Functional Choreographic Programming[⋆]

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

Department of Mathematics and Computer Science, University of Southern Denmark

Abstract Choreographic programming is an emerging programming paradigm for concurrent and distributed systems, where developers write the communications that should be enacted and a compiler then automatically generates a distributed implementation.
Currently, the most advanced incarnation of the paradigm is Choral, an object-oriented choreographic programming language that targets Java. Choral deviated significantly from known theories of choreographies, and in particular introduced the possibility of expressing higher-order choreographies that are fully distributed.
In this article, we introduce Chorλ, the first functional choreographic programming language. It is also the first theory that explains the core ideas of higher-order choreographic programming. We show that bridging the gap between practice and theory requires developing a new evaluation strategy and typing discipline for λ terms that accounts for the distributed nature of computation in choreographies.

Keywords: Choreographies · Concurrency · Lambda calculus · Type Systems

1 Introduction

Choreographies are coordination plans for concurrent and distributed systems, which prescribe the communications that system participants should enact in order to interact correctly with each other. They are widely used in industry, especially for documentation [17,21,32]. Essentially, choreographies are structured compositions of communications. These are expressed using a variation of the communication term from security protocol notation, $\mathsf{Alice} \mathrel{\texttt{->}} \mathsf{Bob}\colon M$, which reads "Alice communicates the message M to Bob" [27].

Choreographic programming is an emerging programming paradigm aimed at producing correct-by-construction distributed implementations of choreographies [16,20,24]. In this paradigm, programs are choreographies in which communications are structured using standard control-flow constructs, e.g., conditionals. A compiler then *projects* a choreography onto each participant, creating an executable program, which enacts the expected message passing behaviour.

⋆ Work partially supported by Villum Fonden, grant no. 29518.

Choreographies can be large in practice—some even over a hundred pages of text [28]. Thus, it is important to study how choreographies can be made *modular*, enabling the writing (preferably disciplined by types) of large choreographies as compositions of smaller, reusable ones. The state-of-the-art on modularity in choreographic programming is currently represented by Choral, an object-oriented choreographic programming language in which choreographies are compiled to Java libraries that applications can use as protocol implementations [15]. Choral is the first choreographic programming language powerful enough to support realistic, mainstream software development. In particular, it introduced higher-order composition to choreographic programming—the ability to define and invoke choreographies parameterised over other choreographies. Higher-order composition is essential to many practical scenarios, e.g. extensible protocols. An example (covered in Section 4.2) is the Extensible Authentication Protocol (EAP), a widely-employed link-layer protocol for the authentication of peers connecting to a network [31]. EAP is parametric over a list of authentication protocols, and therefore requires higher-order composition.

In Choral, data types are equipped with (possibly many) *roles*, which are abstractions of participants. This allows for writing object methods that involve multiple roles (choreographic methods). We illustrate with an example from [15].

Example 1 (Authentication protocol in Choral [15]). Consider a distributed authentication protocol in which a client (C) wishes to use its account at an identity provider (I) to access a service (S). Such a protocol can be implemented in Choral as follows.

$\mathsf{class\ Authenticator@}(S, C, I)$
$\quad\{\ \mathsf{AuthResult@}(C, S)\ \mathsf{authenticate(Credentials@}C\ \mathit{credentials})\{...\}\ \}$

In the Choral code above, class Authenticator is distributed between the three roles S, C, and I. Method authenticate takes the credentials of C (to access its account) and returns the result of the authentication computed at I to C and S. The result $\mathsf{AuthResult@}(C, S)$ is a pair of session tokens, one located at C and the other at S (if the authentication fails, these will be empty). The interested reader can see how this example can be implemented in Chorλ in [6]. $\triangleleft$

While Choral demonstrated the usefulness of higher-order choreographies, its development was driven by practice, and it is not grounded in any existing theory. In particular, the typing and semantics of higher-order choreographies is not formally understood yet. The current contribution aims at closing this gap.

This Article. We present the choreographic λ-calculus, Chorλ for short, a theory of choreographic programming that supports higher-order, modular composition.

Chorλ is the first choreographic programming model based on λ-calculus, which has two advantages. First, we can tap on a well-known foundation for higher-order programming. Second, it reveals that the key design features of

Choral work in the context of functional programming as well. In this way, Chorλ is also the first instance of *functional* choreographic programming.

