p = \llbracket N' \rrbracket_p \llbracket M'' \rrbracket_p$ . For any process  $p'$  such  
 1038 that  $\llbracket N \rrbracket_{p'} = \llbracket M'' \rrbracket_{p'} = \perp$ , by induction we have  $\llbracket N' \rrbracket_{p'} = \perp$ , and therefore  $\llbracket M \rrbracket_{p'} =$   
 1039  $\llbracket M' \rrbracket_{p'} = \perp$ . For any other process  $p''$  such that  $\llbracket N \rrbracket_{p''} = \perp$ , by induction we get  
 1040  $\llbracket N' \rrbracket_{p''} = \perp$  and therefore  $\llbracket M \rrbracket_{p''} = \llbracket M' \rrbracket_{p''} = \llbracket M'' \rrbracket_{p''}$ . For any other process  $p'''$  such  
 1041 that  $\llbracket M'' \rrbracket_{p'''} = \perp$ , we get  $\llbracket M \rrbracket_{p'''} = \llbracket N \rrbracket_{p'''} = \perp$  and  $\llbracket M' \rrbracket_{p'''} = \llbracket N' \rrbracket_{p'''} = \perp$ . And by induction  
 1042  $\llbracket N \rrbracket \xrightarrow{*}_{[D]} \mathcal{N}_N$  and  $N' \xrightarrow{*}_{[D]} N''$  for  $\mathcal{N}_N \equiv \llbracket N'' \rrbracket$ . For any process  $p$  we therefore get  
 1043  $\llbracket N \rrbracket_p \xrightarrow{\mu_0}_{[D]} \xrightarrow{\mu_1}_{[D]} \dots B_p$  for  $B_p \equiv \llbracket N'' \rrbracket_p$  for some sequences of transitions  $\xrightarrow{\mu_0}_{[D]} \xrightarrow{\mu_1}_{[D]}$   
 1044  $\dots$ , and from the network semantics we get

$$1045 \quad \begin{aligned} \llbracket M \rrbracket \rightarrow^* & \prod_{p \in \text{pn}(\text{type}(N)) \cup (\text{pn}(N) \cap \text{pn}(M''))} p[B_p \llbracket M'' \rrbracket_p] \mid \prod_{\llbracket N \rrbracket_{p'} = \llbracket M'' \rrbracket_{p'} = \perp} p'[\perp] \\ & \mid \prod_{\llbracket M \rrbracket_{p''} = \llbracket M'' \rrbracket_{p''}} p''[\llbracket M'' \rrbracket_{p''}] \mid \prod_{\llbracket M \rrbracket_{p'''} = \llbracket N \rrbracket_{p'''}} p'''[B_{p'''}] = \mathcal{N} \end{aligned}$$

1046 and  $M' \rightarrow^* N'' \ M$ . And since  $\llbracket N \rrbracket \xrightarrow{*}_{[D]} \mathcal{N}'$  and  $\llbracket N' \rrbracket \xrightarrow{*}_{[D]} \mathcal{N}'_N$ , we know these  
 1047 sequences of transitions can synchronise when necessary, and if  $\llbracket N \rrbracket_{p'''} \neq \llbracket N' \rrbracket_{p'''} = \perp$   
 1048 then we can do the extra application to get rid of this unit.

1049 ■ Assume  $M = V N$ ,  $M' = V N'$ , and  $N \xrightarrow{\tau, P}_D N'$ . Then for any process  $p \in \text{pn}(\text{type}(V))$ ,  
 1050  $\llbracket M \rrbracket_p = \llbracket V \rrbracket_p \llbracket N \rrbracket_p$  and  $\llbracket M' \rrbracket_p = \llbracket V \rrbracket_p \llbracket N' \rrbracket_p$ . Since  $V$  is a value, for any process  $p' \notin$   
 1051  $\text{pn}(\text{type}(V))$ , we have  $\llbracket V \rrbracket_{p'} = \perp$  and so for any process  $p'$  such that  $\llbracket V \rrbracket_{p'} = \llbracket N \rrbracket_{p'} = \perp$ ,  
 1052 by induction we get  $\llbracket N' \rrbracket_{p'} = \perp$  and therefore  $\llbracket M \rrbracket_{p'} = \llbracket M' \rrbracket_{p'} = \perp$ . For any other process  
 1053  $p''$  such that  $\llbracket V \rrbracket_{p''} = \perp$ , we have  $\llbracket M \rrbracket_{p''} = \llbracket N \rrbracket_{p''}$  and  $\llbracket M' \rrbracket_{p''} = \llbracket N' \rrbracket_{p''}$ . By induction,  
 1054  $\llbracket N \rrbracket \rightarrow^*_{[D]} \mathcal{N}_N$  and  $N' \rightarrow^*_{[D]} N''$  for  $\mathcal{N}_N \sqsubseteq \llbracket N'' \rrbracket$ . For any process  $p$  we therefore  
 1055 get  $\llbracket N \rrbracket_p \xrightarrow{\mu_0}_{[D](p)} \xrightarrow{\mu_1}_{[D](p)} \dots B_p$  for  $B_p \sqsubseteq \llbracket N'' \rrbracket_p$  for some sequences of transitions  
 1056  $\xrightarrow{\mu_0}_{[D](p)} \xrightarrow{\mu_1}_{[D](p)} \dots$  and from the network semantics we get

