Choreographies as Functions

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

We propose a new interpretation of choreographies as functions, whereby coordination protocols for concurrent and distributed systems are expressed in terms of a λ-calculus. Our language is expressive enough to enable, for the first time, the writing of higher-order protocols that do not require central control. Nevertheless, it retains the simplicity and elegance of the λ-calculus, and it is possible to translate choreographies into endpoint implementations.

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

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

Digital Object Identifier 10.4230/LIPIcs.CONCUR.2021.XX

1 Introduction

Choreographic Languages and Endpoint Projection Choreographies are coordination plans for concurrent and distributed systems, which define the communications that should be enacted by a system of processes [24, 33, 37]. Implementing choreographies is notoriously hard, because of the usual issues of concurrent programming. Notably, it requires predicting how processes will interact at runtime, for which programmers do not receive adequate help from mainstream programming technology [25, 29, 34]. This challenge has spawned a prolific area of research within the communities of concurrency theory and programming languages, which focuses on the definition of choreographic languages (languages for expressing choreographies) and how terms in such languages can be correctly translated into abstract models of implementations [3, 23].

Current formulations of choreographic languages are based on the theories of communicating automata [4, 16] and process calculi [36, 5, 22], inspired by earlier studies on message sequence charts [2, 24]. The starting point for these works was to use these well-known theories to formulate the communication primitive of choreographic languages. This key primitive, which comes straight from the "Alice and Bob" notation of security protocols [32], allows for moving data from one process to another. In this context, processes are usually called *roles*. Many choreographic languages come with a translation of choreographies into abstractions of implementations, typically called Endpoint Projection (EPP), and a proof of its correctness, typically defined as an operational correspondence result [6].

The Issue of Compositionality In practice, choreographies are large—some even over a hundred pages of text [35]. Thus, it is important to understand the principles of how choreographies can be made modular, enabling the writing (preferably disciplined by types) of large choreographies as compositions of smaller, reusable ones.

Previous work investigated of how choreographic languages can be extended with parametric procedures, by introducing ad-hoc extensions informally inspired by the λ-calculus and its types [7, 12, 15]. However, none of these choreographic languages were designed with the λ-calculus as basis. This led to visible fragmentation due to differences in how procedures are formulated (the proverbial wheel has been reinvented many times), and also to a series of technical shortcomings, for example: partial application is not supported [19]; most works do not support higher-order composition of choreographies, and those that do require a central coordinator when entering a procedure (going against distribution, which is instead assumed in all other syntactic constructs) [12, 15]; and abstractions of parameters of different types are distinguished syntactically [7].

To date, whether the elegance, power, and canonicity of the λ-calculus can be adopted for the composition of choreographies remains unclear.

This Article We present the Choreographic λ-calculus, Chorλ for short, the first λ-calculus that supports the writing of choreographies.

In Chorλ, the communication primitive of choreographies is formulated as a function: $\mathbf{com}_{S,R}$, which is a λ-expression that takes a value at a *role* S and returns the same value at another role R. In general, for the first time, *all choreographies are λ-terms* that can be composed following the functional programming style.

Terms in Chorλ are located at roles, to reflect distribution. For example, the value 5@Alice reads "the integer 5 at Alice". Terms are typed with novel data types that are annotated with roles. In this case, 5@Alice has the type Int@Alice, read "an integer at Alice". We write type assignments in the usual way: 5@Alice : Int@Alice.

We use types to define a typing discipline for Chorλ that checks that choreographies make sense. Consider the function f defined as $\lambda x : \mathsf{Int@Alice}.\mathbf{com}_{\mathsf{Proxy},\mathsf{Bob}}\ (\mathbf{com}_{\mathsf{Alice},\mathsf{Proxy}}\ x)$, which communicates an integer from Alice to Bob by passing through an intermediary Proxy. For any term M, the composition $f\ M$ makes sense if the evaluation of M returns something of the type expected by f, that is Int@Alice. The composition f 5@Alice makes sense, but f 5@Bob does not, because the argument is not at the role expected by f. We define an operational semantics for Chorλ and prove that our type system supports type preservation and progress.

After the presentation of Chorλ, we define Endpoint Projection (EPP): a translation from terms in Chorλ to implementations in a concurrent λ-calculus (borrowing techniques from process calculi), where roles are enacted by processes that can communicate by the usual send and receive primitives. Our main result is that EPP is sound and complete, in terms of an operational correspondence: the implementation of a choreography enacts only and all the communications defined in the originating choreography. As a corollary of progress for well-typed choreographies and the correctness of EPP, we obtain that implementations of choreographies generated by our EPP always progress.

The expressivity of Chorλ is illustrated with a series of representative examples: remote procedure calls, remote transformations of lists, the Diffie-Hellman protocol for secure key exchange, and a distributed authentication protocol. Notably, we leverage compositionality to show how choreographies can be parameterised over different communication semantics, enabling protocol layering. In particular, we combine distributed authentication with a choreography for encryption to secure communications.

We believe that our results are promising not only for the future development of more expressive choreographic languages in practice, but also for bridging the community of functional programming to that of choreographic languages: this is the first time that choreographies are explained in terms of the solid foundations of λ-calculus.

2 Related Work

Choreographic languages and EPP have been successfully employed in the verification, monitoring, and synthesis of concurrent and distributed programs [3, 23]. For example, in multiparty session types, choreographies are translated to types that used to check that processes written in (variations of) the π-calculus communicate as expected [22].

In some settings, choreographies need to define computation at roles. For instance, many security protocols define how data should be encrypted and/or anonymised, and parallel algorithms define how each process implements its part of a computation. Choreographies that include computation can be defined in *choreographic programming*, which elevates choreographic languages to full-fledged programming languages [30]. Choreographic programming languages showed promise in a number of contexts, including parallel algorithms [11], cyber-physical systems [28, 27, 19], self-adaptive systems [14], system integration [18], information flow [26], and the implementation of security protocols [19].

Technically, Chorλ is a member of choreographic programming, but we believe that our principles could be applied also to other kinds of choreographic languages (e.g., by abstracting from the concrete values that are transmitted). In particular, there are several implementations of choreographic languages that are equipped with ad-hoc, limited variations of choreographic procedures (functions in Chorλ) that could benefit from our results [7, 21, 14, 19].

Our data types are inspired by the Choral programming language, an object-oriented language where object types are annotated with roles to capture choreographies [19]. Choral does not come with a formal model: its semantics and typing are only informally described. In a sense, Chorλ can be seen as the first formal investigation of the principles that underpin Choral. Compared to Choral our formalisation supports partial application (thus, e.g., configurations parameters of choreographies can be set at different stages), the term and type languages are much simpler, and we provide a provably-correct translation of choreographies to concurrent implementations.

Another related line of work is that on multitier programming and its progenitor calculus, Lambda 5 [31]. Similarly to Chorλ, Lambda 5 and multitier languages have data types with locations [38]. However, they are used very differently. In choreographic languages (thus Chorλ), programs have a "global" point of view and express how multiple roles interact with each other. By constrast, in multitier programming programs have the usual "local" point of view of a single role but they can nest (local) code that is supposed to be executed remotely. The reader interested in a detailed comparison of choreographic and multitier programming can consult [20], which presents algorithms for translating choreographies to multitier programs and vice versa. The correctness of these algorithms has never been proven, because they use the informally-specified Choral language as a representative choreographic language. We conjecture that the introduction of Chorλ could be the basis for a future investigation of formal translations between choreographic programs (in terms of Chorλ) and multitier programs (in terms of Lambda 5). In a similar direction, [9] presented a simple first-order multitier language from which it is possible to infer abstract choreographies (computation is not included) that describe the communication flows that multitier programs enact. This language, like all existing multitier languages, does not support higher-order composition of multitier programs. Establishing translations between Chorλ and multitier languages might provide insight on how multitier languages can support higher-order composition (as in our approach).

3 The Choreographic λ-calculus

In this section we introduce the Choreographic λ-calculus, Chorλ, which extends the simply typed λ-calculus [10] with roles and communication.

Syntax

▶ **Definition 1.** *The syntax of* Chorλ *is given by the following grammar*

$$M ::= V \mid f \mid M\ M \mid \textsf{\textbf{case}}\ M\ \textsf{\textbf{of Inl}}\ x \Rightarrow M\textsf{\textbf{;}}\ \textsf{\textbf{Inr}}\ x \Rightarrow M \mid \textsf{\textbf{select}}_{S,R}\ \ell\ M$$
$$V ::= x \mid \lambda x : T.M \mid \textsf{\textbf{Inl}}\ V \mid \textsf{\textbf{Inr}}\ V \mid \textsf{\textbf{fst}} \mid \textsf{\textbf{snd}} \mid \textsf{\textbf{Pair}}\ V\ V \mid ()@R \mid \textsf{\textbf{com}}_{S,R}$$
$$T ::= T \rightarrow_\rho T \mid T + T \mid T \times T \mid ()@R \mid t_\rho$$

where M is a choreography, V is a value, T is a type, x is a variable, ℓ is a label, f is a choreography name, R and S are roles, ρ is a set of roles, and t is a type name.

Abstraction $\lambda x : T.M$, variable x and application MM are as in the standard (typed) λ-calculus. Likewise for pairs and sums. For simplicity, constructors for sums (**Inl** and **Inr**) and products (**Pair**) are only allowed to take values as inputs, but this is only an apparent restriction: we can define, e.g., a function **inl** as $\lambda x : T.\textsf{\textbf{Inl}}\ x$ and then apply it to any choreography. Similarly, we define the functions **inr** and **pair** (the latter is for constructing pairs). We use these utility functions in our examples. Sums and products are deconstructed in the usual way, respectively by the **case** construct and by the **fst** and **snd** primitives.

The primitives $\textsf{\textbf{select}}_{S,R}\ \ell\ M$ and $\textsf{\textbf{com}}_{S,R}$ come from choreographies and are the only primitives of Chorλ that introduce interaction between different roles. The term $\textsf{\textbf{select}}_{S,R}\ \ell\ M$ is a *selection*, where S informs R that it has selected the label ℓ. Selections choreographically represent the communication of an internal choice made by S to R. (In the implementation of choreographies, a selection corresponds to an internal choice at the sender and an external choice at the receiver.) The term $\textsf{\textbf{com}}_{S,R}$, instead, is a *communication*; a communication is in effect a λ-expression that takes a value at role S and returns the same value at role R.

Finally, f is a name for a named choreography, which evaluates to a corresponding choreography as defined by an environment of definitions. Choreography names are used to model recursion. In the typing and semantics of Chorλ, we will use D to range over mappings of choreography names to choreographies.

In Chorλ types record the distribution of values across roles: if role R occurs in the type given to V then part of V will be located at R. Because function may involve more roles besides those listed in the types of their input and output, the type of abstractions $T \rightarrow_\rho T'$ is annotated with a set of roles ρ denoting the roles that may participate in the computation of a function with that type besides those occurring in the input T or the output T'—in the sequel we will often omit this annotation if the set of additional roles is empty thus writing $T \rightarrow T'$ instead of $T \rightarrow_\emptyset T'$. The types for sums and products are the usual ones (forming a sum or product of T and T' does not introduce new roles besides those already listed in T and T'). The type of units is annotated with the role where each unit is located; $()@R$ is the type of the unit value available (only) at role R. Named types t are annotated with the set of roles ρ occurring in their definition (we will discuss type definitions later in this section). The set of roles in a type is formally defined as follows.

▶ **Definition 2** (Roles of a type)**.** *The roles of a type T,* $\mathrm{roles}(T)$*, are defined as follows.*

$$\mathrm{roles}(t_\rho) = \rho \qquad\qquad \mathrm{roles}(T \rightarrow_\rho T') = \mathrm{roles}(T) \cup \mathrm{roles}(T') \cup \rho$$

$$\text{roles}(()@R) = \{R\} \qquad \text{roles}(T + T') = \text{roles}(T \times T') = \text{roles}(T) \cup \text{roles}(T')$$

A choreography term M may involve more roles besides those listed in its type. For instance, the choreography $\textbf{com}_{S,R}\ ()@S$ has type $()@R$ but involves also role S.

A key concern of choreographic languages is knowledge of choice: the property that when a choreography chooses between alternative branches (as with our **case** primitive), all roles that need to behave differently in the branches are properly informed via appropriate selections [8]. We give an example of how selections should be used, and postpone a formal discussion of how knowledge of choice is checked for to our presentation of Endpoint Projection.

▶ **Example 3** (Remote Map). The choreography below defines a remote map function, where f (available at role R) is applied to all elements of a list (available at role S). It consists of two functions: remoteFunction, which applies f to a single element, and remoteMap, which iterates this application over the input list.

```
remoteFunction = λf : Int@R → Int@R. λval : Int@S. com_{R,S} (f (com_{S,R} val))

remoteMap = λf : Int@R → Int@R. λlist : ListInt@S.
  case list of
    Inl x ⇒ select_{S,R} stop ()@S;
    Inr x ⇒ select_{S,R} again (cons (remoteFunction f (fst list)) (remoteMap f (snd list)))
```

Here, $\mathsf{ListInt}@S$ is the recursive type satisfying $\mathsf{ListInt}@S = ()@S + (\mathsf{Int}@S \times \mathsf{ListInt}@S)$ and, for simplicity, a primitive type for integers is assumed. When we introduce typing judgements, we will show how to work with this kind of types.

Notice how the **case** is evaluated on data at role S, so that role is the only one initially knowing which branch has been chosen. Each branch, however, starts with a selection from role S to role R. Since R receives a different label in the two branches, respectively stop and again, it can use this information to figure out whether it should terminate (stop) or the choreography continues (again): from its point of view, R is reactively handling a stream. ◁

We can encode conditional statements in the standard way: we define a type $\mathsf{Bool}@R$ as $()@R + ()@R$, and **if** M **then** M' **else** M'' as an abbreviation for **case** M **of** **Inl** $x \Rightarrow M'$**;** **Inr** $x \Rightarrow M''$.