Chorλ is expressive enough to serve as a model of the core features of Choral, which we illustrate by recreating some of the key examples given as motivation in the original presentation of Choral [15] (including remote computation, secure key exchange, and single sign-on) in our functional setting. We also model a more sophisticated scenario based on the Extensible Authentication Protocol (EAP). Our examples demonstrate that Chorλ allows for parameterising choreographies over different communication semantics, enabling protocol layering, a first for theory of choreographic programming.

To capture the essence of higher-order choreographies in the λ-calculus, we extend its syntax with features from choreographies and ambient calculi (Section 2.1) [3,25]. Namely, in Chorλ, data has explicit location and can be moved between roles using communication primitives. Another innovative feature is that the term for performing a communication is a function, and can therefore be composed with other terms as usual in functional programming.

We develop a typing discipline for Chorλ where types are located at roles (Section 2.2). The key novelty of our type system is that it tracks which roles are involved in which terms; this requires extending the standard connective for typing functions and a dedicated environment in typing judgements.

Another key contribution of this paper is a semantics for choreographies (Section 3) in Chorλ. Formulating an appropriate semantics has been particularly challenging, because there is no prior evaluation strategy for the λ-calculus that is suitable for functional choreographies. Since choreographies express distributed computation, theories of choreographic languages typically support out-of-order execution for subterms that can be evaluated at independent locations [2]. How to formulate the necessary inference rules is well-known in the imperative setting, but it has never been studied in others. This notion of out-of-order execution makes it possible to project the behaviour of each participant and get a correspondence between their behaviours and that of the choreography. This development is outside the scope of the current contribution; the interested reader can find the full discussion in the accompanying technical report [6].

Structure of the paper. Chorλ, along with its typing, is presented in Section 2. Its semantics and key properties are discussed in Section 3. Examples of choreographies inspired by practice are given in Section 4. Related work is given in Section 5. Conclusions are presented in Section 6.

2 The Choreographic λ-calculus

In this section we introduce the Choreographic λ-calculus, Chorλ. This calculus extends the simply typed λ-calculus [5] with recursion, choreographic terms for communication, and roles.

Roles are independent participants in a system based on message passing. 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".

Values can be moved from a role to another using a communication primitive. For example, the term $\mathbf{com}_{\mathsf{Alice},\mathsf{Bob}}$ 5@Alice represents the communication of the value 5 from Alice to Bob. This term evaluates to 5@Bob and has type Int@Bob.

2.1 Syntax

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

$$M ::= V \mid f(\vec{R}) \mid M\ M \mid \mathbf{case}\ M\ \mathbf{of}\ \mathbf{Inl}\ x \Rightarrow M\mathbf{;}\ \mathbf{Inr}\ x \Rightarrow M \mid \mathbf{select}_{R,R}\ l\ M$$
$$V ::= x \mid \lambda x : T.M \mid \mathbf{Inl}\ V \mid \mathbf{Inr}\ V \mid \mathbf{fst} \mid \mathbf{snd} \mid \mathbf{Pair}\ V\ V \mid \mathbf{()}@R \mid \mathbf{com}_{R,R}$$
$$T ::= T \to_\rho T \mid T + T \mid T \times T \mid ()@R \mid t@\vec{R}$$

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 (or function name), R is a role, ρ is a set of roles, and t is a type variable.

Abstraction $\lambda x : T.M$, variable x and application MM are as in the standard (simply typed) λ-calculus, and pairs and sums are added in the standard way. For the sake of 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 $\lambda x : T.\mathbf{Inl}\ x$ and then apply it to any choreography. Similarly, we can define the functions **inr** and **pair** (the latter 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 $\mathbf{com}_{S,R}$ and $\mathbf{select}_{S,R}\ l\ M$ (where S and R are roles) come from choreographies and are the only primitives of Chorλ that introduce interaction between roles. The term $\mathbf{com}_{S,R}$ is a *communication*: it acts as a function that takes a value at role S and returns the same value at role R. The standard choreographic primitive for synchronous communication Alice -> Bob: M is recovered as the function application $\mathbf{com}_{\mathsf{Alice},\mathsf{Bob}}\ M$. The term $\mathbf{select}_{S,R}\ l\ M$ is a *selection*, where S informs R that it has selected the label l before continuing as M. Selections are needed for realisability: with this interaction, S communicates its internal choice to R so that both agree on their future behaviour. Labels are constants chosen from a fixed set (e.g., $\{\mathsf{left}, \mathsf{right}, \mathsf{start}, \mathsf{stop}, \dots\}$).