$$1057 \quad \llbracket M \rrbracket \rightarrow^* \prod_{p \in \text{pn}(\text{type}(N))} p[\llbracket V \rrbracket_p B_p] \mid \prod_{p' \notin \text{pn}(\text{type}(N))} p'[B_{p'}] = \mathcal{N}$$

1058 and

$$1059 \quad M' \rightarrow^* V N''$$

1060 and the result follows.

- 1061 ■ Assume  $M = M'' N$ ,  $M' = M'' N'$ ,  $N \xrightarrow{\tau, P}_D N'$ , and  $\text{pn}(M) \cap P = \emptyset$ . Then for any  $p \in P$ ,  
 1062  $\text{pn}(\llbracket M'' \rrbracket_p) \cap P = \emptyset$  and the result follows from induction and using rule NAPP3.
- 1063 ■ Assume  $M = \text{case } N \text{ of } \text{Inl } x \Rightarrow N'; \text{Inr } x' \Rightarrow N''$ ,  $M' = \text{case } M'' \text{ of } \text{Inl } x \Rightarrow$   
 1064  $N'; \text{Inr } x \Rightarrow N''$ , and  $N \xrightarrow{\tau, P}_D M''$ . Then for any process  $p$  such that  $p \in \text{pn}(\text{type}(N))$ ,  
 1065 we have projections  $\llbracket M \rrbracket_p = \text{case } \llbracket N \rrbracket_p \text{ of } \text{Inl } x \Rightarrow \llbracket N' \rrbracket_p; \text{Inr } x' \Rightarrow \llbracket N'' \rrbracket_p$  and  $\llbracket M' \rrbracket_p =$   
 1066  $\text{case } \llbracket M'' \rrbracket_p \text{ of } \text{Inl } x \Rightarrow \llbracket N' \rrbracket_p; \text{Inr } x' \Rightarrow \llbracket N'' \rrbracket_p$ . For any other process  $p'$  such that  
 1067  $\llbracket N \rrbracket_{p'} = \llbracket N' \rrbracket_{p'} = \llbracket N'' \rrbracket_{p'} = \perp$ , by induction we get  $\llbracket M'' \rrbracket_{p'} = \perp$ , and therefore  $\llbracket M \rrbracket_{p'} =$   
 1068  $\llbracket M' \rrbracket_{p'} = \perp$ . For any other process  $p''$  such that  $\llbracket N \rrbracket_{p''} = \perp$ , we get  $\llbracket M \rrbracket_{p''} = \llbracket M' \rrbracket_{p''} =$   
 1069  $\llbracket N' \rrbracket_{p''} \sqcup \llbracket N'' \rrbracket_{p''}$ . For any other processes  $p'''$  such that  $\llbracket N' \rrbracket_{p'''} = \llbracket N'' \rrbracket_{p'''} = \perp$ , we  
 1070 have  $\llbracket M \rrbracket_{p'''} = \llbracket N \rrbracket_{p'''}$  and  $\llbracket M' \rrbracket_{p'''} = \llbracket M'' \rrbracket_{p'''}$ . For any other process  $p''''$ , we have  
 1071  $\llbracket M \rrbracket_{p''''} = (\lambda x : \perp. \llbracket N' \rrbracket_{p''''} \sqcup \llbracket N'' \rrbracket_{p''''}) \llbracket N \rrbracket_{p''''}$  and  $\llbracket M' \rrbracket_{p''''} = (\lambda x. \llbracket N' \rrbracket_{p''''} \sqcup \llbracket N'' \rrbracket_{p''''}) \llbracket M'' \rrbracket_{p''''}$   
 1072 for  $x \notin \text{fv}(N') \cup \text{fv}(N'')$ . The rest follows by simple induction similar to the second case.
- 1073 ■ Assume  $M = \text{case } N \text{ of } \text{Inl } x \Rightarrow N_1; \text{Inr } x' \Rightarrow N_2$ ,  $M' = \text{case } N \text{ of } \text{Inl } x \Rightarrow N'_1; \text{Inr } x' \Rightarrow$   
 1074  $N'_2$ ,  $N_1 \xrightarrow{\tau, P}_D N'_1$ ,  $N_2 \xrightarrow{\tau, P}_D N'_2$ , and  $P \cap \text{pn}(N) = \emptyset$ . Then for any process  $p$  such  