Free and bound variables are defined as expected, noting that x and y are bound in **case** M **of** **Inl** $x \Rightarrow M'$**;** **Inr** $y \Rightarrow M''$. We write $\text{fv}(M)$ for the set of free variables in choreography M, and likewise for types. A choreography M is closed if $\text{fv}(M) = \emptyset$. The formal definition is given in Appendix A.

Typing

We now show how to type choreographies following the intuitions already given earlier. Typing judgements have the form $\Theta; \Sigma; \Gamma \vdash M : T$, where: Θ the set of roles that can be used for typing M; Σ is collection of type definitions i.e. expressions of the form $t_\rho = T$; and Γ is a typing environment for variables (and choreography names) that are free in M. We further require that a type name t is defined at most once in Σ, that definitions are contractive, and that $\text{roles}(T) = \rho$ for any $t_\rho = T \in \Sigma$. We call $\Theta; \Sigma; \Gamma$ jointly a *typing context*. Many of the rules resemble those for simply typed λ-calculus, but with roles added, and the additional requirements that only the roles in the type are used in the term being typed. We include some representative ones in Figure 1 (the complete set of typing rules is given in Appendix A).

$$\frac{x : T \in \Gamma \quad \mathrm{roles}(T) \subseteq \Theta}{\Theta;\Sigma;\Gamma \vdash x : T}\,[\textsc{TVar}] \qquad \frac{f : T \in \Gamma \quad \mathrm{roles}(T) \subseteq \Theta}{\Theta;\Sigma;\Gamma \vdash f : T}\,[\textsc{TDef}]$$

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

$$\frac{R, S \in \Theta \quad \mathrm{roles}(T) = \{S\}}{\Theta;\Sigma;\Gamma \vdash \mathbf{com}_{S,R} : T \to_\emptyset T[S := R]}\,[\textsc{TCom}] \qquad \frac{\Theta;\Sigma;\Gamma \vdash M : T \quad R, S \in \Theta}{\Theta;\Sigma;\Gamma \vdash \mathbf{select}_{S,R}\; l\; M : T}\,[\textsc{TSel}]$$

$$\frac{\Theta;\Sigma;\Gamma \vdash M : T' \quad \{T = T', T' = T\} \cap \Sigma \neq \emptyset}{\Theta;\Sigma;\Gamma \vdash M : T}\,[\textsc{TEq}]$$

Figure 1 Typing rules for Chorλ (representative selection).

Rules TVar, TDef, and TAbs exemplify how role checks are added to the standard typing rules for simply typed λ-calculus. Rule TCom types communication actions, moving subterms that were placed at role S to role R ($T[S := R]$ is the type expression obtained by replacing S with R). Rule TSel types selections as no-ops, again checking that the sender and receiver of the selection are legal roles. Rule TEq allows rewriting a type according to Σ in order to mimic recursive types (see Example 3).

We also write $\Theta;\Sigma;\Gamma \vdash D$ to denote that a set of definitions D, mapping names to choreographies, is well-typed. Sets of definitions play a key role in the semantics of choreographies, and can be typed by the rule below.

$$\frac{\forall f \in \mathsf{domain}(D) \quad f : T \in \Gamma \quad \Theta;\Sigma;\Gamma \vdash D(f) : T}{\Theta;\Sigma;\Gamma \vdash D}\,[\textsc{TDefs}]$$

▶ **Example 4.** The set of definitions in Example 3 can be typed in the typing context $\Theta;\Sigma;\Gamma$ where $\Theta = \{R, S\}$, $\Sigma = \{\mathsf{ListInt}@S = ()@S + (\mathsf{Int}@S \times \mathsf{ListInt}@S)\}$ and $\Gamma = \{\mathsf{remoteFunction} : (\mathsf{Int}@\mathsf{R} \to \mathsf{Int}@R) \to (\mathsf{Int}@S \to \mathsf{Int}@S), \mathsf{remoteMap} : (\mathsf{Int}@R \to \mathsf{Int}@R) \to (\mathsf{ListInt}@S \to \mathsf{ListInt}@S)\}$. ◁

Semantics

Chorλ comes with a reduction semantics that captures the essential ingredients of the calculi that inspired it: β- and ι-reduction, from λ-calculus, and the usual reduction rules for communications and selections. Some representative rules are given in Figure 2.

Rules AppAbs, App1, and App2 implement a call-by-value λ-calculus. Rules Case and CaseL and its counterpart CaseR implement ι-reductions for sums, and likewise for rules Proj1 and Proj2 wrt pairs. The communication rule Com changes the associated role of a value, moving it from S to R, while the selection rule Sel implements selection as a no-op. Rule Def allows reductions to use choreographies defined in D.

As stated earlier, we focus on closed choreographies. Our first result shows that closed choreographies remain closed under reductions.

▶ **Proposition 5.** *Let M be a closed choreography. If $M \to_D M'$ then M' is closed.*

Proof. Straightforward from the semantics. ◀

One of the hallmark properties of choreographies is that well-typed choreographies should continue to reduce until they reach a value. We split this result in two independent statements.

$$\lambda x : T.M\ V \rightarrow_D M[x := V]\ [\textsc{AppAbs}]$$

$$\frac{M \rightarrow_D M'}{M\ N \rightarrow_D M'\ N}\ [\textsc{App1}] \qquad \frac{N \rightarrow_D N'}{V\ N \rightarrow_D V\ N'}\ [\textsc{App2}]$$

$$\frac{N \rightarrow_D N'}{\textsf{case}\ N\ \textsf{of Inl}\ x \Rightarrow M;\ \textsf{Inr}\ x' \Rightarrow M' \rightarrow_D \textsf{case}\ N'\ \textsf{of Inl}\ x \Rightarrow M;\ \textsf{Inr}\ x' \Rightarrow M'}\ [\textsc{Case}]$$

$$\textsf{case Inl}\ V\ \textsf{of Inl}\ x \Rightarrow M;\ \textsf{Inr}\ x' \Rightarrow M' \rightarrow_D M[x := V]\ [\textsc{CaseL}]$$

$$\textsf{fst Pair}\ V\ V' \rightarrow_D V\ [\textsc{Proj1}] \qquad f \rightarrow_D D(f)\ [\textsc{Def}]$$

$$\textsf{com}_{S,R}\ V \rightarrow_D V[S := R]\ [\textsc{Com}] \qquad \textsf{select}_{S,R}\ \ell\ M \rightarrow_D M\ [\textsc{Sel}]$$

Figure 2 Semantics of Chorλ.

▶ **Theorem 6** (Progress). *Let M be a closed choreography and D a collection of named choreographies with all the necessary definitions for M. If there exists a typing context $\Theta; \Sigma; \Gamma$ such that $\Theta; \Sigma; \Gamma \vdash M : T$ and $\Theta; \Sigma; \Gamma \vdash D$, then either M is a value (and $M \not\rightarrow_D$) or there exists a choreography M' such that $M \rightarrow_D M'$.*

Proof. Straightforward by induction on the typing derivation of $\Theta; \Sigma; \Gamma \vdash M : T$. ◀

▶ **Theorem 7** (Type Preservation). *Let M be a closed choreography. If there exists a typing context $\Theta; \Sigma; \Gamma$ such that $\Theta; \Sigma; \Gamma \vdash M : T$ and $\Theta; \Sigma; \Gamma \vdash D$, then $\Theta; \Sigma; \Gamma \vdash M' : T$ for any M' such that $M \rightarrow_D M'$.*

Proof. Straightforward from the typing and semantic rules. ◀

Combining these results, we conclude that if M is a well-typed, closed, choreography, then either M is a value or M reduces to some well-typed, closed choreography M'. Since M' still satisfies the hypotheses of the above results, either it is a value or it can reduce.

4 Endpoint Projection

In order to implement a choreography, one must determine how each individual role behaves. We introduce a process calculus to specify these behaviours, and show how to generate implementations of choreographies automatically. In this context, roles are implemented by *processes* and we use the two terms interchangeably.

Process Language

▶ **Definition 8.** *The syntax of behaviours, local values and local types is defined by the following grammar.*

$$B ::= L \mid f \mid B\ B \mid \textsf{case}\ B\ \textsf{of Inl}\ x \Rightarrow B;\ \textsf{Inr}\ x \Rightarrow B \mid \oplus_R\ \ell\ B \mid \&_R\{\ell_1 : B_1, \dots \ell_n : B_n\}$$
$$L ::= x \mid \lambda x : T.B \mid \textsf{Inl}\ L \mid \textsf{Inr}\ L \mid \textsf{fst} \mid \textsf{snd} \mid \textsf{Pair}\ L\ L \mid () \mid \textsf{recv}_R \mid \textsf{send}_R$$
$$T ::= T \rightarrow T \mid T + T \mid T \times T \mid () \mid t$$

Behaviours correspond directly to local counterparts of choreographic actions. The terms from the λ-calculus are unchanged (except that there are no role annotations now); the choreographic actions generate two terms each. Selection yields the *offer* term $\&_R\{\ell_1 : B_1, \dots \ell_n :$

$$\mathsf{send}_R\ L \xrightarrow{\mathsf{send}_R\ v}_d ()\ [\text{NSend}] \qquad \mathsf{recv}_R\ () \xrightarrow{\mathsf{recv}_R\ v}_d L\ [\text{NRecv}]$$

$$\frac{B \xrightarrow{\mathsf{send}_R\ L}_{\mathbb{D}(S)} B_1' \quad B_2 \xrightarrow{\mathsf{recv}_S\ L[S:=R]}_{\mathbb{D}(R)} B_2'}{S[B_1] \mid R[B_2] \xrightarrow{\tau_{S,R}}_{\mathbb{D}} S[B_1'] \mid R[B_2']}\ [\text{NCom}]$$

$$\oplus_R\ \ell\ B \xrightarrow{\oplus_R\ \ell}_d B\ [\text{NCho}] \qquad \&_R\{\ell_1 : B_1, \dots, \ell_n : B_n\} \xrightarrow{\&_R \ell_i}_d B_i\ [\text{NOff}]$$

$$\frac{B_1 \xrightarrow{\oplus_R\ \ell}_{\mathbb{D}(S)} B_1' \quad B_2 \xrightarrow{\&_S\ \ell}_{\mathbb{D}(R)} B_2'}{S[B_1] \mid R[B_2] \xrightarrow{\tau_{S,R}}_{\mathbb{D}} S[B_1'] \mid R[B_2']}\ [\text{NSel}]$$

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

$$\frac{B \rightarrow_d B''}{B\ B' \xrightarrow{\tau}_d B''\ B'}\ [\text{NApp1}] \qquad \frac{B \rightarrow_d B'}{L\ B \xrightarrow{\tau}_d L\ B'}\ [\text{NApp2}]$$

$$\frac{B \xrightarrow{\tau}_{\mathbb{D}(R)} B'}{R[B] \xrightarrow{\tau_R}_{\mathbb{D}} R[B']}\ [\text{NPro}] \qquad \frac{\mathcal{N} \xrightarrow{\tau_R}_{\mathbb{D}} \mathcal{N}''}{\mathcal{N} \mid \mathcal{N}' \xrightarrow{\tau_R}_{\mathbb{D}} \mathcal{N}'' \mid \mathcal{N}'}\ [\text{NPar}]$$

Figure 3 Network semantics (representative rules).

$$\frac{\Sigma; \Gamma \vdash B : T}{\Sigma; \Gamma \vdash \oplus_R\ \ell\ B : T}\ [\text{NTChor}] \qquad \frac{\Sigma; \Gamma \vdash B_i : T \text{ for } 1 \leq i \leq n}{\Sigma; \Gamma \vdash \&_R\{\ell_1 : B_1, \dots \ell_n : B_n\} : T}\ [\text{NTOff}]$$

$$\Sigma; \Gamma \vdash \mathsf{send}_R : T \rightarrow ()\ [\text{NTSend}] \qquad \Sigma; \Gamma \vdash \mathsf{recv}_R : () \rightarrow T\ [\text{NTRecv}]$$

Figure 4 Typing rules for behaviours (representative rules).

$B_n\}$, which offers a number of different ways it can continue for another process R to choose from, and the *choice* term $\oplus_R\ \ell\ B$, which directs R to continue as the behaviour labelled ℓ. Likewise, communication has been divided into a *send* to R action, send_R, and a *receive* from R action, recv_R. Types are defined exactly as for choreographies, but without roles.

▶ **Definition 9.** *A network $\mathcal{N}$ is a finite map from a set of processes to behaviours.*

The parallel composition of two networks $\mathcal{N}$ and $\mathcal{N}'$ with disjoint domains, $\mathcal{N} \mid \mathcal{N}'$, simply assigns to each process its behaviour in the network defining it. Any network is equivalent to a parallel composition of networks with singleton domain, and therefore we often write $R_1[B_1] \mid \dots \mid R_n[B_n]$ for the network where process R_i has behaviour B_i.

The semantics of networks is given as a labelled transition system. Representative rules that define transitions are included in Table 3. Most of these rules are similar to the ones for choreographies; the difference is that communications and selections now require synchronisation between the processes implementing the two local actions. This is achieved by matching the appropriate labels on the reductions.

Our calculus includes a typing system for typing behaviours. Typing judgements now have the form $\Sigma; \Gamma \vdash B : T$. Most of the rules are direct counterparts to those in Figure 1, obtained by removing Θ and any side conditions involving roles. The new rules are given in Figure 4.

Endpoint Projection (EPP)

We now have the necessary ingredients to define the endpoint projection (EPP) of a choreography M for an individual role R given a typing derivation showing that $\Theta;\Sigma;\Gamma \vdash M : T$ for some type T. Formally, the definition of EPP depends on this derivation; but to keep notation simple we write $[\![M]\!]_R$.