Finally, $f(\vec{R})$ stands for a (choreographic) function f instantiated with the roles $\vec{R}$, which evaluates to the body of the function as given by an environment of definitions (a mapping from function names to choreographies). Function names are used to model recursion. In the typing and semantics of Chorλ, we use D to range over mappings of function names to choreographies. Within a choreography, there is no need to distinguish between roles that are statically fixed and role parameters: inside of a function definition, roles are parameters of the function; otherwise, roles are statically determined. All roles are treated in the same way by our theory.

To illustrate base values, we also have a term $()@R$ which denotes a unit value at the role R—other base values, like $5@R$ used in the examples above, can be easily included following the same approach. Values are not limited to one role in general; for example, $\textsf{\textbf{Pair}}\ ()@S\ ()@R$ denotes a distributed pair where the first element resides at S and the second at R. We say a choreography (or value or type) is local to R if R is the only role mentioned in any subterm of the choreography, e.g., $\lambda x : ()@R.(\textsf{\textbf{Pair}}\ x\ ()@R)$ is a local function located at R.

Types in Chorλ 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 a function may involve more roles besides those listed in the types of their input and output, the type of abstractions $T \to_\rho T'$ 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'. We often omit this annotation if the set of additional roles is empty, writing $T \to T'$ instead of $T \to_\emptyset T'$. For example, if Alice wants to communicate an integer to Bob directly (without intermediaries), then she should use a choreography of type $\mathsf{Int}@\mathsf{Alice} \to \mathsf{Int}@\mathsf{Bob}$; however, if the communication might go through a proxy, then she can use a choreography of type $\mathsf{Int}@\mathsf{Alice} \to_{\{\mathsf{Proxy}\}} \mathsf{Int}@\mathsf{Bob}$. This annotation is vital to the theory of projection, which is not presented in this paper.[1]

Aside from the annotations on arrows, our types resemble those of simply typed λ-calculus and serve the same primary purpose of keeping track of input and output of functions in order to prevent nonsensical choreographies. Consider the function $h = \lambda x : \mathsf{Int}@\mathsf{Alice}.\textsf{\textbf{com}}_{\mathsf{Proxy},\mathsf{Bob}}\ (\textsf{\textbf{com}}_{\mathsf{Alice},\mathsf{Proxy}}\ x)$, which communicates an integer from Alice to Bob by passing through an intermediary Proxy and has the type $\mathsf{Int}@\mathsf{Alice} \to_{\{\mathsf{Proxy}\}} \mathsf{Int}@\mathsf{Bob}$. For any term M, the composition $h\ M$ makes sense if the evaluation of M returns something of the type expected by h, that is $\mathsf{Int}@\mathsf{Alice}$. The composition $h\ 5@\mathsf{Alice}$ makes sense, but $h\ 5@\mathsf{Bob}$ does not, because the argument is not at the role expected by h.

Types for sums and products are the usual ones. 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. Recursive type variable $t@\vec{R}$ are annotated with the roles $\vec{R}$, instantiating the roles occurring in their definition (we discuss type definitions in Section 2.2).

Definition 2 (Roles of a type). *The roles of a type T,* roles(T)*, are defined as follows.*

$$\mathrm{roles}(t@\vec{R}) = \vec{R} \qquad\qquad \mathrm{roles}(T \to_\rho T') = \mathrm{roles}(T) \cup \mathrm{roles}(T') \cup \rho$$
$$\mathrm{roles}(()@R) = \{R\} \quad \mathrm{roles}(T + T') = \mathrm{roles}(T \times T') = \mathrm{roles}(T) \cup \mathrm{roles}(T')$$

In our examples we also assume the usual datatypes for integers (Int) and strings (String) together with their usual operations.

[1] The interested reader can find it in Figure 6, in the appendix.