 1075 that  $p \in \text{pn}(\text{type}(N))$ , we have  $\llbracket M \rrbracket_p = \text{case } \llbracket N \rrbracket_p \text{ of } \text{Inl } x \Rightarrow \llbracket N'_1 \rrbracket_p; \text{Inr } x' \Rightarrow \llbracket N'_2 \rrbracket_p$   
 1076 For any other process  $p'$  such that  $\llbracket N \rrbracket_{p'} = \llbracket N_1 \rrbracket_{p'} = \llbracket N_2 \rrbracket_{p'} = \perp$ , by induction we get  
 1077  $\llbracket N'_1 \rrbracket_{p'} = \llbracket N'_2 \rrbracket_{p'} = \perp$ , and therefore  $\llbracket M \rrbracket_{p'} = \llbracket M' \rrbracket_{p'} = \perp$ . For any other process  $p''$  such  
 1078 that  $\llbracket N \rrbracket_{p''} = \perp$ , we get  $\llbracket M \rrbracket_{p''} = \llbracket N_1 \rrbracket_{p''} \sqcup \llbracket N_2 \rrbracket_{p''}$ . For any other processes  $p'''$  such that  
 1079  $\llbracket N_1 \rrbracket_{p'''} = \llbracket N_2 \rrbracket_{p'''} = \perp$ , we have  $\llbracket M \rrbracket_{p'''} = \llbracket N \rrbracket_{p'''}$ . For any other process  $p''''$ , we have  
 1080  $\llbracket M \rrbracket_{p''''} = (\lambda x : \perp. \llbracket N_1 \rrbracket_{p''''} \sqcup \llbracket N_2 \rrbracket_{p''''}) \llbracket N \rrbracket_{p''''}$ . If  $\llbracket N'_1 \rrbracket_p \sqcup \llbracket N'_2 \rrbracket_p$  is defined for all  $p$  then  
 1081 the result follows from induction. Otherwise we have  $M_1$  and  $M_2$  such that  $N'_1 \xrightarrow{\tau, P}_D M_1$   
 1082 and  $N'_2 \rightarrow \tau, P_D M_2$  and  $\llbracket M_1 \rrbracket_p \sqcup \llbracket M_2 \rrbracket_p$  for all  $p$ , and the result follows from induction  
 1083 on these transitions.
- 1084 ■ Assume  $M = \text{case } \text{Inl } V \text{ of } \text{Inl } x \Rightarrow N; \text{Inr } x' \Rightarrow N'$  and  $M' = N[x := V]$ . Then for any  
 1085 process  $p \in \text{pn}(\text{type}(\text{Inl } V))$ , we have  $\llbracket M \rrbracket_p = \text{case } \text{Inl } \llbracket V \rrbracket_p \text{ of } \text{Inl } x \Rightarrow \llbracket N \rrbracket_p; \text{Inr } x' \Rightarrow$   
 1086  $\llbracket N' \rrbracket_p$  and  $\llbracket M' \rrbracket_p = \llbracket N[x := \llbracket V \rrbracket_p] \rrbracket_p$ . By Lemma 38,  $\llbracket N[x := \llbracket V \rrbracket_p] \rrbracket_p = \llbracket N \rrbracket_p[x := \llbracket V \rrbracket_p]$ .  
 1087 For any other process  $p' \notin \text{pn}(\text{type}(\text{Inl } V))$ ,  $\llbracket \text{Inl } V \rrbracket_{p'} = \perp$ , and therefore  $\llbracket M \rrbracket_{p'} =$   
 1088  $\llbracket N \rrbracket_{p'} \sqcup \llbracket N' \rrbracket_{p'} \sqsupset \llbracket N \rrbracket_{p'} = \llbracket M' \rrbracket_{p'}$ . The result follows.
- 1089 ■ Assume  $M = \text{case } \text{Inr } V \text{ of } \text{Inl } x \Rightarrow N; \text{Inr } x' \Rightarrow N'$  and  $M' = N'[x' := V]$ . This case is  
 1090 similar to the previous.
- 1091 ■ Assume  $M = \text{case } N \text{ of } \text{Inl } x \Rightarrow N_1; \text{Inr } x' \Rightarrow N_2$ ,  $M' = \text{case } N \text{ of } \text{Inl } x \Rightarrow N'_1; \text{Inr } x' \Rightarrow$   
 1092  $N'_2$ ,  $N_1 \xrightarrow{P}_D N'_1$ ,  $N_2 \xrightarrow{P}_D N'_2$ , and  $P \cap \text{pn}(N) = \emptyset$ . This case is similar to case four.