Intuitively, the projection simply translates each choreography action to the corresponding local behaviour. For example, a communication action projects to a send (for the sender), a receive (for the receiver), or a unit (for the remaining processes). In order to define EPP precisely, we need a few additional ingredients, which we briefly describe.

Projecting a term requires knowing the roles involved in its type. This is implicitly given in the derivation provided to EPP. It can easily be shown by structural induction that, if the derivation contains two different typing judgements for the same term, then the roles involved in that term's type are the same. So we write without ambiguity $\mathrm{roles}(\mathrm{type}(M))$ for this set of roles.

The second ingredient concerns knowledge of choice. When projecting **case** M **of Inl** $x \Rightarrow M'$**; Inr** $y \Rightarrow M''$, roles 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. This is done by means of a standard partial *merge* operator ($\sqcup$), adapted from [6, 13, 22], whose key property is

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

and which is homomorphically defined for the remaining constructs (see the Appendix A for the full definition). Merging of incompatible behaviours is undefined.

▶ **Definition 10.** *The EPP of a choreography M for role R is defined by the rules in Figure 5.*

To project a network from a choreography, we therefore project the choreography for each role and combine the results in parallel: $[\![M]\!] = \prod_{R\in\mathrm{roles}(M)} R\,[[\![M]\!]_R]$.

Intuitively, projecting a value to a role that is not involved in it returns a unit. More complex choreographies, though, may involve roles that are not shown in their type. This explains the second clause for projecting an application: even if R does not appear in the type of M, it may participate in interactions inside M. A similar observation applies to the projection of **case**, where merging is also used.

Selections and communications follow the intuition given above, with one interesting detail: self-selections are ignored, and self-communications project to the identity function. This is different from many standard choreography calculi, where self-communications are not allowed – we do not want to impose this in Chorλ, since one of the planned future developments for this language is to add polymorphism.

Likewise, projecting a type yields () at any role not used in that type. (As a particular case, $()@R$ always gets projected as (), but for different reasons.) The projection of a set of function definitions maps choreography names to behaviours.

▶ **Proposition 11.** *Let M be a closed choreography. If $\Theta;\Sigma;\Gamma \vdash M : T$, then for any role R appearing in M, we have that $[\![\Sigma]\!]\,;[\![\Gamma]\!] \vdash [\![M]\!]_R : [\![T]\!]_R$, where $[\![\Sigma]\!]$ and $[\![\Gamma]\!]$ are defined by applying EPP to all the types occuring in those sets.*

Proof. Straightforward from the typing and projection rules. ◀

▶ **Example 12.** The projections of the choreographies in Example 3 are the following.

$$\llbracket M\ \ N \rrbracket_R = \begin{cases} \llbracket M \rrbracket_R\ \ \llbracket N \rrbracket_R & \text{if } R \in \text{roles}(\text{type}(M)) \\ (\lambda x : ().\, \llbracket N \rrbracket_R)\ \ \llbracket M \rrbracket_R & \text{for some } x \notin \text{fv}(N) \cup \text{fv}(M) \text{ otherwise} \end{cases}$$

$$\llbracket \lambda x : T. M \rrbracket_R = \begin{cases} \lambda x : \llbracket T \rrbracket_R \,.\, \llbracket M \rrbracket_R & \text{if } R \in \text{roles}(\text{type}(x : T.M)) \\ () & \text{otherwise} \end{cases}$$

$$\llbracket \textbf{case } M \textbf{ of Inl } x \Rightarrow N \textbf{; Inr } x' \Rightarrow N' \rrbracket_R =$$

$$\begin{cases} \textbf{case } \llbracket M \rrbracket_R \textbf{ of Inl } x \Rightarrow \llbracket N \rrbracket_R \textbf{; Inr } x' \Rightarrow \llbracket N' \rrbracket_R & \text{if } R \in \text{roles}(\text{type}(M)) \\ (\lambda x'' : ().\, \llbracket N \rrbracket_R \sqcup \llbracket N' \rrbracket_R)\ \ \llbracket M \rrbracket_R & \text{for some } x'' \notin \text{fv}(N) \cup \text{fv}(N') \\ & \text{otherwise} \end{cases}$$

$$\llbracket \textbf{select}_{S,S'}\ \ell\ M \rrbracket_R = \begin{cases} \oplus_{S'}\ \ell\ \ \llbracket M \rrbracket_R & \text{if } R = S \neq S' \\ \&_S\{\ell : \llbracket M \rrbracket_R\} & \text{if } R = S' \neq S \\ \llbracket M \rrbracket_R & \text{otherwise} \end{cases}$$

$$\llbracket \textbf{com}_{S,S'} \rrbracket_R = \begin{cases} \lambda x : \llbracket T \rrbracket_R \,.x & \text{if } R = S = S' \text{ and type}(\textbf{com}_{S,S'}) = T \to_\emptyset T \\ \textbf{send}_{S'} & \text{if } R = S \neq S' \\ \textbf{recv}_S & \text{if } R = S' \neq S \\ () & \text{otherwise} \end{cases}$$

$$\llbracket ()@S \rrbracket_R = () \qquad \llbracket f \rrbracket_R = f \qquad \llbracket x \rrbracket_R = \begin{cases} x & \text{if } R \in \text{roles}(\text{type}(x)) \\ () & \text{otherwise} \end{cases}$$

Types:

$$\llbracket ()@S \rrbracket_R = () \qquad \llbracket T \to_\rho T' \rrbracket_R = \begin{cases} \llbracket T \rrbracket_R \to \llbracket T' \rrbracket_R & \text{if } R \in \rho \cup \text{roles}(T) \cup \text{roles}(T') \\ () & \text{otherwise} \end{cases}$$

$$\llbracket t_\rho \rrbracket_R = \begin{cases} t & \text{if } R \in \rho \\ () & \text{otherwise} \end{cases} \qquad \llbracket T \times T' \rrbracket_R = \begin{cases} \llbracket T \rrbracket_R \times \llbracket T' \rrbracket_R & \text{if } R \in \text{roles}(T \times T') \\ () & \text{otherwise} \end{cases}$$

Definitions:

$$\llbracket D \rrbracket\,(R) = \{ f \mapsto \llbracket D(f) \rrbracket_R \mid f \in \mathsf{domain}(D) \}$$

Figure 5 Projecting a choreography in Chorλ onto a role

$$\begin{aligned}
&\llbracket D(\mathsf{remoteFunction}) \rrbracket_S = \lambda f : ().\ \lambda \mathit{val} : \mathsf{Int}.\ \textbf{recv}_R\ ((\lambda x : ().\ \textbf{send}_R\ \mathit{val})\ ())\\
&\llbracket D(\mathsf{remoteFunction}) \rrbracket_R = \lambda f : (\mathsf{Int} \to \mathsf{Int}).\ \lambda \mathit{val} : ().\ \textbf{send}_S\ (f\ (\textbf{recv}_S\ ()))\\
\\
&\llbracket D(\mathsf{remoteMap}) \rrbracket_S = \lambda f : ().\ \lambda \mathit{list} : \mathsf{ListInt}.\\
&\quad \textbf{case } \mathit{list} \textbf{ of Inl } x \Rightarrow (\oplus_R\ \mathsf{stop}\ ())\,\textbf{;}\\
&\qquad\qquad\qquad \textbf{Inr } x \Rightarrow (\oplus_R\ \mathsf{again}\ (\mathsf{cons}\ (\mathsf{remoteFunction}\ ()\ (\textbf{fst}\ \mathit{list}))\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{remoteMap}\ ()\ (\textbf{snd}\ \mathit{list}))))\\
\\
&\llbracket D(\mathsf{remoteMap}) \rrbracket_R = \lambda f : \mathsf{Int} \to \mathsf{Int}.\ \lambda \mathit{list} : ().\\
&\quad \&_S\{\mathsf{stop} : (), \mathsf{again} : (\lambda x : ().\ \mathsf{remoteMap}\ f\ ())\ (\mathsf{remoteFunction}\ f\ ())\}
\end{aligned}$$

This example illustrates the key features discussed in the text: projection of communications as two dual actions; the use of merge in the projection of **case**; and the way function applications are projected when the role does not appear in the function's 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 iff its projection can execute the same action, and both transition to new terms in the same relation. However, this is not completely true: if a choreography C reduces by rule CASE, then the result has fewer branches than the network obtained by performing the corresponding reduction in the projection of C.

In order to capture this, we revert to the notion of pruning [6, 13], defined by $B \sqsupseteq B'$ iff $B \sqcup B' = B$. Intuitively, if $B \sqsupseteq B'$, then B offers the same and possibly more behaviours than B'. This notion extends to networks by defining $\mathcal{N} \sqsupseteq \mathcal{N}'$ to mean that, for any role R, $\mathcal{N}(R) \sqsupseteq \mathcal{N}'(R)$.

▶ **Example 13.** Consider the choreography

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

Its projection for role S is $[\![C]\!]_S = (\lambda x : ().\&_R\{\mathsf{left} : 0, \mathsf{right} : 1\})\ ()$.

After entering the conditional in the choreography, C reduces to $C' = \textbf{select}_{R,S}$ left $0@S$, whereas if S executes the corresponding action its behaviour becomes $\&_R\{\mathsf{left} : 0, \mathsf{right} : 1\}$, which is not the projection of C'. However, $\&_R\{\mathsf{left} : 0, \mathsf{right} : 1\} \sqcup \&_R\{\mathsf{left} : 0\} = \&_R\{\mathsf{left} : 0, \mathsf{right} : 1\}$, so $[\![C']\!]_S$ is a pruning of this behaviour. ◁

▶ **Theorem 14** (Soundness). *Given a closed choreography M, if $M \to_D M'$ and $\Theta; \Sigma; \Gamma \vdash M : T$, then there exist networks $\mathcal{N}$ and $\mathcal{N}'$ such that: $[\![M]\!] \to^+_{[\![D]\!]} \mathcal{N}$; $[\![M']\!] \to^* \mathcal{N}'$; and $\mathcal{N} \sqsupseteq \mathcal{N}'$.*

Proof. By structural induction on the derivation of $M \to_D M'$. ◀

▶ **Theorem 15** (Completeness). *Given a closed choreography M, if $\Theta; \Sigma; \Gamma \vdash M : T$ and $[\![M]\!] \xrightarrow{\tau_{\mathbf{R}}}_{\mathbb{D}} \mathcal{N}$ for some network $\mathcal{N}$, then there exist a choreography M' and a network $\mathcal{N}'$ such that: $M \to M'$; $\mathcal{N} \to^* \mathcal{N}'$; and $\mathcal{N}' \sqsupseteq [\![M']\!]$.*

Proof. By structural induction on M. ◀

From Theorems 6, 7, 14, and 15, we get the following corollary, which states that a network derived from a well-typed closed choreography can continue to reduce until all roles contain only local values.

▶ **Corollary 16.** *Given a closed choreography M and a function environment D containing all the functions of M, if $\Theta; \Sigma; \Gamma \vdash M : T$ and $\Theta; \Sigma; \Gamma \vdash D$, then: whenever $[\![M]\!] \to^*_{[\![D]\!]} \mathcal{N}$ for some network $\mathcal{N}$, either there exists $\mathbf{R}$ such that $\mathcal{N} \xrightarrow{\tau_{\mathbf{R}}}_{[\![D]\!]} \mathcal{N}'$ or $\mathcal{N} = \prod_{R \in \mathrm{roles}(M)} R[L_R]$.*

5 An Illustrative Example: Secure Authentication

In this section, we illustrate more in depth the expressivity of Chorλ: we write a distributed authentication protocol inspired by the OpenID specification [35] and modularly combine it with the Diffie–Hellman protocol for key exchange [17] to secure its communications. Since we do not have polymorphism, our implementation is not completely generic; in particular, we can only communicate strings, and we need multiple implementations of the same function for different roles. For simplicity, we assume primitive types Int and String.

Our protocol has three roles: a Client, a Server, and an identity provider IP. Each role $R \in \{\mathsf{Client}, \mathsf{Server}, \mathsf{IP}\}$ has functions

$$\begin{aligned}
\mathsf{modPow}_R &: \mathsf{Int}@R \to \mathsf{Int}@R \to \mathsf{Int}@R \to \mathsf{Int}@R \\
\mathsf{encrypt}_R &: \mathsf{Int}@R \to \mathsf{String}@R \to \mathsf{String}@R \\
\mathsf{decrypt}_R &: \mathsf{Int}@R \to \mathsf{String}@R \to \mathsf{String}@R
\end{aligned}$$

used for computing powers with a given modulo, encrypting and decrypting messages, respectively. The implementation of these functions is immaterial for the presentation.

The Client also has functions

$$\begin{aligned}
\mathsf{username} &: \mathsf{Credentials}@\mathsf{Client} \to \mathsf{String}@\mathsf{Client} \\
\mathsf{password} &: \mathsf{Credentials}@\mathsf{Client} \to \mathsf{String}@\mathsf{Client} \\
\mathsf{calcHash} &: \mathsf{String}@\mathsf{Client} \to \mathsf{String}@\mathsf{Client} \to \mathsf{String}@\mathsf{Client}
\end{aligned}$$

computing, respectively, the username and password from a local type Credentials@Client (which can be implemented as a pair, for example), and the hash of a string with a given salt.

The identity provider IP in turn uses functions

$$\begin{aligned}
\mathsf{getSalt} &: \mathsf{String}@\mathsf{IP} \to \mathsf{String}@\mathsf{IP} \\
\mathsf{check} &: \mathsf{String}@\mathsf{IP} \to \mathsf{String}@\mathsf{IP} \to \mathsf{Bool}@\mathsf{IP} \\
\mathsf{createToken} &: \mathsf{String}@\mathsf{IP} \to \mathsf{String}@\mathsf{IP}
\end{aligned}$$

respectively for retrieving the salt, checking the hash, and creating a token for a given username.