Example 2 (Remote Function). We can use Chorλ to define a small choreography, $\mathsf{remFun}(C, S)$ for a distributed computation in which a client, C sends an integer *val* to a server S where a function *fun* located at S is applied to *val* before the result gets returned to C.

$$\mathsf{remFun}(C, S) = \lambda f : \mathsf{Int}@S \to \mathsf{Int}@S.\ \lambda v : \mathsf{Int}@C.\ \mathbf{com}_{S,C}\ (f\ (\mathbf{com}_{C,S}\ v))$$

This choreography is parametrised on the roles S and C as well as the local function *fun* and value *val*. ◁

Crucially, a choreographic term M may involve more roles than those listed in its type. For instance, the three choreographies $(()@R)$, $(\mathbf{com}_{S,R}\ ()@S)$, and $(\mathbf{com}_{P,R}\ (\mathbf{com}_{S,P}\ ()@S))$ all have type $()@R$, but they implement different behaviours involving different roles. This yields a substitution principle for choreographies that makes them compositional, and will be important in establishing type preservation later.

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 [4]. We give an example of how selections should be used.

Example 3 (Remote Map). We now build on the remote function from Example 2 by using it to create a choreography $\mathsf{remMap}(C, S)$, where the server S applies a local function to not just one value received from the client C, but instead to each element of a list sent individually from C to S and then returned after the computation at S is complete.

```
$\mathsf{remMap}(C, S) = \lambda f : \mathsf{Int}@S \to \mathsf{Int}@S.\ \lambda list : [\mathsf{Int}]@C.$
  $\mathbf{case}\ list\ \mathbf{of}$
  $\mathbf{Inl}\ x \Rightarrow \mathbf{select}_{C,S}\ \mathsf{stop}\ ()@C;$
  $\mathbf{Inr}\ x \Rightarrow \mathbf{select}_{C,S}\ \mathsf{go}\ \mathsf{cons}(C)\ (\mathsf{remFun}(C, S)\ f\ (\mathbf{fst}\ x))\ (\mathsf{remMap}(C, S)\ f\ (\mathbf{snd}\ x))$
```

Here, $[\mathsf{Int}]@C$ is the recursive type satisfying $[\mathsf{Int}]@C = ()@C+(\mathsf{Int}@C\times[\mathsf{Int}]@C)$, representing a list of integers and $\mathsf{cons}(C)$ is the usual list constructor located at C. In general, we write $[t]@(R_1, \ldots, R_n)$ to mean the recursive type satisfying

$$[t]@(R_1, \ldots, R_n) = (()@R_1 \times \cdots \times ()@R_n)+(t@(R_1, \ldots, R_n)\times[t]@(R_1, \ldots, R_n)).$$

When we introduce typing judgements later, we will show how to work with this kind of type equations.

The choreography uses selections so that S is informed about how it should behave (terminate or recur) depending on a local choice at C. This is essential if the choreography is to be implemented in a fully distributed way, since the information is initially available only at C. Notice how the **case** is evaluated on data at role C, so that role is the only one initially knowing which branch has been chosen. Each branch, however, starts with role C sending a label to role S. On the other hand, S must wait to receive a label from C to figure out whether it should terminate (label stop) or continue (label again): from its point of view, S is reactively handling a stream. ◁

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

$$\frac{f(\vec{R}') : T \in \Gamma \quad \vec{R} \subseteq \Theta \quad ||\vec{R}|| = ||\vec{R}'|| \quad \mathrm{distinct}(\vec{R})}{\Theta; \Sigma; \Gamma \vdash f(\vec{R}) : T[\vec{R}' := \vec{R}]}\ [\textsc{TDef}]$$

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

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

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

Figure 1. Typing rules for Chorλ (representative selection).

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 $\mathrm{fv}(M)$ for the set of free variables in term M. The formal definition can be found in [6].We call a choreography closed if it has no free variables, and restrict our results to closed choreographies.