- 1093 ■ Assume  $M = \mathbf{com}_{q,p} V$  and  $M' = V[q := p]$  and  $\text{fv}(V) = \emptyset$ . Then if  $q \neq p$ ,  
1094  $\llbracket M \rrbracket_p = \mathbf{recv}_q \perp$ ,  $\llbracket M' \rrbracket_p = \llbracket V[q := p] \rrbracket_p = \llbracket V \rrbracket_p[q := p]$  since  $\text{pn}(\text{type}(V)) = q$ ,  
1095  $\llbracket M \rrbracket_q = \mathbf{send}_p \llbracket V \rrbracket_q$ ,  $\llbracket M' \rrbracket_q = \perp$ , and for any  $p' \notin \{q, p\}$ ,  $\llbracket M \rrbracket_{p'} = \llbracket M' \rrbracket_{p'} = \perp$ . We there-  
1096 fore get  $\llbracket M \rrbracket_p \xrightarrow{\mathbf{recv}_q \llbracket V \rrbracket_q[q := p]} \llbracket D \rrbracket \llbracket M' \rrbracket_p$ ,  $\llbracket M \rrbracket_q \xrightarrow{\mathbf{send}_p \llbracket V \rrbracket_q} \llbracket D \rrbracket \llbracket M' \rrbracket_q$ , and  $\llbracket M \rrbracket_{p'} = \llbracket M' \rrbracket_{p'}$ .  
1097 We define  $\mathcal{N} = \mathcal{N}' = \llbracket M' \rrbracket$  and the result follows. If  $q = p$ , then  $\llbracket M \rrbracket_p = (\lambda x.x) \llbracket V \rrbracket_p$   
1098 and  $\llbracket M' \rrbracket_p = \llbracket V \rrbracket_p$  and  $\mathcal{N} = \mathcal{N}' = \llbracket M' \rrbracket$  and the result follows.
- 1099 ■ Assume  $M = \mathbf{select}_{q,p} l M'$ . Then  $\llbracket M \rrbracket_q = \oplus_p l \llbracket M' \rrbracket_q$ ,  $\llbracket M \rrbracket_p = \&\{l : \llbracket M' \rrbracket_p\}$ , and for  
1100 any  $p' \notin \{q, p\}$ ,  $\llbracket M \rrbracket_{p'} = \llbracket M' \rrbracket_{p'}$ . We therefore get  $\llbracket M \rrbracket \xrightarrow{\tau_{p,q}} \llbracket D \rrbracket \llbracket M \rrbracket \setminus \{p, q\} \mid p[\llbracket M' \rrbracket_p] \mid$   
1101  $q[\llbracket M' \rrbracket_q]$  and the result follows.
- 1102 ■ Assume  $M = \mathbf{select}_{q,p} l N$ ,  $M' = \mathbf{select}_{q,p} l N'$ ,  $N \xrightarrow{\tau, P} N'$ , and  $P \cap \{q, p\} = \emptyset$ . Then  
1103  $\llbracket M \rrbracket_q = \oplus_p l \llbracket N \rrbracket_q$ ,  $\llbracket M' \rrbracket_q = \oplus_p l \llbracket N' \rrbracket_q$ ,  $\llbracket M \rrbracket_p = \&\{l : \llbracket N \rrbracket_p\}$ ,  $\llbracket M' \rrbracket_p = \&\{l : \llbracket N' \rrbracket_p\}$ , and  
1104 for any  $p' \notin \{q, p\}$ ,  $\llbracket M \rrbracket_{p'} = \llbracket N \rrbracket_{p'}$  and  $\llbracket M' \rrbracket_{p'} = \llbracket N' \rrbracket_{p'}$ . The result follows from induction  
1105 and using rules **NOFF2** and **NCHO2**.
- 1106 ■ Assume  $M = \mathbf{fst Pair} V V'$  and  $M' = V$ . Then for any process  $p \in \text{pn}(\text{type}(\mathbf{Pair} M' V'))$ ,  
1107  $\llbracket M \rrbracket_p = \mathbf{fst Pair} \llbracket M' \rrbracket_p \llbracket V' \rrbracket_p$  and for any other process  $p' \notin \text{pn}(\text{type}(\mathbf{Pair} M' V'))$ , we  
1108 have  $\llbracket M \rrbracket_{p'} = \perp$  and  $\llbracket M' \rrbracket_{p'} = \perp$ . We define  $\mathcal{N} = \mathcal{N}' = \llbracket M' \rrbracket$  and the result follows.
- 1109 ■ Assume  $M = \mathbf{snd Pair} V V'$  and  $M' = V'$ . Then the case is similar to the previous.
- 1110 ■ Assume  $M = f(\vec{p})$  and  $M' = D(f(\vec{p}))[\vec{p} := \vec{p}]$ . Then the result follows from the  
1111 definition of  $\llbracket D \rrbracket$ .
- 1112 ■ Assume there exists  $N$  such that  $M \rightsquigarrow N$  and  $N \xrightarrow{\tau, P} M'$ . Then the result follows  
1113 from induction and Lemma 40.