We denote by Γ the set of typings of all the above functions.

The first step is implementing the Diffie–Hellman algorithm for each pair of roles. This is done by means of the following family of choreographies, indexed on pairs of roles.

$$\begin{array}{l}
\mathsf{diffieHellman}_{P,Q} = \\
\quad \lambda psk : \mathsf{Int}@P.\ \lambda qsk : \mathsf{Int}@Q.\ \lambda psg : \mathsf{Int}@P.\ \lambda qsg : \mathsf{Int}@Q.\ \lambda psp : \mathsf{Int}@P.\ \lambda qsp : \mathsf{Int}@Q. \\
\qquad (\lambda rk : \mathsf{Int}@P \times \mathsf{Int}@Q.\ \mathbf{pair}\ (\mathsf{modPow}_P\ psg\ (\mathbf{fst}\ rk)\ psp)\ (\mathsf{modPow}_Q\ qsg\ (\mathbf{snd}\ rk)\ qsp)) \\
\qquad\quad ((\lambda pk : \mathsf{Int}@P \times \mathsf{Int}@Q.\ \mathbf{pair}\ (\mathbf{com}_{Q,P}\ (\mathbf{snd}\ pk))\ (\mathbf{com}_{P,Q}\ (\mathbf{fst}\ pk))) \\
\qquad\qquad ((\lambda sk : \mathsf{Int}@P \times \mathsf{Int}@Q.\ \mathbf{pair}\ (\mathsf{modPow}_P\ psg\ (\mathbf{fst}\ sk)\ psp)\ (\mathsf{modPow}_Q\ qsg\ (\mathbf{snd}\ sk)\ qsp)) \\
\qquad\qquad\quad (\mathbf{Pair}\ psk\ qsk)))
\end{array}$$

This choreography applies the sequence of distributed transformations specified in the Diffie–Hellman key exchange algorithm to a pair of secret keys. We can check that

$$\begin{aligned}
&\Theta; \Sigma; \Gamma \vdash \mathsf{diffieHellman}_{\mathsf{P},\mathsf{Q}} : \\
&\qquad\qquad \mathsf{Int}@P \to \mathsf{Int}@Q \to \mathsf{Int}@P \to \mathsf{Int}@Q \to \mathsf{Int}@P \to \mathsf{Int}@Q \to \mathsf{Int}@P \times \mathsf{Int}@Q
\end{aligned}$$

The second ingredient is a family of choreographies to create pairs of secure channels between two roles, encrypting and decrypting messages using each role's function. These choreographies take a key as an argument, which is used for encrypting and decrypting.

$$\begin{array}{l}
\mathsf{makeSecureChannels}_{P,Q} = \lambda key : \mathsf{Int}@P \times \mathsf{Int}@Q. \\
\quad \mathbf{Pair}\ (\lambda val : \mathsf{String}@P.\ (\mathsf{decrypt}_Q\ (\mathbf{snd}\ key)\ (\mathbf{com}_{P,Q}\ (\mathsf{encrypt}_P\ (\mathbf{fst}\ key)\ val)))) \\
\qquad\quad (\lambda val : \mathsf{String}@Q.\ (\mathsf{decrypt}_P\ (\mathbf{fst}\ key)\ (\mathbf{com}_{Q,P}\ (\mathsf{encrypt}_Q\ (\mathbf{snd}\ key)\ val))))
\end{array}$$

Again, we can show that

$$\Theta; \Sigma; \Gamma \vdash \mathsf{makeSecureChannels}_{P,Q} : \\ (\mathsf{Int}@P \times \mathsf{Int}@Q) \to ((\mathsf{String}@P \to \mathsf{String}@Q) \times (\mathsf{String}@Q \to \mathsf{String}@P))$$

The third ingredient is an authentication protocol where Client and Server get a token from IP if authentication succeeds. We use if-then-else as syntactic sugar.

$$\begin{array}{l}
\mathsf{authenticate} = \lambda \mathit{credentials} : \mathsf{Credentials}@\mathsf{Client}. \\
\quad \lambda \mathit{comcip} : \mathsf{String}@\mathsf{Client} \to \mathsf{String}@\mathsf{IP}.\ \lambda \mathit{comipc} : \mathsf{String}@\mathsf{IP} \to \mathsf{String}@\mathsf{Client}. \\
\quad \lambda \mathit{comips} : \mathsf{String}@\mathsf{IP} \to \mathsf{String}@\mathsf{Server}. \\
\quad\quad ((\lambda \mathit{user} : \mathsf{String}@\mathsf{IP}.\ (\lambda \mathit{salt} : \mathsf{String}@\mathsf{Client}.\ (\lambda \mathit{hash} : \mathsf{String}@\mathsf{IP}. \\
\quad\quad\quad \textbf{if}\ \mathsf{check}\ \mathit{user}\ \mathit{hash}\ \textbf{then} \\
\quad\quad\quad\quad \textbf{select}_{\mathsf{IP},\mathsf{Client}}\ \mathsf{ok}\ (\textbf{select}_{\mathsf{IP},\mathsf{Server}}\ \mathsf{ok} \\
\quad\quad\quad\quad\quad (\lambda \mathit{token} : \mathsf{String}@\mathsf{IP}.\ \textbf{inl}\ (\textbf{pair}\ (\mathit{comipc}\ \mathit{token})\ (\mathit{comips}\ \mathit{token})))\ (\mathsf{createToken}\ \mathit{user})) \\
\quad\quad\quad \textbf{else} \\
\quad\quad\quad\quad \textbf{select}_{\mathsf{IP},\mathsf{Client}}\ \mathsf{ko}\ (\textbf{select}_{\mathsf{IP},\mathsf{Server}}\ \mathsf{ko}\ \textbf{inr}\ ()@\mathsf{IP})) \\
\quad\quad (\mathit{comcip}\ (\mathsf{calcHash}\ \mathit{salt}\ (\mathsf{password}\ \mathit{credentials})))) \\
\quad\quad (\mathit{comipc}\ (\mathsf{getSalt}\ \mathit{user}))) \\
\quad\quad (\mathit{comcip}\ (\mathsf{username}\ \mathit{credentials})))
\end{array}$$

This protocol is parameterised on three channels between the participants. Client sends their username to IP, who replies with the appropriate salt; Client then uses this to hash their password and send it to IP, who checks the result and either sends a token to both participants or returns a unit.

We can type this choreography as follows.

$$\begin{array}{l}
\Theta; \Sigma; \Gamma \vdash \mathsf{authenticate} : \mathsf{Credentials}@\mathsf{Client} \to \\
(\mathsf{String}@\mathsf{Client} \to \mathsf{String}@\mathsf{IP}) \to (\mathsf{String}@\mathsf{IP} \to \mathsf{String}@\mathsf{Client}) \to \\
\qquad (\mathsf{String}@\mathsf{IP} \to \mathsf{String}@\mathsf{Server}) \to ((\mathsf{String}@\mathsf{Client} \times \mathsf{String}@\mathsf{Server}) + ()@\mathsf{IP})
\end{array}$$

We now use these choreographies as function definitions, and write a choreography where the three roles perform a secure authentication, by using channels created by makeSecureChannels backed by an encryption key provided by diffieHellman.

$$\begin{array}{l}
(\lambda k1 : \mathsf{Int}@\mathsf{Client} \times \mathsf{Int}@\mathsf{IP}. \lambda k2 : \mathsf{Int}@\mathsf{IP} \times \mathsf{Int}@\mathsf{Server}. \\
\quad (\lambda c1 : (\mathsf{String}@\mathsf{Client} \to \mathsf{String}@\mathsf{IP}) \times (\mathsf{String}@\mathsf{IP} \to \mathsf{String}@\mathsf{Client}). \\
\quad \lambda c2 : (\mathsf{String}@\mathsf{IP} \to \mathsf{String}@\mathsf{Server}) \times (\mathsf{String}@\mathsf{Server} \to \mathsf{String}@\mathsf{IP}). \\
\quad\quad (\lambda t : \mathsf{String}. \\
\quad\quad\quad \textbf{case}\ t\ \textbf{of} \\
\quad\quad\quad\quad \textbf{Inl}\ x \Rightarrow \texttt{"Authentication successful"}@\mathsf{Client} \\
\quad\quad\quad\quad \textbf{Inr}\ x \Rightarrow \texttt{"Authentication failed"}@\mathsf{Client}) \\
\quad\quad (\mathsf{authenticate}\ (\textbf{fst}\ c1)\ (\textbf{snd}\ c1)\ (\textbf{fst}\ c2))) \\
\quad (\mathsf{makeSecureChannels}_{\mathsf{IP},\mathsf{Server}}\ k2)\ (\mathsf{makeSecureChannels}_{\mathsf{Client},\mathsf{IP}}\ k1)) \\
(\mathsf{diffieHellman}_{\mathsf{IP},\mathsf{Server}}\ \mathit{ipsk}\ \mathit{ssk}\ \mathit{ipsg}\ \mathit{ssg}\ \mathit{ipsp}\ \mathit{ssp})\ (\mathsf{diffieHellman}_{\mathsf{Client},\mathsf{IP}}\ \mathit{csk}\ \mathit{ipsk}\ \mathit{csg}\ \mathit{ipsg}\ \mathit{csp}\ \mathit{ipsp})
\end{array}$$

In this example, the protocol simply tells the client whether authentication has succeeded; but this could of course be replaced with more meaningful code.

Let Γ' be obtained from Γ by adding the types for $\mathsf{diffieHellman}_{P,Q}$, $\mathsf{makeSecureChannels}_{P,Q}$ and authenticate given earlier. Then this choreography has type String@Client in the typing environment $\Theta; \Sigma; \Gamma'$.

To complete this section, we illustrate how implementations of the individual roles can be obtained by projecting each choreography. Projecting our choreographies $\mathsf{diffieHellman}_{P,Q}$ and $\mathsf{makeSecureChannels}_{P,Q}$ for role P yields the following behaviours.

$$
\begin{array}{l}
\llbracket D(\mathsf{diffieHellman}_{P,Q})\rrbracket_P = \lambda psk : \mathsf{Int}.\ \lambda qsk : ().\ \lambda psg : \mathsf{Int}.\ \lambda qsg : ().\ \lambda psp : \mathsf{Int}.\ \lambda qsp : ().\\
\ (\lambda rk : \mathsf{Int} \times ().\ \mathbf{pair}\ (\mathsf{modPow}_P\ psg\ (\mathbf{fst}\ rk)\ psp)\ ((\lambda x : ().())\ ((\lambda y : ().\ \mathbf{snd}\ rk)\ ((\lambda z : ().\ ())\ ()))))\\
\quad ((\lambda pk : \mathsf{Int} \times ().\ \mathbf{pair}\ (\mathbf{recv}_Q\ (\mathbf{snd}\ pk))\ (\mathbf{send}_Q\ (\mathbf{fst}\ pk)))\\
\quad\ ((\lambda sk : \mathsf{Int} \times ().\ \mathbf{pair}\ (\mathsf{modPow}_P\ psg\ (\mathbf{fst}\ sk)\ psp)\ ((\lambda x : ().())\ ((\lambda y : ().\mathbf{snd}\ rk)\ ((\lambda z : ().())\ ()))))\\
\qquad (\mathbf{Pair}\ psk\ ())))
\end{array}
$$

$$
\begin{array}{l}
\llbracket D(\mathsf{makeSecureChannels}_{P,Q})\rrbracket_P = \lambda key : \mathsf{Int} \times ().\\
\quad \mathbf{Pair}\ (\lambda val : \mathsf{String}.\ (\lambda x : ().\ (\mathbf{send}_Q\ (\mathsf{encrypt}_P\ (\mathbf{fst}\ key)\ val)))\ ((\lambda y : ().\ (\mathbf{snd}\ key))\ ()))\\
\qquad\quad (\lambda val : ().\ (\mathsf{decrypt}_P\ (\mathbf{fst}\ key)\ (\mathbf{recv}_Q\ ((\lambda x : ().\ ())\ ((\lambda y : ().\ (\mathbf{snd}\ key))\ ()))))))
\end{array}
$$

By reducing terms containing only units, we obtain the more readable

$$
\begin{array}{l}
\llbracket \mathsf{diffieHellman}_{P,Q}\rrbracket_P \to^* \lambda psk : \mathsf{Int}.\ \lambda qsk : ().\ \lambda psg : \mathsf{Int}.\ \lambda qsg : ().\ \lambda psp : \mathsf{Int}.\ \lambda qsp : ().\\
\quad (\lambda rk : \mathsf{Int} \times ().\ \mathbf{pair}\ (\mathsf{modPow}_P\ psg\ (\mathbf{fst}\ rk)\ psp)\ ())\\
\qquad ((\lambda pk : \mathsf{Int} \times ().\ \mathbf{pair}\ (\mathbf{recv}_Q\ (\mathbf{snd}\ pk))\ (\mathbf{send}_Q\ (\mathbf{fst}\ pk)))\\
\qquad\quad ((\lambda sk : \mathsf{Int} \times ().\ \mathbf{pair}\ (\mathsf{modPow}_P\ psg\ (\mathbf{fst}\ sk)\ psp)\ ())\\
\qquad\qquad (\mathbf{Pair}\ psk\ qsk)))
\end{array}
$$

$$
\begin{array}{l}
\llbracket \mathsf{makeSecureChannels}_{P,Q}\rrbracket_P \to^* \lambda key : \mathsf{Int} \times ().\\
\quad \mathbf{Pair}\ (\lambda val : \mathsf{String}.\ (\mathbf{send}_Q\ (\mathsf{encrypt}_P\ (\mathbf{fst}\ key)\ val)))\\
\qquad\quad (\lambda val : ().\ (\mathsf{decrypt}_P\ (\mathbf{fst}\ key)\ (\mathbf{recv}_Q\ ())))
\end{array}
$$