2.2 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 is the set of roles used for typing M; Σ is a set of type definitions parameterised on roles, i.e., expressions of the form $t@\vec{R} = T$ where the elements of $\vec{R}$ are distinct; and Γ is a typing environment, i.e. a list of assignments of variable names to their type $(x : T)$ and of choreography names to the their set of bound roles and type $(f(\vec{R}) : T)$. We require that a type variable t is defined at most once in Σ, that definitions are contractive [29], and that $\mathrm{roles}(T) = \vec{R}$ for any $t@\vec{R} = T \in \Sigma$. We can use Σ to define common types such as $\mathsf{Bool}@R = ()@R + ()@R$ and the lists described in Example 3. We call $\Theta; \Sigma; \Gamma$ 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 typing rules are given in [6]).We use the predicate $\mathrm{distinct}(\vec{R})$ to indicate that the elements of $\vec{R}$ are distinct and $||\vec{R}||$ to denote the number of elements of $\vec{R}$.

One novel part of our type system is the annotation ρ on the function type $T \to_\rho T'$, which, while not necessary for the results of this paper, ensures that the type of any value contains all the roles of that value. Rule TAbs uses Θ to ensure that ρ contains every additional role used in the function by requiring every role to be in Θ and restricting Θ to the roles of T, ρ, and T'.

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 (here $T[S := R]$ is the type expression obtained by replacing S with R). Note that the type of the value being communicated must be located entirely at S. Rule TSel types selections as no-ops, only 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.

$$\frac{\forall f(\vec{R}) \in \mathsf{domain}(D): \quad f(\vec{R}) : T \in \Gamma \quad \vec{R}; \Sigma; \Gamma \vdash D(f(\vec{R})) : T \quad \text{distinct}(\vec{R})}{\Theta; \Sigma; \Gamma \vdash D}$$

Example 4. The set of definitions in Examples 2 and 3 can be typed in the typing context:

$$\Theta = \{C, S\} \qquad \Sigma = \{[\mathsf{Int}]@R = ()@R + (\mathsf{Int}@R \times [\mathsf{Int}]@R)\}$$
$$\Gamma = \begin{Bmatrix} \mathsf{remFun}(C, S) : (\mathsf{Int}@S \to \mathsf{Int}@S) \to \mathsf{Int}@C \to \mathsf{Int}@C, \\ \mathsf{remMap}(C, S) : (\mathsf{Int}@S \to \mathsf{Int}@S) \to [\mathsf{Int}]@C \to [\mathsf{Int}]@C \end{Bmatrix}$$

$\triangleleft$

3 Semantics of Chor$\boldsymbol{\lambda}$

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.

The key idea of our semantics is that terms at different roles can be evaluated independently, unless interaction is specified within the choreography. This kind of role-based out-of-order execution is typical for choreographic calculi [2], but we port it to λ-calculus here for the first time. In addition to functional choreographies having a different structure to imperative, out-of-order execution in higher-order choreographies is complicated by having actions where multiple roles are involved but no synchronisation happens, namely applications of values located at multiple roles such as choreographies and pairs with elements located at different roles.

The semantics are annotated with a label, ℓ, and a set of synchronising roles, $\mathbf{R}$. The label is either λ, when an action is propagated out through a λ-term as in rule InAbs, or τ otherwise. The set of synchronising roles is empty if no synchronisations are taking place. The purpose of the label and synchronising roles is to ensure that synchronisations between the same roles occur in the expected order, the importance of which will become clear later.

$$\frac{}{\lambda x : T.M\ V \xrightarrow{\tau,\emptyset}_D M[x := V]}\ [\textsc{AppAbs}] \qquad \frac{M \xrightarrow{\ell,\mathbf{R}}_D M'}{\lambda x : T.M \xrightarrow{\lambda,\mathbf{R}}_D \lambda x : T.M'}\ [\textsc{InAbs}]$$

$$\frac{M \xrightarrow{\ell,\mathbf{R}}_D M' \quad \ell = \lambda \Rightarrow \mathbf{R} \cap \mathrm{roles}(N) = \emptyset}{M\ N \xrightarrow{\tau,\mathbf{R}}_D M'\ N}\ [\textsc{App1}]$$

$$\frac{N \xrightarrow{\tau,\mathbf{R}}_D N'}{V\ N \xrightarrow{\tau,\mathbf{R}}_D V\ N'}\ [\textsc{App2}] \qquad \frac{N \xrightarrow{\tau,\mathbf{R}}_D N' \quad \mathbf{R} \cap \mathrm{roles}(M) = \emptyset}{M\ N \xrightarrow{\tau,\mathbf{R}}_D M\ N'}\ [\textsc{App3}]$$