1114

## 1115 A.2 Proof of Theorem 26

1116 ▶ **Definition 41.** Given a network  $\mathcal{N} = \prod_{p \in \rho} p[B_p]$ , we have  $\mathcal{N}_{\setminus \rho'} = \prod_{p \in (\rho \setminus \rho')} p[B_p]$

1117 ▶ **Lemma 42.** For any process  $p$  and network  $\mathcal{N}$ , if  $\mathcal{N} \xrightarrow{\tau, P} \mathcal{N}'$  and  $p \notin P$  then  $\mathcal{N}(p) = \mathcal{N}'(p)$ .

1118 **Proof.** Straightforward from the network semantics. ◀

1119 ▶ **Lemma 43.** For any set of processes  $P$  and network  $\mathcal{N}$ , if  $\mathcal{N} \xrightarrow{\tau, P'} \mathcal{N}'$  and  $P \cap P' = \emptyset$  then  
1120  $\mathcal{N} \setminus P \xrightarrow{\tau, P'} \mathcal{N}' \setminus P$ .

1121 **Proof.** Straightforward from the network semantics. ◀

1122 ▶ **Lemma 44.** If  $\llbracket M \rrbracket_p \rightsquigarrow B$  then there exists  $M'$  such that  $M \rightsquigarrow M'$  and  $B \equiv \llbracket M' \rrbracket_p$

1123 **Proof.** Follows from case analysis on  $\llbracket M \rrbracket_p \rightsquigarrow B$  keeping in mind that  $\llbracket M \rrbracket_p$  cannot be  
1124  $\perp \perp$ . ◀

1125 **Proof of Theorem 26.** If  $\llbracket M \rrbracket \rightarrow_{\llbracket D \rrbracket}^* \mathcal{N}$  uses rule **NSTR** then this follows from Lemma 44.  
1126 Otherwise we prove this by structural induction on  $M$ .

- 1127 ■ Assume  $M = N_1 N_2$ . Then for any process  $p \in \text{pn}(\text{type}(N_1)) \cup (\text{pn}(N_1) \cap \text{pn}(N_2))$ ,  
1128  $\llbracket M \rrbracket_p = \llbracket N_1 \rrbracket_p \llbracket N_2 \rrbracket_p$ , for any process  $p'$  such that  $\llbracket N_1 \rrbracket_{p'} = \llbracket N_2 \rrbracket_{p'} = \perp$ , we have  
1129  $\llbracket M \rrbracket_{p'} = \perp$ . For any other process  $p''$  such that  $\llbracket N_1 \rrbracket_{p''} = \perp$ ,  $\llbracket M \rrbracket_{p''} = \llbracket N_2 \rrbracket_{p''}$ . For any  
1130 other process  $p'''$  such that  $\llbracket N_2 \rrbracket_{p'''} = \perp$ , we get  $\llbracket M \rrbracket_{p'''} = \llbracket N_1 \rrbracket_{p'''}$ . We then have 2 cases.

- 1131 = Assume  $N_2 = V$ . Then  $\llbracket N_2 \rrbracket_p = L$  by Lemma 36, and for any  $p'$  such that  $p' \notin$   
 1132  $\text{pn}(\text{type}(N_2)) \subseteq \text{pn}(\text{type}(N_1))$ , by Lemma 38,  $\llbracket N_2 \rrbracket_{p'} = \perp$  and therefore  $\llbracket M \rrbracket_{p'} =$   
 1133  $\llbracket N_1 \rrbracket_{p'}$ , and we have 5 cases.
- 1134 \* Assume  $N_1 = \lambda x : T.N_3$ . Then for any process  $p \in \text{pn}(\text{type}(N_1))$ ,  $\llbracket N_1 \rrbracket_p = \lambda x :$   
 1135  $\llbracket T \rrbracket_p . \llbracket N_3 \rrbracket_p$ . And for any process  $p' \notin \text{pn}(\text{type}(N_1))$ ,  $\llbracket N_1 \rrbracket_{p'} = \perp$ . We have two cases,  
 1136 using either rule NABSAPP or rules NINABS and NAPP1.  
 1137 If we use rule NABSAPP, then there exists  $p''$  such that  $P = p''$  and  $p'' \in$   
 1138  $\text{pn}(\text{type}(N_1))$ . We then get  $\llbracket M \rrbracket \xrightarrow{\tau, P}_{[D]} \mathcal{M} = \llbracket M \rrbracket \setminus \{p''\} \mid p''[\llbracket N_3 \rrbracket_{p''}[x := \llbracket N_2 \rrbracket_{p''}]]$ .  
 1139 Since  $\mathcal{M} \rightarrow^* \llbracket N_3[x := N_2] \rrbracket$  and the remaining transitions in  $\llbracket M \rrbracket \rightarrow^*_{[D]} \mathcal{N}$  take  
 1140 place in  $N_3$ , the result follows from using rule NABSAPP in every process in  
 1141  $\text{pn}(\text{type}(N_1))$  and induction.  
 If we use rules NINABS and NAPP1 then there exists  $p''$  such that  $P = p''$  and  
 $\llbracket N_3 \rrbracket_{p''} \xrightarrow{\mu} B$  and

$$\llbracket M \rrbracket \xrightarrow{\mu}_{[D]} \llbracket M \rrbracket \setminus \{p''\} \mid p''[\lambda x.B \llbracket N_2 \rrbracket_{p''}] \rightarrow^*_{[D]} \mathcal{N}$$