In turn, authenticate has the following projections for the client and the server.

$$
\begin{array}{l}
\llbracket D(\mathsf{authenticate})\rrbracket_{\mathsf{Client}} = \lambda credentials : \mathsf{Credentials}.\\
\quad \lambda comcip : \mathsf{String} \to ().\ \lambda comipc : () \to \mathsf{String}.\ \lambda comips : ().\\
\qquad ((\lambda user : ().\ (\lambda salt : \mathsf{String}.\ (\lambda hash : ().\\
\qquad\quad (\lambda x : ().\ \&_{\mathsf{IP}}\{\\
\qquad\qquad \mathsf{ok} : (\lambda token : ().\ \mathbf{inl}\ (\mathbf{pair}\ (comipc\ ())\ ((\lambda y : ().\ ())\ ())))\ ((\lambda z : ().\ ())\ ()),\\
\qquad\qquad \mathsf{ko} : \mathbf{inr}\ ()\})\\
\qquad\quad\ ((\lambda y : ().\ ())\ ((\lambda z : ().\ ())\ ()))))\\
\qquad\ (comcip\ (\mathsf{calcHash}\ salt\ (\mathsf{password}\ credentials))))\\
\qquad\ (comipc\ ((\lambda x : ().\ ())\ ())))\\
\qquad\ (comcip\ (\mathsf{username}\ credentials)))\\
\to^* \lambda credentials : \mathsf{Credentials}.\ \lambda comcip : \mathsf{String} \to ().\ \lambda comipc : () \to \mathsf{String}.\ \lambda comips : ().\\
\qquad ((\lambda user : ().\ (\lambda salt : \mathsf{String}.\ (\lambda hash : ().\ \&_{\mathsf{IP}}\{\mathsf{ok} : \mathbf{inl}\ (\mathbf{pair}\ (comipc\ ())\ ()), \mathsf{ko} : \mathbf{inr}\ ()\})\\
\qquad\ (comcip\ (\mathsf{calcHash}\ salt\ (\mathsf{password}\ credentials))))\\
\qquad\ (comipc\ ()))\\
\qquad\ (comcip\ (\mathsf{username}\ credentials)))
\end{array}
$$

$$
\begin{array}{l}
\llbracket D(\mathsf{authenticate})\rrbracket_{\mathsf{Server}} = \lambda credentials : ().\ \lambda comcip : ().\ \lambda comipc : ().\ \lambda comips : () \to \mathsf{String}.\\
\qquad ((\lambda user : ().\ (\lambda salt : ().\ (\lambda hash : ().\\
\qquad\quad (\lambda x : ().\ \&_{\mathsf{IP}}\{\\
\qquad\qquad \mathsf{ok} : (\lambda token : ().\ \mathbf{inl}\ (\mathbf{pair}\ ((\lambda y : ().\ ())\ ())\ (comips\ ())))\ ((\lambda z : ().\ ())\ ()),\\
\qquad\qquad \mathsf{ko} : \mathbf{inr}\ ()\}\\
\qquad\quad\ ((\lambda y : ().\ ())\ ((\lambda z : ().\ ())\ ()))))\\
\qquad\ ((\lambda x : ().\ ((\lambda y : ().\ ((\lambda z : ().\ ())\ ()))\ ((\lambda w : ().\ ())\ ())))\ ()))\\
\qquad\ ((\lambda x : ().\ ((\lambda y : ().\ ())\ ()))\ ()))\\
\qquad\ ((\lambda x : ().\ ((\lambda y : ().\ ())\ ()))\ ()))\\
\to^* \lambda credentials : ().\ \lambda comcip : ().\ \lambda comipc : ().\ \lambda comips : () \to \mathsf{String}.\\
\qquad ((\lambda user : ().\ (\lambda salt : ().\ (\lambda hash : ().\ \&_{\mathsf{IP}}\{\mathsf{ok} : \mathbf{inl}\ (\mathbf{pair}\ ()\ (comips\ ())), \mathsf{ko} : \mathbf{inr}\ ()\})\ ())\ ())\ ())
\end{array}
$$

It is simple to check that the types of these projections are indeed the projections of the types of the original choreographies.

References

1 Elvira Albert and Ivan Lanese, editors. *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, volume 9688 of *Lecture Notes in Computer Science*. Springer, 2016.

2 Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Inference of message sequence charts. In Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf, editors, *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000*, pages 304–313. ACM, 2000. `doi:10.1145/337180.337215`.

3 Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. Behavioral types in programming languages. *Foundations and Trends in Programming Languages*, 3(2–3):95–230, 2016.

4 Samik Basu, Tevfik Bultan, and Meriem Ouederni. Deciding choreography realizability. In John Field and Michael Hicks, editors, *Procs. POPL*, pages 191–202. ACM, 2012. `doi:10.1145/2103656.2103680`.

5 Mario Bravetti and Gianluigi Zavattaro. Towards a unifying theory for choreography conformance and contract compliance. In Markus Lumpe and Wim Vanderperren, editors, *Software Composition, 6th International Symposium, SC 2007, Braga, Portugal, March 24-25, 2007, Revised Selected Papers*, volume 4829 of *Lecture Notes in Computer Science*, pages 34–50. Springer, 2007. `doi:10.1007/978-3-540-77351-1\_4`.

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

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

8 Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party sessions. In *Formal Techniques for Distributed Systems*, pages 1–28. Springer, 2011.

9 David Castro-Perez and Nobuko Yoshida. Compiling first-order functions to session-typed parallel code. In Louis-Noël Pouchet and Alexandra Jimborean, editors, *CC '20: 29th International Conference on Compiler Construction, San Diego, CA, USA, February 22-23, 2020*, pages 143–154. ACM, 2020. `doi:10.1145/3377555.3377889`.

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

11 Luís Cruz-Filipe and Fabrizio Montesi. Choreographies in practice. In Albert and Lanese [1], pages 114–123. `doi:10.1007/978-3-319-39570-8_8`.

12 Luís Cruz-Filipe and Fabrizio Montesi. Procedural choreographic programming. In *FORTE*, volume 10321 of *Lecture Notes in Computer Science*, pages 92–107. Springer, 2017.

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

14 Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic choreographies: Theory and implementation. *Log. Methods Comput. Sci.*, 13(2), 2017. `doi:10.23638/LMCS-13(2:1)2017`.

15 Romain Demangeon and Kohei Honda. Nested protocols in session types. In Maciej Koutny and Irek Ulidowski, editors, *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings*,

volume 7454 of *Lecture Notes in Computer Science*, pages 272–286. Springer, 2012. doi:10.1007/978-3-642-32940-1_20.

16 Pierre-Malo Deniélou and Nobuko Yoshida. Multiparty session types meet communicating automata. In Helmut Seidl, editor, *Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, volume 7211 of *Lecture Notes in Computer Science*, pages 194–213. Springer, 2012. doi:10.1007/978-3-642-28869-2_10.

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

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

19 Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choreographies as objects. *CoRR*, abs/2005.09520, 2020. URL: https://arxiv.org/abs/2005.09520.

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

21 Kohei Honda, Aybek Mukhamedov, Gary Brown, Tzu-Chun Chen, and Nobuko Yoshida. Scribbling interactions with a formal foundation. In Raja Natarajan and Adegboyega K. Ojo, editors, *Distributed Computing and Internet Technology - 7th International Conference, ICDCIT 2011, Bhubaneshwar, India, February 9-12, 2011. Proceedings*, volume 6536 of *Lecture Notes in Computer Science*, pages 55–75. Springer, 2011. doi:10.1007/978-3-642-19056-8_4.

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

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

24 Intl. Telecommunication Union. Recommendation Z.120: Message Sequence Chart, 1996.

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

26 Alberto Lluch-Lafuente, Flemming Nielson, and Hanne Riis Nielson. Discretionary information flow control for interaction-oriented specifications. In Narciso Martí-Oliet, Peter Csaba Ölveczky, and Carolyn L. Talcott, editors, *Logic, Rewriting, and Concurrency*, volume 9200 of *Lecture Notes in Computer Science*, pages 427–450. Springer, 2015. doi:10.1007/978-3-319-23165-5_20.

27 Hugo A. López and Kai Heussen. Choreographing cyber-physical distributed control systems for the energy sector. In Ahmed Seffah, Birgit Penzenstadler, Carina Alves, and Xin Peng, editors, *Procs. SAC*, pages 437–443. ACM, 2017. doi:10.1145/3019612.3019656.

28 Hugo A. López, Flemming Nielson, and Hanne Riis Nielson. Enforcing availability in failure-aware communicating systems. In Albert and Lanese [1], pages 195–211. doi:10.1007/978-3-319-39570-8_13.

29 Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proc. of ASPLOS*, pages 329–339, 2008.

30 Fabrizio Montesi. *Choreographic Programming.* Ph.D. Thesis, IT University of Copenhagen, 2013.

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

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

33 Object Management Group. Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0/, 2011.

34 Peter W. O'Hearn. Experience developing and deploying concurrency analysis at facebook. In Andreas Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 56–70. Springer, 2018. doi:10.1007/978-3-319-99725-4_5.

35 OpenID Foundation. OpenID Specification. https://openid.net/developers/specs/, 2014.

36 Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *WWW*, pages 973–982, United States, 2007. IEEE Computer Society Press.

37 W3C. WS Choreography Description Language. http://www.w3.org/TR/ws-cdl-10/, 2004.

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

A Full definitions and proofs

▶ **Definition 17** (Free Variables). *Given a choreography M, the free variables of M, $\mathrm{fv}(M)$ are defined as:*

$$\begin{array}{ll}
\mathrm{fv}(N\ N') = \mathrm{fv}(N) \cup \mathrm{fv}(N') & \mathrm{fv}(\textbf{select}_{S,R}\ l\ M) = \mathrm{fv}(M) \\
\mathrm{fv}(x) = x & \mathrm{fv}(\lambda x : T.N) = \mathrm{fv}(N) \setminus \{x\} \\
\mathrm{fv}(()@R) = \emptyset & \mathrm{fv}(\textbf{com}_{S,R}) = \emptyset \\
\mathrm{fv}(f) = \emptyset & \mathrm{fv}(\textbf{Pair}\ V\ V') = \mathrm{fv}(V) \cup \mathrm{fv}(V') \\
\multicolumn{2}{l}{\mathrm{fv}(\textbf{case}\ N\ \textbf{of Inl}\ x \Rightarrow M; \textbf{Inr}\ y \Rightarrow M') = \mathrm{fv}(N) \cup (\mathrm{fv}(M) \setminus \{x\}) \cup (\mathrm{fv}(M') \setminus \{y\})} \\
\mathrm{fv}(\textbf{fst}) = \mathrm{fv}(\textbf{snd}) = \emptyset & \mathrm{fv}(\textbf{Inl}\ V) = \mathrm{fv}(\textbf{Inr}\ V) = \mathrm{fv}(V)
\end{array}$$

▶ **Definition 18** (Merging). *Given two behaviours B and B', $B \sqcup B'$ is defined as follows.*

$$B_1\ B_2 \sqcup B_1'\ B_2' = (B_1 \sqcup B_1')\ (B_2 \sqcup B_2')$$

$$\textbf{case}\ B_1\ \textbf{of Inl}\ x \Rightarrow B_2; \textbf{Inr}\ y \Rightarrow B_3 \sqcup \textbf{case}\ B_1'\ \textbf{of Inl}\ x \Rightarrow B_2'; \textbf{Inr}\ y \Rightarrow B_3' =$$

$$\textbf{case}\ (B_1 \sqcup B_1')\ \textbf{of Inl}\ x \Rightarrow (B_2 \sqcup B_2'); \textbf{Inr}\ y \Rightarrow (B_3 \sqcup B_3')$$

$$\oplus_R\ \ell\ B \sqcup \oplus_R\ \ell\ B' = \oplus_R\ \ell\ (B \sqcup B')$$

$$\&\{\ell_i : B_i\}_{i\in I} \sqcup \&\{\ell_j : B_j'\}_{j\in J} = \&\left(\{\ell_k : B_k \sqcup B_k'\}_{k\in I\cap J} \cup \{\ell_i : B_i\}_{i\in I\setminus J} \cup \{\ell_j : B_j'\}_{j\in J\setminus I}\right)$$

$$x \sqcup x = x \qquad \lambda x : T.B \sqcup \lambda x : T.B' = \lambda x : T.(B \sqcup B')$$

$$\textbf{fst} \sqcup \textbf{fst} = \textbf{fst} \qquad \textbf{snd} \sqcup \textbf{snd} = \textbf{snd}$$

$$\textbf{Inl}\ L \sqcup \textbf{Inl}\ L' = \textbf{Inl}\ (L \sqcup L') \qquad \textbf{Inr}\ L \sqcup \textbf{Inr}\ L' = \textbf{Inr}\ (L \sqcup L')$$

$$\textbf{Pair}\ L_1\ L_2 \sqcup \textbf{Pair}\ L_1'\ L_2' = \textbf{Pair}\ (L_1 \sqcup L_1')\ (L_2 \sqcup L_2') \qquad f \sqcup f = f$$

$$\textbf{recv}_R \sqcup \textbf{recv}_R = \textbf{recv}_R \qquad \textbf{send}_R \sqcup \textbf{send}_R = \textbf{send}_{send} R$$

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

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

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

$$\frac{\Theta; \Sigma; \Gamma \vdash M : T \quad S, R \in \Theta}{\Theta; \Sigma; \Gamma \vdash \textsf{select}_{S,R}\ l\ M : T}\ [\textsc{TSel}] \qquad \frac{f : T \in \Gamma \quad \mathrm{roles}(T) \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash f : T}\ [\textsc{TDef}]$$

$$\frac{R \in \Theta}{\Theta; \Sigma; \Gamma \vdash ()@R : ()@R}\ [\textsc{TUnit}] \qquad \frac{S, R \in \Theta \quad \mathrm{roles}(T) = S}{\Theta; \Sigma; \Gamma \vdash \textsf{com}_{S,R} : T \to_\emptyset T[S := R]}\ [\textsc{TCom}]$$

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

$$\frac{\mathrm{roles}(T \times T') \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \textsf{fst} : (T \times T') \to_\emptyset T}\ [\textsc{TProj1}] \qquad \frac{\mathrm{roles}(T \times T') \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \textsf{snd} : (T \times T') \to_\emptyset T'}\ [\textsc{TProj2}]$$

$$\frac{\Theta; \Sigma; \Gamma \vdash V : T \quad \mathrm{roles}(T + T') \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \textsf{Inl}\ V : (T + T')}\ [\textsc{TInl}] \qquad \frac{\Theta; \Sigma; \Gamma \vdash V : T' \quad \mathrm{roles}(T + T') \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash \textsf{Inr}\ V : (T + T')}\ [\textsc{TInr}]$$

$$\frac{\Theta; \Sigma; \Gamma \vdash M : T' \quad \{T = T', T' = T\} \cap \Sigma \neq \emptyset}{\Theta; \Sigma; \Gamma \vdash M : T}\ [\textsc{TEq}]$$

$$\frac{\forall f \in \mathsf{domain}(D) \quad f : T \in \Gamma \quad \Theta; \Sigma; \Gamma \vdash D(f) : T}{\Theta; \Sigma; \Gamma \vdash D}\ [\textsc{TDefs}]$$

Figure 6 Full set of typing rules for Chorλ.