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

$$\frac{M_1 \xrightarrow{\ell,\mathbf{R}}_D M_1' \quad M_2 \xrightarrow{\ell,\mathbf{R}}_D M_2' \quad \mathbf{R} \cap \mathrm{roles}(N) = \emptyset}{\textsf{case}\ N\ \textsf{of Inl}\ x \Rightarrow M_1;\ \textsf{Inr}\ x' \Rightarrow M_2 \xrightarrow{\ell,\mathbf{R}}_D \textsf{case}\ N\ \textsf{of Inl}\ x \Rightarrow M_1';\ \textsf{Inr}\ x' \Rightarrow M_2'}\ [\textsc{InCase}]$$

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

$$\frac{}{\textsf{fst Pair}\ V\ V' \xrightarrow{\tau,\emptyset}_D V}\ [\textsc{Proj1}] \qquad \frac{D(f(\vec{R'})) = M}{f(\vec{R}) \xrightarrow{\tau,\emptyset}_D M[\vec{R'} := \vec{R}]}\ [\textsc{Def}]$$

$$\frac{\mathrm{fv}(V) = \emptyset}{\textsf{com}_{S,R}\ V \xrightarrow{\tau,\{S,R\}}_D V[S := R]}\ [\textsc{Com}] \qquad \frac{}{\textsf{select}_{S,R}\ l\ M \xrightarrow{\tau,\{S,R\}}_D M}\ [\textsc{Sel}]$$

$$\frac{M \xrightarrow{\ell,\mathbf{R}}_D M' \quad \mathbf{R} \cap \{S,R\} = \emptyset}{\textsf{select}_{S,R}\ \ell\ M \xrightarrow{\ell,\mathbf{R}}_D \textsf{select}_{S,R}\ \ell\ M'}\ [\textsc{InSel}] \qquad \frac{M \rightsquigarrow^* N \quad N \xrightarrow{\tau,\mathbf{R}} M'}{M \xrightarrow{\tau,\mathbf{R}}_D M'}\ [\textsc{Str}]$$

Figure 2. Semantics of Chorλ.

Rules AppAbs, App1 and App2 implement a call-by-value λ-calculus. Rules Case and CaseL and its counterpart rule 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.

In addition to the fairly standard λ-calculus semantics, we have some rules for out-of-order execution. These include rewriting terms as described in Figure 3 and being able to propagate some transitions past an abstraction, case, and selection as in rules InAbs, InCase and InSel. We also have a "role-aware" variation of full β-reduction by using rule App3, the need for which is illustrated by Example 5. These rules serve the purpose of making our semantics decentralised, in the sense that actions at distinct roles can proceed independently.

Example 5. Consider the choreography $M = f(S)\ ((\lambda x : T@R.V@S)\ V'@R)$. (Note that we abuse notation when we write $V@S$ and $T@R$ to denote that V and T are located entirely at roles S and R, respectively, though this is not

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

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

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

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

$$\frac{}{(\textsf{select}_{S,R}\ l\ N)\ M \rightsquigarrow \textsf{select}_{S,R}\ l\ (N\ M)}\ [\text{R-SelR}]$$

$$\frac{\mathrm{sroles}(M) \cap \mathrm{roles}(N) = \emptyset}{M\ (\textsf{select}_{S,R}\ l\ N) \rightsquigarrow \textsf{select}_{S,R}\ l\ (M\ N)}\ [\text{R-SelL}]$$

Figure 3. Rewriting of Chorλ.

part of the syntax of Chorλ.) The choreography includes two independent roles, R and S, but the two never actually interact: the inner application involves an abstraction and an argument located only at R, so it should be evaluated independently of S. Likewise, $f(S)$ is located entirely at role S, so it should be evaluated independently of R.