By induction,  $N_3 \rightarrow_D^* N'_3$  and  $(\llbracket N_3 \rrbracket \setminus \{p''\} \mid p''[B] \rightarrow_D \mathcal{N}'')$  such that  $\llbracket N_3 \rrbracket \supseteq \mathcal{N}''$ ,  
 and we define  $M' = \lambda x : T.N'_3 N_2$  and

$$\mathcal{N}' = (\mathcal{N} \setminus \text{pn}(\text{type}(N_1))) \mid \prod_{p \in \text{pn}(\text{type}(N_3))} p[(\lambda x.\mathcal{N}''(p)) \llbracket N_2 \rrbracket_{p'}]$$

1142 and the result follows by using rules INABS, APP1, NINABS, and NAPP1 and  
 1143 induction.

- 1144 \* Assume  $N_1 = \mathbf{com}_{q,p}$ . Then if  $q \neq p$ ,  $\llbracket M \rrbracket_q = \mathbf{send}_p \llbracket N_2 \rrbracket_q$ ,  $\llbracket M \rrbracket_p = \mathbf{recv}_p \perp$ , and  
 1145 for  $p' \notin \{q, p\}$ ,  $\llbracket N_1 \rrbracket_{p'} = \perp = \llbracket M \rrbracket_{p'}$ , and therefore  $P = q, p$ , and if  $q = p$  then  
 1146  $\llbracket N_1 \rrbracket_p = \lambda x.x$ .  
 1147 If  $P = q, p$  then  $\mathcal{N} = \llbracket M \rrbracket \setminus \{q, p\} \mid q[\perp] \mid p[\llbracket N_2 \rrbracket_q[q := p]]$ . Because  $\llbracket N_2 \rrbracket_p = \perp$  and  
 1148  $\llbracket N_2 \rrbracket_q = V$ ,  $N_2 = V$ . Therefore  $M \xrightarrow{P}_D V[q := p]$  and the result follows.  
 1149 If  $P = p$  then  $q = p$ ,  $\mathcal{N} = \llbracket M \rrbracket \setminus \{p\} \mid p[\llbracket N_2 \rrbracket_p]$  and the rest is similar to above.
- 1150 \* Assume  $N_1 = \mathbf{fst}$ . Then  $N_2 = \mathbf{Pair} V V'$  and for any process  $p \in \text{pn}(\text{type}(\mathbf{Pair} V V'))$ ,  
 1151  $\llbracket M \rrbracket_p = \mathbf{fst} \mathbf{Pair} \llbracket V \rrbracket_p \llbracket V' \rrbracket_p$  and for any other process  $p' \notin \text{pn}(\text{type}(\mathbf{Pair} V V'))$ , by  
 1152 Lemma 38 we have  $\llbracket M \rrbracket_{p'} = \llbracket N_1 \rrbracket_{p'} = \perp$ , and therefore  $\llbracket M \rrbracket_{p'} \rightarrow$ .

1153 If  $P = p \in \text{pn}(\text{type}(\mathbf{Pair} V V'))$  then  $\mathcal{N} = \llbracket M \rrbracket \setminus \{p\} \mid p[\llbracket V \rrbracket_p]$  and  $M \xrightarrow{P}_D V$ . The  
 1154 result follows by use of rule NPROJ1 and Lemma 38.

- 1155 \* Assume  $N_1 = \mathbf{snd}$ . This case is similar to the previous.
- 1156 \* Otherwise,  $N_1 \neq V$  and either  $\llbracket M \rrbracket \xrightarrow{\tau_p}_{[D]} \mathcal{M} \rightarrow^*_{[D]} \mathcal{N}$  or  $\llbracket M \rrbracket \xrightarrow{\tau_{p,q}}_{[D]} \mathcal{M} \rightarrow^*_{[D]} \mathcal{N}$ .  
 1157 If  $\llbracket M \rrbracket \xrightarrow{\tau_p}_{[D]} \mathcal{M} \rightarrow^*_{[D]} \mathcal{N}$  then either  $\llbracket N_1 \rrbracket_p \xrightarrow{\tau} B$  and  $p \in \text{pn}(\text{type}(N_1))$ ,  $\mathcal{M} =$   
 1158  $\llbracket M \rrbracket \setminus \{p\} \mid p[B \llbracket N_2 \rrbracket_p]$ . We therefore have  $\llbracket N_1 \rrbracket \xrightarrow{\tau_p} \llbracket N_1 \rrbracket \setminus \{p\} \mid p[B]$ , and by  
 1159 induction,  $N_1 \rightarrow_D^* N'_1$  such that  $\llbracket N_1 \rrbracket \setminus \{p\} \mid p[B] \rightarrow^* \mathcal{N}_1 \supseteq \llbracket N'_1 \rrbracket$ . Since all these  
 1160 transitions can be propagated past  $N_2$  in the network and  $\llbracket N_2 \rrbracket_{p'}$  in any process  $p'$   
 1161 involved, we get the result for  $M' = N'_1 N_2$ .