A.1 Proof of Theorem 14

▶ **Lemma 19.** *Given a choreography M, if $\Theta; \Sigma; \Gamma \vdash M : T$ then for any role R in M, $[\![M]\!]_R = L$ if and only in $M = V$.*

Proof. Straightforward from the projection rules. ◀

▶ **Lemma 20.** *Given a type T, for any role $R \notin \mathrm{roles}(T)$, $[\![T]\!]_R = ()$.*

Proof. Straightforward from induction on T. ◀

▶ **Lemma 21.** *Given a value V, for any role $R \notin \mathrm{roles}(\mathrm{type}(V))$, $[\![V]\!]_R = ()$.*

Proof. Follows from Lemmas 19 and 20. ◀

▶ **Lemma 22.** *Given a choreography, M, if $\Theta; \Sigma; \Gamma \vdash M : T$ and $\Theta; \Sigma; \Gamma \vdash D$ and $R \notin \Theta$ then $[\![M]\!]_R \xrightarrow{\tau}{}^*_{[\![D]\!](R)} ()$.*

Proof. Straightforward from induction on $\Theta; \Sigma; \Gamma \vdash M : T$. ◀

Proof of Theorem 14. We prove this by structural induction on $M \to_D M'$.

$$\lambda x : T.M\ V \to_D M[x := V]\ [\textsc{AppAbs}]$$

$$\frac{M \to_D M'}{M\ N \to_D M'\ N}\ [\textsc{App1}] \qquad \frac{N \to_D N'}{V\ N \to_D V\ N'}\ [\textsc{App2}]$$

$$\frac{N \to_D N'}{\textsf{case}\ N\ \textsf{of Inl}\ x \Rightarrow M; \textsf{Inr}\ x' \Rightarrow M' \to_D \textsf{case}\ N'\ \textsf{of Inl}\ x \Rightarrow M; \textsf{Inr}\ x' \Rightarrow M'}\ [\textsc{Case}]$$

$$\textsf{case Inl}\ V\ \textsf{of Inl}\ x \Rightarrow M; \textsf{Inr}\ x' \Rightarrow M' \to_D M[x := V]\ [\textsc{CaseL}]$$

$$\textsf{case Inr}\ V\ \textsf{of Inl}\ x \Rightarrow M; \textsf{Inr}\ x' \Rightarrow M' \to_D M'[x' := V]\ [\textsc{CaseR}]$$

$$\textsf{fst Pair}\ V\ V' \to_D V\ [\textsc{Proj1}] \qquad \textsf{snd Pair}\ V\ V' \to_D V'\ [\textsc{Proj2}] \qquad f \to_D D(f)\ [\textsc{Def}]$$

$$\textsf{com}_{S,R}\ V \to_D V[S := R]\ [\textsc{Com}] \qquad \textsf{select}_{S,R}\ \ell\ M \to_D M\ [\textsc{Sel}]$$

Figure 7 Semantics of Chorλ

- Assume $M = \lambda x : T.N\ V$ and $M' = N[x := V]$. Then for any role $R \in \text{roles}(\text{type}(\lambda x : T.N))$, we have $[\![M]\!]_R = (\lambda x : [\![T]\!]_R.[\![N]\!]_R)\ [\![V]\!]_R$ and $[\![M']\!]_R = [\![N]\!]_R[x := [\![V]\!]_R]$, and for any $R' \notin \text{roles}(\text{type}(\lambda x : T.N))$, we have $[\![M]\!]_{R'} = (\lambda x' : ().[\![V]\!]_{R'})\ ()$ and $[\![M']\!]_{R'} = [\![N]\!]_{R'}[x := [\![V]\!]_{R'}]$ Since $R' \notin \text{roles}(\text{type}(\lambda x : T.N))$, $]\!]_{R'} = [\![N]\!]_{R'}[x := [\![V]\!]_{R'}] =]\!]_{R'} = [\![N]\!]_{R'}$, and $[\![V]\!]_{R'} = ()$, and $[\![N]\!]_{R'} \xrightarrow{\tau}{}^{*}_{[\![D]\!](R')} ()$ by Lemma 22. We therefore get $R[[\![M]\!]_R] \xrightarrow{\tau}{}^{*}_{[\![D]\!](R)} [\![M']\!]_R$ for all $R \in \text{roles}(\text{type}(\lambda x : T.N)$ and define $\mathcal{N} = \prod_{R \in \text{roles}(\text{type}(\lambda x:T.N))} R[[\![M']\!]_R] \mid \prod_{R' \notin \text{roles}(\text{type}((\lambda x:T.N))} R'[()]$ and the result follows.
- Assume $M = N\ M''$, $M' = N'\ M''$, and $N \to_D N'$. Then for any role $R \in \text{roles}(\text{type}(N))$, $[\![M]\!]_R = [\![N]\!]_R\ [\![M'']\!]_R$ and $[\![M']\!]_R = [\![N']\!]_R\ [\![M'']\!]_R$. For any role $R' \notin \text{roles}(\text{type}(N))$, we have $[\![M]\!]_{R'} = (\lambda x : ().[\![M'']\!]_{R'})\ [\![N]\!]_{R'}$ and $[\![M']\!]_{R'} = (\lambda x : ().[\![M'']\!]_{R'})\ [\![N']\!]_{R'}$. And by induction $[\![N]\!] \to^*_{[\![D]\!]} \mathcal{N}_N$ and $[\![N']\!] \to^*_{[\![D]\!]} \mathcal{N}'_N$ for $\mathcal{N}_N \sqsupseteq \mathcal{N}'_N$. For any role R we therefore get $[\![N]\!]_R \xrightarrow{\mu_0}_{[\![D]\!](R)} \xrightarrow{\mu_1}_{[\![D]\!](R)} \dots B_R$ and $[\![N']\!]_R \xrightarrow{\mu'_0}_{[\![D]\!](R)} \xrightarrow{\mu'_1}_{[\![D]\!](R)} \dots B'_R$ for $B_R \sqsupseteq B'_R$ for some sequences of transitions $\xrightarrow{\mu'_0}_{[\![D]\!](R)} \xrightarrow{\mu'_1}_{[\![D]\!](R)} \dots$ and $[\![N']\!]_R \xrightarrow{\mu'_0}_{[\![D]\!](R)} \xrightarrow{\mu'_1}_{[\![D]\!](R)} \dots B'_R$, and from the network semantics we get

  $$[\![M]\!] \to^* \prod_{R \in \text{roles}(\text{type}(N))} R[B_R\ [\![M'']\!]_R] \mid \prod_{R' \notin \text{roles}(\text{type}(N))} R'[\lambda x : ().[\![M'']\!]_{R'}\ B_{R'}] = \mathcal{N}$$

  and

  $$[\![M']\!] \to^* \prod_{R \in \text{roles}(\text{type}(N))} R[B'_R\ [\![M'']\!]_R] \mid \prod_{R' \notin \text{roles}(\text{type}(N))} R'[\lambda x : ().[\![M'']\!]_{R'}\ B'_{R'}] = \mathcal{N}'$$

  And since $[\![N]\!] \to^*_{[\![D]\!]} \mathcal{N}'$ and $[\![N']\!] \to^*_{[\![D]\!]} \mathcal{N}'_N$, we know these sequences of transitions can synchronise when necessary.
- Assume $M = V\ N$, $M' = V\ N'$, and $N \to N'$. Then for any role $R \in \text{roles}(\text{type}(V))$, $[\![M]\!]_R = [\![V]\!]_R\ [\![N]\!]_R$ and $[\![M']\!]_R = [\![V]\!]_R\ [\![N']\!]_R$. For any role $R' \notin \text{roles}(\text{type}(V))$, we have $[\![M]\!]_{R'} = (\lambda x : ().[\![N]\!]_{R'})\ [\![V]\!]_{R'}$ and $[\![M']\!]_{R'} = (\lambda x : ().[\![N']\!]_{R'})\ [\![V]\!]_{R'}$. By rule NAbsApp and Lemma 21, $(\lambda x : ().[\![N]\!]_{R'})\ [\![V]\!]_{R'} \xrightarrow{\tau}_{[\![D]\!](R)} [\![N]\!]_{R'}$ and $(\lambda x : ().[\![N]\!]_{R'})\ [\![V]\!]_{R'} \xrightarrow{\tau}_{[\![D]\!](R)} [\![N]\!]_{R'}$. And by induction $[\![N]\!] \to^*_{[\![D]\!]} \mathcal{N}_N$ and $[\![N']\!] \to^*_{[\![D]\!]} \mathcal{N}'_N$ for $\mathcal{N}_N \sqsupseteq \mathcal{N}'_N$. For any role R we therefore get $[\![N]\!]_R \xrightarrow{\mu_0}_{[\![D]\!](R)} \xrightarrow{\mu_1}_{[\![D]\!](R)} \dots B_R$ and $[\![N']\!]_R \xrightarrow{\mu'_0}_{[\![D]\!](R)} \xrightarrow{\mu'_1}_{[\![D]\!](R)}$

$$\textsf{send}_R\ L \xrightarrow{\textsf{send}_R\ v}_d () \ [\textsc{NSend}] \qquad \textsf{recv}_R\ () \xrightarrow{\textsf{recv}_R\ v}_d L \ [\textsc{NRecv}]$$

$$\frac{B \xrightarrow{\textsf{send}_R\ L}_{\mathbb{D}(S)} B_1' \quad B_2 \xrightarrow{\textsf{recv}_S\ L[S:=R]}_{\mathbb{D}(R)} B_2'}{S[B_1] \mid R[B_2] \xrightarrow{\tau_{S,R}}_{\mathbb{D}} S[B_1'] \mid R[B_2']} \ [\textsc{NCom}]$$

$$\oplus_R\ \ell\ B \xrightarrow{\oplus_R\ \ell}_d B \ [\textsc{NCho}] \qquad \&_R\{\ell_1 : B_1, \ldots, \ell_n : B_n\} \xrightarrow{\&_R \ell_i}_d B_i \ [\textsc{NOff}]$$

$$\frac{B_1 \xrightarrow{\oplus_R\ \ell}_{\mathbb{D}(S)} B_1' \quad B_2 \xrightarrow{\&_S\ \ell}_{\mathbb{D}(R)} B_2'}{S[B_1] \mid R[B_2] \xrightarrow{\tau_{S,R}}_{\mathbb{D}} S[B_1'] \mid R[B_2']} \ [\textsc{NSel}]$$

$$(\lambda x : T.B)\ L \xrightarrow{\tau}_d B[x := L] \ [\textsc{NAbsApp}]$$

$$\frac{B \to_d B''}{B\ B' \xrightarrow{\tau}_d B''\ B'} \ [\textsc{NApp1}] \qquad \frac{B \to_d B'}{L\ B \xrightarrow{\tau}_d L\ B'} \ [\textsc{NApp2}]$$

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

$$\frac{B \xrightarrow{\mu}_d B'''}{\textsf{case}\ B\ \textsf{of Inl}\ x \Rightarrow B'; \textsf{Inr}\ x' \Rightarrow B'' \xrightarrow{\mu}_d \textsf{case}\ B'''\ \textsf{of Inl}\ x \Rightarrow B'; \textsf{Inr}\ x' \Rightarrow B''} \ [\textsc{NCase}]$$

$$\textsf{case Inl}\ L\ \textsf{of Inl}\ x \Rightarrow B; \textsf{Inr}\ x' \Rightarrow B' \xrightarrow{\tau}_d B[x := L] \ [\textsc{NCaseL}]$$

$$\textsf{case Inr}\ L\ \textsf{of Inl}\ x \Rightarrow B; \textsf{Inr}\ x' \Rightarrow B' \xrightarrow{\tau}_d B'[x' := L] \ [\textsc{NCaseR}]$$

$$\textsf{fst Pair}\ L\ L' \xrightarrow{\tau}_d L \ [\textsc{NProj1}] \qquad \textsf{snd Pair}\ L\ L' \xrightarrow{\tau}_d L' \ [\textsc{NProj2}] \qquad f \xrightarrow{\tau}_d d(f) \ [\textsc{NDef}]$$

Figure 8 Semantics of networks.