Without rule App3, M would be unable to evaluate the inner application before $f(S)$ finished running, which may be never if f diverges, breaking the assumption that roles execute in a decentralised way. $\triangleleft$

The rewriting rules are not standard for the λ-calculus, but they are not as strange as they first appear. Take, for example, rule R-AbsR; it simply states that if you have a function with two variables, $\lambda x : T.\lambda y : T'.M$, then x and y can be instantiated in any order as long as they each get the correct value. On the other hand, rule R-AbsL says that more of the computation can be pushed into an abstraction so long as it does not affect the order of synchronisations. The other rewriting rules work on similar principles, but dealing with conditionals and selections. These rules all work to ensure that while actions can be performed in different orders the result of the computation must remain the same before and after rewriting. In Example 6 we see why we need the rewriting rules in order to support the out-of-order executions necessary for the choreography to allow concurrent execution of computations located at different roles.

Example 6 (Rewriting). Consider the choreography with an abstraction at S inside an abstraction at R, $M = ((\lambda x : ()@R.\lambda x' : T@S.()@S)\ f(R))\ V@S$. As in Example 5, R and S each independently execute their part of the choreography. R evaluates $f(R)$ and then applies the result. Independently, S, executes the other application $\lambda x' : T@S.()@S\ V@S$. For M to be able to execute the application at S independently of R's actions, we need rule R-AbsR to get $\lambda x : T@S.()@S$ and $V@S$ next to each other by rewriting to $((\lambda x : ()@R.(\lambda x : T@S.()@S)\ V@S)\ f(R))$ and rule InAbs to propagate the application of $(\lambda x : T@S.()@S)$ and $V@S$ past $\lambda x : ()@R$. $\triangleleft$

Some of the out-of-order-execution rules, specifically the ones pushing the left part of an application further in, have restrictions on them because we want to avoid there being more than one communication or synchronisation available at the same time on the same roles. This is because we need to ensure that communications and selections on a specific set of roles must always happen in the same order, as we otherwise get the problems illustrated by Example 7.

Example 7 (Communication order). Consider a choreography with two $\mathbf{com}_{S,R}$ primitives, $M = (\lambda x : T@R.(\mathbf{com}_{S,R}\ V@S))\ (\mathbf{com}_{S,R}\ V'@S)$. This has a similar structure to $((\lambda x : ()@R.(\lambda x' : T@S.M)\ V@S)\ f(R))$ from Example 6, with part of the computation hidden behind an abstraction. However, while Example 6 needed to use rule InAbs to allow the computation inside of the abstraction to execute before the computation outside, doing so would cause problems in $(\lambda x : T@R.(\mathbf{com}_{S,R}\ V@S))\ (\mathbf{com}_{S,R}\ V'@S)$.

Without going into the technical details, intuitively, the behaviour of M at R should be $(\lambda x : T.(\mathbf{recv}_S\ \bot))\ (\mathbf{recv}_S\ \bot)$ and the behaviour at S should be $(\lambda x : \bot.(\mathbf{send}_R\ V))\ (\mathbf{send}_R\ V')$ where $\mathbf{send}_R$ and $\mathbf{recv}_S$ are the obvious local actions implementing $\mathbf{com}_{S,R}$. In these behaviours, term $\bot$ denotes a part of the computation that takes place elsewhere.

It is common in choreographic programming, including in Choral, for the implementation of choreographies to assume that each pair of roles has one channel between them, which they use for all communications. In such a model, if the two communications can be performed in any order then S is currently able to send either V or V' and R is correspondingly able to receive either inside or outside the abstraction. Since S and R act independently, we have no guarantee that if S chooses to send V first R will also choose to use its left receive action or vice versa. This can create situations where S sending V synchronises with the right receive at R, creating a state not intended by the choreography. $\triangleleft$

We therefore restrict the out-of-order communications by restricting the synchronising names in rules InAbs, App1, App3, InCase and InSel. To show that these rules restrict as intended, we have Proposition 1 stating that any reductions available at the same time must have different (or no) synchronisation roles.

Proposition 1. *Given a choreography M, if $M \xrightarrow{\ell,\mathbf{R}} M'$ and $M \xrightarrow{\ell',\mathbf{R}'} M''$ and there does not exist N such that $M' \rightsquigarrow^* N$, and $M'' \rightsquigarrow^* N$, then $\mathbf{R} \cap \mathbf{R}' = \emptyset$.*

Proof. The key here is that unless these transitions are either communications or selections, $\mathbf{R}$ and $\mathbf{R}'$ are empty. Once this is clear, the rest follows by induction on M. $\square$

We use the label λ in rule InAbs to restrict out-of-order communications, since we do not know which roles we need to restrict communication on in situations such as Example 7 until we reach the application, at which point the λ label becomes a τ again if it is allowed to propagate.