1162 If  $\llbracket M \rrbracket \xrightarrow{\tau_{p,q}}_{[D]} \mathcal{M} \rightarrow^*_{[D]} \mathcal{N}$  then the case is similar.

- 1163 = If  $N_2 \neq V$  then we have 2 cases.

- 1164 \* If  $\llbracket M \rrbracket \xrightarrow{\tau_p}_{[D]} \mathcal{M} \rightarrow^*_{[D]} \mathcal{N}$  then either  $\llbracket N_1 \rrbracket_p \xrightarrow{\tau} B$  or  $\llbracket N_2 \rrbracket_p \xrightarrow{\tau} B$  and the case is  
 1165 similar to the previous.
- 1166 \* If  $\llbracket M \rrbracket \xrightarrow{\tau_{p,q}}_{[D]} \mathcal{M} \rightarrow^*_{[D]} \mathcal{N}$  then there exists  $L$  such that either  $\llbracket N_1 \rrbracket_q \xrightarrow{\mathbf{send}_p L} B_q$   
 1167 or  $\llbracket N_2 \rrbracket_q \xrightarrow{\mathbf{send}_p L} B_q$  and  $\llbracket N_1 \rrbracket_p \xrightarrow{\mathbf{recv}_q L[q:=p]} B_p$  or  $\llbracket N_2 \rrbracket_p \xrightarrow{\mathbf{recv}_q L[q:=p]} B_p$ .

1168 If  $\llbracket N_1 \rrbracket_q \xrightarrow{\text{send}_p L} B_q$  then  $\llbracket N_1 \rrbracket_q \neq L'$  and therefore  $\llbracket N_1 \rrbracket_p \xrightarrow{\text{recv}_q L[q:=p]} B_p$  and the  
 1169 case is similar to the previous. If  $\llbracket N_2 \rrbracket_q \xrightarrow{\text{send}_p L} B_q$  then  $\llbracket N_1 \rrbracket_q = L'$ , and therefore  
 1170  $\llbracket N_2 \rrbracket_p \xrightarrow{\text{recv}_q L[q:=p]} B_p$  and the case is similar to the previous.

1171 ■ Assume  $M = \text{case } N \text{ of } \text{Inl } x \Rightarrow N'; \text{Inr } x' \Rightarrow N''$ . Then for any process  $p \in \text{pn}(\text{type}(N))$ ,  
 1172  $\llbracket M \rrbracket_p = \text{case } \llbracket N \rrbracket_p \text{ of } \text{Inl } x \Rightarrow \llbracket N' \rrbracket_p; \text{Inr } x' \Rightarrow \llbracket N'' \rrbracket_p$ . And for any other process  
 1173  $p' \notin \text{pn}(\text{type}(N))$ ,  $\llbracket M \rrbracket_{p'} = (\lambda x. \llbracket N' \rrbracket_{p'} \sqcup \llbracket N'' \rrbracket_{p'}) \llbracket N \rrbracket_{p'}$ . We know that  $\llbracket M \rrbracket \xrightarrow{\tau_p}_{[D]} \mathcal{M}$   
 1174  $\mathcal{M} \rightarrow_{[D]}^* \mathcal{N}$  and we have three cases.

1175 ■ Assume  $P = p \in \text{pn}(\text{type}(N))$ . Then we have three cases.  
 1176 \* Assume  $N = \text{Inl } V$ . Then  $\llbracket N \rrbracket_p = \text{Inl } \llbracket V \rrbracket_p$  and  $\mathcal{M} = \llbracket M \rrbracket \setminus \{p\} \mid p[\llbracket N' \rrbracket[x := \llbracket V \rrbracket_p]]$ .  
 1177 We define  $M'' = N'$  and the transitions used in  $\mathcal{M} \rightarrow_{[D]}^* \mathcal{N}$  can be used on  
 1178  $M''$ . By induction, since  $\llbracket N' \rrbracket_{p'} \sqsubseteq \llbracket N' \rrbracket_{p'} \sqcup \llbracket N'' \rrbracket_{p'}$  the result follows from using  
 1179 rules NABSAPP and NCASEL.  
 1180 \* Assume  $N = \text{Inr } V$ . Then the case is similar to the previous.  
 1181 \* Otherwise, we use either rule NCASE or rule NCASE2. If we use rule NCASE, we  
 1182 have a transition  $\llbracket N \rrbracket_p \xrightarrow{\tau} B$  such that

$$1183 \quad \mathcal{M} = \llbracket M \rrbracket \setminus \{p\} \mid p[\text{case } B \text{ of } \text{Inl } x \Rightarrow \llbracket N' \rrbracket_p; \text{Inr } x' \Rightarrow \llbracket N'' \rrbracket_p]$$