$\ldots B_R'$ for $B_R \sqsupseteq B_R'$ for some sequences of transitions $\xrightarrow{\mu_0'}_{[\![D]\!](R)} \xrightarrow{\mu_1'}_{[\![D]\!](R)} \ldots$ and $[\![N']\!]_R \xrightarrow{\mu_0'}_{[\![D]\!](R)} \xrightarrow{\mu_1'}_{[\![D]\!](R)} \ldots B_R'$, and from the network semantics we get

$$[\![M]\!] \to^* \prod_{R \in \text{roles}(\text{type}(N))} R[[\![V]\!]_R\ B_R] \mid \prod_{R' \notin \text{roles}(\text{type}(N))} R'[B_{R'}] = \mathcal{N}$$

and

$$[\![M']\!] \to^* \prod_{R \in \text{roles}(\text{type}(N))} R[B_R'\ [\![V]\!]_R] \mid \prod_{R' \notin \text{roles}(\text{type}(N))} R'[B_{R'}'] = \mathcal{N}'$$

- Assume $M = \textsf{case}\ N\ \textsf{of Inl}\ x \Rightarrow N'; \textsf{Inr}\ x' \Rightarrow N''$, $M' = \textsf{case}\ M''\ \textsf{of Inl}\ x \Rightarrow N'; \textsf{Inr}\ x \Rightarrow N''$, and $N \to_D M''$. Then for any role R such that $R \in \text{roles}(\text{type}(N))$, $[\![M]\!]_R = \textsf{case}\ [\![N]\!]_R\ \textsf{of Inl}\ x \Rightarrow [\![N']\!]_R; \textsf{Inr}\ x' \Rightarrow [\![N'']\!]_R$ and $[\![M']\!]_R = \textsf{case}\ [\![M'']\!]_R\ \textsf{of Inl}\ x \Rightarrow [\![N']\!]_R; \textsf{Inr}\ x' \Rightarrow [\![N'']\!]_R$. For any other role $R' \notin \text{roles}(\text{type}(N))$, $[\![M]\!]_{R'} = (\lambda x : ().[\![N']\!]_{R'} \sqcup [\![N'']\!]_{R'})\ [\![N]\!]_{R'}$ and $[\![M']\!]_{R'} = (\lambda x.[\![N']\!]_{R'} \sqcup [\![N'']\!]_R)\ [\![M'']\!]_{R'}$ for $x \notin \text{fv}(N') \cup \text{fv}(N'')$. The rest follows by simple induction.
- Assume $M = \textsf{case Inl}\ V\ \textsf{of Inl}\ x \Rightarrow N; \textsf{Inr}\ x' \Rightarrow N'$ and $M' = N[x := V]$. Then for any role $R \in \text{roles}(\text{type}(\textsf{Inl}\ V)$, we have $[\![M]\!]_R = \textsf{case Inl}\ [\![V]\!]_R\ \textsf{of Inl}\ x \Rightarrow [\![N]\!]_R; \textsf{Inr}\ x' \Rightarrow [\![N']\!]_R$ and $[\![M']\!]_R = [\![N[x := [\![V]\!]_R]]\!]_R$. By Lemma 21, $[\![N[x := [\![V]\!]_R]]\!]_R = [\![N]\!]_R[x := [\![V]\!]_R]$. For any other role $R' \notin \text{roles}(\text{type}(\textsf{Inl}\ V))$, $[\![M]\!]_{R'} = (\lambda x : ().[\![N]\!]_{R'} \sqcup [\![N']\!]_{R'})\ (\lambda x' : ().()\ ())$ and $[\![M']\!]_{R'} = [\![N]\!]_{R'}[() := ()]$. We get $[\![M]\!]_R \xrightarrow{\tau}_{[\![D]\!](R)}$

$$\frac{\Sigma;\Gamma \vdash B : T}{\Sigma;\Gamma \vdash \oplus_R\ \ell\ B : T}\ [\textsc{NTChor}] \qquad \frac{\Sigma;\Gamma \vdash B_i : T \text{ for } 1 \le i \le n}{\Sigma;\Gamma \vdash \&_R\{\ell_1 : B_1, \dots \ell_n : B_n\} : T}\ [\textsc{NTOff}]$$

$$\Sigma;\Gamma \vdash \mathsf{send}_R : T \to ()\ [\textsc{NTSend}] \qquad \Sigma;\Gamma \vdash \mathsf{recv}_R : () \to T\ [\textsc{NTRecv}]$$

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

$$\frac{\Sigma;\Gamma \vdash B : T \to T' \quad \Sigma;\Gamma \vdash B : T}{\Sigma;\Gamma \vdash B\ B' : T'}\ [\textsc{NTApp}]$$

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

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

$$\Sigma;\Gamma \vdash \mathsf{Pair} : T \to_\emptyset T' \to_\emptyset (T \times T')\ [\textsc{NTPair}]$$

$$\Sigma;\Gamma \vdash \mathsf{fst} : (T \times T') \to_\emptyset T\ [\textsc{NTProj1}] \qquad \Sigma;\Gamma \vdash \mathsf{snd} : (T \times T') \to_\emptyset T'\ [\textsc{NTProj2}]$$

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

$$\frac{\forall f \in \mathsf{domain}(d) \quad f : T \in \Gamma \quad \Sigma;\Gamma \vdash d(f) : T}{\Sigma;\Gamma \vdash d}\ [\textsc{NTDefs}]$$

Figure 9 Typing rules for simple processes.

$[\![M']\!]_R$ and $[\![M]\!]_{R'} \xrightarrow{\tau}_{[\![D]\!](R')} \xrightarrow{\tau}_{[\![D]\!](R')} [\![M']\!]_{R'} \sqcup [\![N]\!]_{R'}[() := ()][\![M']\!]_{R'} \sqcup [\![N]\!]_{R'}$, and since $[\![M']\!]_{R'} \sqcup [\![N]\!]_{R'} \sqsupseteq [\![M']\!]_{R'}$ the result follows.

- Assume $M = \mathbf{case\ Inr}\ V\ \mathbf{of\ Inl}\ x \Rightarrow N; \mathbf{Inr}\ x' \Rightarrow N'$ and $M' = N'[x' := V]$. This case is similar to the previous.
- Assume $M = \mathsf{com}_{S,R} V$ and $M' = V[S := R]$. Then if $S \neq R$, $[\![M]\!]_R = \mathsf{recv}_S\ ()$, $[\![M']\!]_R = [\![V[S := R]]\!]_R = [\![V]\!]_R[S := R]$ since $\text{roles}(\text{type}(V)) = S$, $[\![M]\!]_S = \mathsf{send}_R\ [\![V]\!]_S$, $[\![M']\!]_S = ()$, and for any $R' \notin \{S, R\}$, $[\![M]\!]_{R'} = ()$ and $[\![M']\!]_{R'} = ()$. We therefore get $[\![M]\!]_R \xrightarrow{\mathsf{recv}_S [\![V]\!]_S[S:=R]}_{[\![D]\!](R)} [\![M']\!]_R$, $[\![M]\!]_S \xrightarrow{\mathsf{send}_R [\![V]\!]_S}_{[\![D]\!](S)} [\![M']\!]_S$, and $[\![M]\!]_{R'} = [\![M']\!]_{R'}$. We define $\mathcal{N} = \mathcal{N}' = [\![M']\!]$ and the result follows. If $S = R$, then $[\![M]\!]_R = (\lambda x : [\![T]\!]_R.x)\ [\![V]\!]_R$ where $\text{type}(\mathsf{com}_{S,R}) = T \to_\emptyset T$ and $[\![M']\!]_R = [\![V]\!]_R$ and $\mathcal{N} = \mathcal{N}' = [\![M']\!]$ and the result follows.
- Assume $M = \mathsf{select}_{S,R}\ \ell\ N$ and $M' = N$. Then if $S \neq R$, $[\![M]\!]_S = \oplus_R\ \ell\ [\![N]\!]_S$, $[\![M]\!]_R = \&\{\ell : [\![N]\!]_R\}$, and for any $R' \notin \{S, R\}$, $[\![M]\!]_{R'} = [\![N]\!]_{R'}$. We therefore get $[\![M]\!] \xrightarrow{\tau_{R,S}}_{[\![D]\!]} [\![M]\!] \setminus \{R, S\} \mid R[[\![N]\!]_R] \mid S[[\![N]\!]_S]$ and for all $R' \notin \{S, R\}$, $[\![M]\!]_{R'} \xrightarrow{\tau}_{[\![D]\!](R')} [\![N]\!]_{R'}$ and the result follows. If $S = R$
- Assume $M = \mathsf{fst\ Pair}\ V\ V'$ and $M' = V$. Then for any role $R \in \text{roles}(\text{type}(\mathsf{Pair}\ M'\ V'))$, $[\![M]\!]_R = \mathsf{fst\ Pair}\ [\![M']\!]_R\ [\![V']\!]_R$ and for any other role $R' \notin \text{roles}(\text{type}(\mathsf{Pair}\ M'\ V')$, we have $[\![M]\!]_{R'} = (\lambda x : ().())\ ()$ and $[\![M']\!]_{R'} = ()$. We define $\mathcal{N} = \mathcal{N}' = [\![M']\!]$ and the result follows.
- Assume $M = \mathsf{snd\ Pair}\ V\ V'$ and $M' = V'$. Then the case is similar to the previous.

$$\llbracket M\ N \rrbracket_R = \begin{cases} \llbracket M \rrbracket_R\ \llbracket N \rrbracket_R & \text{if } R \in \text{roles}(\text{type}(M)) \\ (\lambda x : ().\llbracket N \rrbracket_R)\ \llbracket M \rrbracket_R & \text{for some } x \notin \text{fv}(N) \cup \text{fv}(M) \text{ otherwise} \end{cases}$$

$$\llbracket \lambda x : T.M \rrbracket_R = \begin{cases} \lambda x : \llbracket T \rrbracket_R.\llbracket M \rrbracket_R & \text{if } R \in \text{roles}(\text{type}(\lambda x : T.M)) \\ () & \text{otherwise} \end{cases}$$

$$\llbracket \textsf{case}\ M\ \textsf{of Inl}\ x \Rightarrow N; \textsf{Inr}\ x' \Rightarrow N' \rrbracket_R$$

$$\begin{cases} \textsf{case}\ \llbracket M \rrbracket_R\ \textsf{of Inl}\ x \Rightarrow \llbracket N \rrbracket_R; \textsf{Inr}\ x' \Rightarrow \llbracket N' \rrbracket_R & \text{if } R \in \text{roles}(\text{type}(M)) \\ (\lambda x'' : ().\llbracket N \rrbracket_R \sqcup \llbracket N' \rrbracket_R)\ \llbracket M \rrbracket_R & \text{for some } x'' \notin \text{fv}(N) \cup \text{fv}(N') \\ & \text{otherwise} \end{cases}$$

$$\llbracket \textsf{select}_{S,S'}\ l\ M \rrbracket_R = \begin{cases} \oplus_{S'}\ l\ \llbracket M \rrbracket_R & \text{if } R = S \neq S' \\ \&_S\{l : \llbracket M \rrbracket_R\} & \text{if } R = S' \neq S \\ \llbracket M \rrbracket_R & \text{otherwise} \end{cases}$$

$$\llbracket \textsf{com}_{S,S'} \rrbracket_R = \begin{cases} \lambda x : \llbracket T \rrbracket_R\,.x & \text{if } R = S = S' \text{ and type}(\textsf{com}_{S,S'}) = T \to T \\ \textsf{send}_{S'} & \text{if } R = S \neq S' \\ \textsf{recv}_S & \text{if } R = S' \neq S \\ () & \text{otherwise} \end{cases}$$

$$\llbracket f \rrbracket_R = f \qquad \llbracket x \rrbracket_R = \begin{cases} x & \text{if } R \in \text{roles}(\text{type}(x)) \\ () & \text{otherwise} \end{cases} \qquad \llbracket ()@S \rrbracket_R = ()$$

$$\llbracket \textsf{Pair}\ V\ V' \rrbracket_R = \begin{cases} \textsf{Pair}\ \llbracket V \rrbracket_R\ \llbracket V' \rrbracket_R & \text{if } R \in \text{roles}(\text{type}(V) \times \text{type}(V')) \\ () & \text{otherwise} \end{cases}$$

$$\llbracket \textsf{fst} \rrbracket_R = \begin{cases} \textsf{fst} & \text{if } R \in \text{roles}(\text{type}(\textsf{fst})) \\ () & \text{otherwise} \end{cases} \qquad \llbracket \textsf{snd} \rrbracket_R = \begin{cases} \textsf{snd} & \text{if } R \in \text{roles}(\text{type}(\textsf{snd})) \\ () & \text{otherwise} \end{cases}$$

$$\llbracket \textsf{Inl}\ V \rrbracket_R = \begin{cases} \textsf{Inl}\ \llbracket V \rrbracket_R & \text{if } R \in \text{roles}(\text{type}(\textsf{Inl}\ V)) \\ () & \text{otherwise} \end{cases}$$

$$\llbracket \textsf{Inr}\ V \rrbracket_R = \begin{cases} \textsf{Inr}\ \llbracket V \rrbracket_R & \text{if } r \in \text{roles}(\text{type}(\textsf{Inr}\ V)) \\ () & \text{otherwise} \end{cases}$$

Types:

$$\llbracket T \to_\rho T' \rrbracket_R = \begin{cases} \llbracket T \rrbracket_R \to \llbracket T' \rrbracket_R & \text{if } R \in \rho \cup \text{roles}(T) \cup \text{roles}(T') \\ () & \text{otherwise} \end{cases}$$

$$\llbracket T \times T' \rrbracket_R = \begin{cases} \llbracket T \rrbracket_R \times \llbracket T' \rrbracket_R & \text{if } R \in \text{roles}(T \times T') \\ () & \text{otherwise} \end{cases} \qquad \llbracket ()@S \rrbracket_R = ()$$

$$\llbracket T + T' \rrbracket_R = \begin{cases} \llbracket T \rrbracket_R + \llbracket T' \rrbracket_R & \text{if } R \in \text{roles}(T + T') \\ () & \text{otherwise} \end{cases} \qquad \llbracket t_\rho \rrbracket_R = \begin{cases} t & \text{if } R \in \rho \\ () & \text{otherwise} \end{cases}$$

$$\llbracket D \rrbracket\,(R) = \{f \mapsto \llbracket D(f) \rrbracket_R \mid f \in \mathsf{domain}(D)\}$$

Figure 10 Projecting Chorλ onto a role

- Assume $M = f$ and $M' = D(f)$. Then the result follows from the definition of $[\![D]\!]$. ◀

A.2 Proof of Theorem 15

▶ **Definition 23.** *Given a network* $\mathcal{N} = \prod_{R\in\rho} R[B_R]$*, we have* $\mathcal{N} \setminus \rho' = \prod_{R\in(\rho\setminus\rho')} R[B_R]$

▶ **Lemma 24.** *For any role* R *and network* $\mathcal{N}$*, if* $\mathcal{N} \xrightarrow{\tau_{\mathbf{R}}} \mathcal{N}'$ *and* $R \notin \mathbf{R}$ *then* $\mathcal{N}(R) = \mathcal{N}'(R)$.

Proof. Straightforward from the network semantics. ◀

▶ **Lemma 25.** *For any set of roles* $\mathbf{R}$ *and network* $\mathcal{N}$*, if* $\mathcal{N} \xrightarrow{\tau_{\mathbf{R}'}} \mathcal{N}'$ *and* $\mathbf{R} \cap \mathbf{R}' = \emptyset$ *then* $\mathcal{N} \setminus \mathbf{R} \xrightarrow{\tau_{\mathbf{R}'}} \mathcal{N}' \setminus \mathbf{R}$.

Proof. Straightforward from the network semantics. ◀

Proof. We prove this by structural induction on M.

- Assume $M = V$. Then for any role R, $[\![M]\!]_R = L$, and $[\![M]\!] \not\rightarrow$.
- Assume $M = N_1\ N_2$. Then for any role $R \in \text{roles}(\text{type}(N_1))$, $[\![M]\!]_R = [\![N_1]\!]_R\ [\![N_2]\!]_R$ and for any role $R' \notin \text{roles}(\text{type}(N_1))$, $[\![M]\!]_{R'} = (\lambda x : ().[\![N_2]\!]_{R'})\ [\![N_1]\!]_{R'}$ and we have 2 cases.
  - Assume $N_2 = V$. Then $[\![N_2]\!]_R = L$, by Lemma 21, $[\![N_2]\!]_{R'} = ()$, and we have 5 cases.
    * Assume $N_1 = \lambda x : T.N_3$. Then for any role $R \in \text{roles}(\text{type}(N_1))$, $[\![N_1]\!]_R = \lambda x : [\![T]\!]_R.[\![N_3]\!]_R$. And for any role $R' \notin \text{roles}(\text{type}(N_1))$, $[\![N_1]\!]_R = ()$. This means there exists R'' such that $\mathbf{R} = R''$ and if $R'' \in \text{roles}(\text{type}(N_1))$ then $\mathcal{N} = [\![M]\!] \setminus \{R''\} \mid R''[[\![N_3]\!]_{R''}[x := [\![N_2]\!]_{R''}]]$ and otherwise $\mathcal{N} = [\![M]\!] \setminus \{R''\} \mid R''[()]$. We say that $M' = N_3[x := N_2]$ and the result follows from using rule NAbsApp in every role in $\text{roles}(\text{type}(N_1))$ and by Lemma 22.
    * Assume $N_1 = \textsf{com}_{S,R}$. Then if $S \neq R$, $[\![M]\!]_S = \textsf{send}_R\ [\![N_2]\!]_S$, $[\![M]\!]_R = \textsf{recv}_R\ ()$, and for $R' \notin \{S, R\}$ $[\![M]\!]_{R'} = \lambda x : ().()\ ()$, and either $\mathbf{R} = S, R$ or $\mathbf{R} = R'$ for $R' \notin \{S, R\}$ and if $S = R$ then $[\![N_1]\!]_R = (\lambda x : [\![T]\!].x$ where $\text{type}(N_1) = T \to T$.
    If $\mathbf{R} = S, R$ then $\mathcal{N} = [\![M]\!] \setminus \{S, R\} \mid S[()] \mid R[[\![N_2]\!]_S[S := R]]$. Because $[\![N_2]\!]_R = ()$ and $[\![N_2]\!]_S = v$, $N_2 = V$. Therefore $M \to V[S := R]$ and the result follows.
    If $\mathbf{R} = R$ then $S = R\ \mathcal{N} = [\![M]\!] \setminus \{R\} \mid R[[\![N_2]\!]_R]$ and the rest is similar to above.
    If $\mathbf{R} = R' \notin \{S, R\}$ then $\mathcal{N} = [\![M]\!] \setminus \{R'\} \mid R'[()]$. Since $[\![N_2]\!]_{R'} = L$, we get $N_2 = V$, and M and $\mathcal{N}$ can therefore both do the communication, and the result follows similarly to the previous case.
    * Assume $N_1 = \textsf{fst}$. Then $N_2 = \textsf{Pair}\ V\ V'$ and for any role $R \in \text{roles}(\text{type}(\textsf{Pair}\ V\ V'))$, $[\![M]\!]_R = \textsf{fst}\ \textsf{Pair}\ [\![V]\!]_R\ [\![V']\!]_R$ and for any other role $R' \notin \text{roles}(\text{type}(\textsf{Pair}\ V\ V')$, by Lemma 21 we have $[\![M]\!]_{R'} = (\lambda x : ().())\ ()$.
    If $\mathbf{R} = R \in \text{roles}(\text{type}(\textsf{Pair}\ V\ V'))$ then $\mathcal{N} = [\![M]\!] \setminus \{R\} \mid R[[\![V]\!]_R]$ and $M \to_D V$. The result follows by use of rules NProj1 and NAbsApp and Lemma 21.
    If $\mathbf{R} = R' \notin \text{roles}(\text{type}(\textsf{Pair}\ V\ V'))$ then $\mathcal{N} = [\![M]\!] \setminus \{R'\} \mid R'[()]$ and $M \to_D V$. The result follows similarly to above
    * Assume $N_1 = \textsf{snd}$. This case is similar to the previous.
    * Otherwise, $N_1 \neq V$ and either $\mathbf{R} = R$ or $\mathbf{R} = R, S$.
    If $\mathbf{R} = R$ then $[\![N_1]\!]_R \xrightarrow{\tau} B$ and if $R \in \text{roles}(\text{type}(N_1))$, $\mathcal{N} = [\![M]\!] \setminus \{R\} \mid R[B\ [\![N_2]\!]_R]$ and otherwise $\mathcal{N} = [\![M]\!] \setminus \{R\} \mid R[(\lambda x : ().[\![N_2]\!]_R)\ B]$. We therefore have $[\![N_1]\!] \xrightarrow{\tau_R} [\![N_1]\!] \setminus \{R\} \mid R[B]$, and by induction, $N_1 \to N_1'$ such that $[\![N_1]\!] \setminus \{R\} \mid R[B] \to^* \mathcal{N}_1$ for $\mathcal{N}_1 \sqsupseteq [\![N_1']\!]$. Since all these transitions can be propagated past N_2 in the

network and $[\![N_2]\!]_{R'}$ or $\lambda x : ().[\![N_2]\!]_{R'}$ in any role R' involved, we get the result for $M' = N_1'\ N_2$.

If $\mathbf{R} = R, S$ then the case is similar.

- If $N_2 \neq V$ then we have 2 cases.
  - If $\mathbf{R} = R$ then either $[\![N_1]\!]_R \xrightarrow{\tau} B$ and the case is similar to the previous or $N_1 = V$ and $[\![N_2]\!]_R \xrightarrow{\tau} B$.

    In the second case, if $R \in \text{roles}(\text{type}(N_1))$ then $\mathcal{N} = [\![M]\!] \setminus \{R\} \mid R[N_1\ B]$, and by induction, $N_2 \to N_2'$ such that $[\![N_2]\!] \setminus \{R\} \mid R[B] \to^* \mathcal{N}_2$ for $\mathcal{N}_2 \sqsupseteq [\![N_2']\!]$. We say $\mathcal{N}' = \prod_{R \in \text{roles}(\text{type}(N_1))} R[N_1\ \mathcal{N}_2'(R)] \mid \prod_{R' \in \text{roles}(\text{type}(N_1))} R'[\mathcal{N}_2'(R)]$ and the result follows straightforwardly from the semantics. If $R \notin \text{roles}(\text{type}(N_1))$ the case is similar.
  - If $\mathbf{R} = S, R$ then there exists L such that either $[\![N_1]\!]_S \xrightarrow{\textsf{send}_R\ L} B_S$ or $[\![N_2]\!]_S \xrightarrow{\textsf{send}_R\ L} B_S$ and $[\![N_1]\!]_R \xrightarrow{\textsf{recv}_S\ L[S:=R]} B_R$ or $[\![N_2]\!]_R \xrightarrow{\textsf{recv}_S\ L[S:=R]} B_R$.

    If $[\![N_1]\!]_S \xrightarrow{\textsf{send}_R\ L} B_S$ then $[\![N_1]\!]_S \neq L'$ and therefore $[\![N_1]\!]_R \xrightarrow{\textsf{recv}_S\ L[S:=R]} B_R$ and the case is similar to the previous. If $[\![N_2]\!]_S \xrightarrow{\textsf{send}_R\ L} B_S$ then $[\![N_1]\!]_S = L'$, and therefore $[\![N_2]\!]_R \xrightarrow{\textsf{recv}_S\ L[S:=R]} B_R$ and the case is similar to the previous.

- Assume $M = \textsf{case}\ N\ \textsf{of}\ \textsf{Inl}\ x \Rightarrow N';\ \textsf{Inr}\ x' \Rightarrow N''$. Then for any role $R \in \text{roles}(\text{type}(N))$, $[\![M]\!]_R = \textsf{case}\ [\![N]\!]_R\ \textsf{of}\ \textsf{Inl}\ x \Rightarrow [\![N']\!]_R;\ \textsf{Inr}\ x' \Rightarrow [\![N'']\!]_R$. And for any other role $R' \notin \text{roles}(\text{type}(N))$, $[\![M]\!]_{R'} = (\lambda x.[\![N']\!]_{R'} \sqcup [\![N'']\!]_{R'})\ [\![N]\!]_{R'}$. We have three cases.
  - Assume $\mathbf{R} = R \in \text{roles}(\text{type}(N))$. Then we have three cases.
    - Assume $N = \textsf{Inl}\ V$. Then $[\![N]\!]_R = \textsf{Inl}\ [\![V]\!]_R$ and $\mathcal{N} = [\![M]\!] \setminus \{R\} \mid R[[\![N'[x := [\![V]\!]_R]]\!]_R]$. We define $M' = N'$ and since $[\![N']\!]_{R'} \sqsupseteq [\![N']\!]_{R'} \sqcup [\![N'']\!]_{R'}$ the result follows from using rules NAbsApp and NCaseL.
    - Assume $N = \textsf{Inr}\ V$. Then the case is similar to the previous.
    - Otherwise, we have a transition $[\![N]\!]_R \xrightarrow{\tau} B$ such that

$$\mathcal{N} = [\![M]\!] \setminus \{R\} \mid R[\textsf{case}\ B\ \textsf{of}\ \textsf{Inl}\ x \Rightarrow [\![N']\!]_R;\ \textsf{Inr}\ x' \Rightarrow [\![N'']\!]_R]$$

      and the result follows from induction similar to the last application case.
  - Assume $\mathbf{R} = R \notin \text{roles}(\text{type}(N))$. Then we have three cases.
    - Assume $N = \textsf{Inl}\ V$. Then $[\![N]\!]_R = ()$ and $\mathcal{N} = [\![M]\!] \setminus \{R\} \mid R[[\![N']\!]_R \sqcup [\![N'']\!]_R]$. We define $M' = N'$ and the result follows.
    - Assume $N = \textsf{Inr}\ V$. Then the case is similar to the previous.
    - Otherwise, $[\![N]\!]_R \neq L$ and we therefore have $[\![N]\!]_R \xrightarrow{\tau} B$ and $\mathcal{N} = [\![M]\!] \setminus \{R\} \mid R[(\lambda x.[\![N']\!]_R \sqcup [\![N'']\!]_R)\ B]$. We therefore have $[\![N]\!] \xrightarrow{\tau_R} [\![N]\!] \setminus \{R\} \mid R[B]$, and by induction, $N \to_D N'''$ such that $[\![N]\!] \setminus \{R\} \mid R[B] \to^* \mathcal{N}'''$ for $\mathcal{N}''' \sqsupseteq [\![N''']\!]$. Since all these transitions can be propagated past N_2 in the network and the conditional or $(\lambda x.[\![N']\!]_{R''} \sqcup [\![N'']\!]_{R''})$ in any other role R' involved, we get the result for $M' = \textsf{case}\ N'''\ \textsf{of}\ \textsf{Inl}\ x \Rightarrow N';\ \textsf{Inr}\ x' \Rightarrow N''$.
  - Assume $\mathbf{R} = S, R$. Then the logic is similar to the third subcases of the previous two cases.
- Assume $M = \textsf{select}_{S,R}\ \ell\ N$. This is similar to the $N_1 = \textsf{com}_{S,R}$ case above.
- Assume $M = f$. Then $[\![M]\!] = \Pi_R R[f]$. We therefore have some role R such that $\mathbf{R} = R$ and $\mathcal{N} = [\![M]\!] \setminus R[[\![D]\!](R)]$. We then define $M' = D(f)$ and $\mathcal{N}' = \Pi_R R[[\![D]\!](R)(f)]$ and the result follows. ◀