1184 and the result follows from induction similar to the last application case.  
 1185 If we use rule NCASE2 then  $\llbracket N' \rrbracket_p \xrightarrow{\tau_D} B$  and  $\llbracket N'' \rrbracket_p \xrightarrow{\tau_D} B$ . If  $\llbracket N' \rrbracket_p \xrightarrow{\tau_D} B$  then  
 1186 by induction,  $N' \rightarrow_D^* N'''$  and  $\llbracket N' \rrbracket \setminus \{p\} \mid p[B] \rightarrow_D^* \mathcal{N}''$  such that  $\mathcal{N}'' \sqsupseteq \llbracket N''' \rrbracket$  and  
 1187  $N'' \rightarrow_D^* N'''$  and  $\llbracket N'' \rrbracket \setminus \{p\} \mid p[B] \rightarrow_D^* \mathcal{N}'''$  such that  $\mathcal{N}''' \sqsupseteq \llbracket N''' \rrbracket$ . Since  $N'$  and  
 1188  $N''$  are mergeable on other processes, the result follows from using rule INCASE.

1189 ■ Assume  $P = p \notin \text{pn}(\text{type}(N))$ . Then we have three cases.  
 1190 \* Assume  $N = \text{Inl } V$ . Then  $\llbracket N \rrbracket_p = \perp$  and  $\mathcal{M} = \llbracket M \rrbracket \setminus \{p\} \mid p[\llbracket N' \rrbracket_p \sqcup \llbracket N'' \rrbracket_p]$ . We  
 1191 define  $M' = N'$  and the result follows.  
 1192 \* Assume  $N = \text{Inr } V$ . Then the case is similar to the previous.  
 1193 \* Otherwise,  $\llbracket N \rrbracket_p \neq L$  and we therefore have  $\llbracket N \rrbracket_p \xrightarrow{\tau} B$  and  $\mathcal{M} = \llbracket M \rrbracket \setminus \{p\} \mid$   
 1194  $p[(\lambda x. \llbracket N' \rrbracket_p \sqcup \llbracket N'' \rrbracket_p) B]$ . We therefore have  $\llbracket N \rrbracket \xrightarrow{\tau_p}_{[D]} \llbracket N \rrbracket \setminus \{p\} \mid p[B]$ , and by  
 1195 induction,  $N \rightarrow_D^* N'''$  such that  $\llbracket N \rrbracket \setminus \{p\} \mid p[B] \rightarrow_D^* \mathcal{N}'''$  for  $\mathcal{N}''' \sqsupseteq \llbracket N''' \rrbracket$ . Since all  
 1196 these transitions can be propagated past  $N_2$  in the network and the conditional  
 1197 or  $(\lambda x. \llbracket N' \rrbracket_{p'} \sqcup \llbracket N'' \rrbracket_{p'})$  in any other process  $p'$  involved, we get the result for  
 1198  $M' = \text{case } N''' \text{ of } \text{Inl } x \Rightarrow N'; \text{Inr } x' \Rightarrow N''$ .

1199 ■ Assume  $P = q, p$ . Then the logic is similar to the third subcases of the previous two  
 1200 cases.

1201 ■ Assume  $M = \text{select}_{q,p} \ell N$ . This is similar to the  $N_1 = \text{com}_{q,p}$  case above.

1202 ■ Assume  $M = f(p_1, \dots, p_n)$ . Then

$$1203 \quad \llbracket M \rrbracket = \prod_{i=1}^n p_i[f_i(p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n)] \mid \prod_{p \notin \{p_1, \dots, p_n\}} p[\perp]$$

1202 We therefore have some process  $p$  such that  $P = p$  and  $(\llbracket M \rrbracket \setminus p_i) \mid p_i[\llbracket D \rrbracket(f_i(\vec{p}'))][\vec{p}' :=$   
 1203  $p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n]] \rightarrow^* \mathcal{N}$ . We then define the required choreography  $M'' =$   
 1204  $D(f(p'_1, \dots, p'_n))[\vec{p}'_1, \dots, \vec{p}'_n := p_1, \dots, p_n]$  and network

$$1205 \quad \mathcal{N} = \llbracket M'' \rrbracket = \prod_{i=1}^n p_i[\llbracket D \rrbracket(f_i(\vec{p}'))][\vec{p}' := p_1, \dots, p_{i-1}, p_{i+1}, \dots, p_n]] \mid \prod_{p \notin p_1, \dots, p_n} p[\perp]$$

1206 and the result follows from induction.