The restrictions on out-of-order communication in out-of-order execution rules force us to add similar restrictions in the rewriting rules, as illustrated by Example 8. For this purpose we use the concept of synchronisation roles.

Definition 3. *We define the set of synchronising roles of a choreography M, $\mathrm{sroles}(M)$, by recursion on the structure of M:*
$\mathrm{sroles}(\mathbf{com}_{S,R}) = \{S, R\}$, $\mathrm{sroles}(\mathbf{select}_{S,R}\ l\ M) = \{S, R\} \cup \mathrm{sroles}(M)$,
$\mathrm{sroles}(f(\vec{R})) = \vec{R}$*, and homomorphically on all other cases.*

Example 8. Consider a choreography with two communications between S and R, $(\mathbf{com}_{S,R}\ V@S)\ ((\lambda x : T@R.M)\ (\mathbf{com}_{S,R}\ V'@S))$. Here, thanks to rule App3 restricting on synchronisation roles, only the left $\mathbf{com}_{S,R}$ on V is available. If rule R-AbsL had no restriction on synchronisation roles, we could rewrite the choreography to $(\lambda x : T@R.((\mathbf{com}_{S,R}\ V@S)\ M))\ (\mathbf{com}_{S,R}\ V'@S)$. This instead leaves the rightmost $\mathbf{com}_{S,R}$ on V' available. This means we have both communication available depending on whether we decide to rewrite and we have the same problem as in Example 7 of S potentially choosing to send V while R has rewritten and is expecting to receive the left $\mathbf{com}_{S,R}$. We therefore do not allow such a rewrite and use synchronisation roles to prevent it. $\triangleleft$

We now show that closed choreographies remain closed under reductions.

Proposition 2. *Let M be a closed choreography. If $M \to_D M'$ then M' is closed.*

Proof. Straightforward from the semantics. $\square$

A hallmark property of choreographies is that well-typed choreographies should continue to reduce until they reach a value. We split this result into two independent statements.

Theorem 1 (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 $M \not\to_D$) or there exists a choreography M' such that $M \xrightarrow{\tau,\mathbf{R}}_D M'$.*

Proof. Follows by induction on the typing derivation of $\Theta; \Sigma; \Gamma \vdash M : T$. See details in [6]. $\square$

Theorem 2 (Type Preservation). *Let M be a 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 $\Theta; \Sigma; \Gamma \vdash M' : T$ for any M' such that $M \xrightarrow{\ell,\mathbf{R}}_D M'$.*

Proof. Follows from induction on the derivation of $M \xrightarrow{\ell,\mathbf{R}}_D M'$. See details in [6]. $\square$

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 Illustrative Examples

In this section, we illustrate the expressivity of Chorλ with some representative examples. Specifically, we use Chorλ to implement the Diffie-Hellman protocol for key exchange [14] and the Extensible Authentication Protocol [31]. The first is used in [15] to illustrate the expressiveness of Choral, and we show how it can be adapted to Chorλ's functional paradigm. The second example requires using higher-order composition of choreographies, as the choreography is parametrised on a list of authentication protocols. Chorλ is the first theory capable of modelling these choreographies as they are parametric on roles and include functions which are parametric on other choreographies and no previous formalism includes both these features.

4.1 Secure Communication

We write a choreography for the Diffie–Hellman key exchange protocol [14], which allows two roles to agree on a shared secret key without assuming secrecy of communications. As in Example 3, we use the primitive type Int.

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

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

which computes powers with a given modulo. Like all local functions in Chorλ, modPow(R) is modelled by a choreography located entirely at one role. Given modPow(R), we can implement Diffie–Hellman as the following choreography:

$$\begin{array}{l}
\mathsf{diffieHellman}(P, Q) = \\
\quad \lambda psk : \mathsf{Int}@P.\ \lambda qsk : \mathsf{Int}@Q.\ \lambda psg : \mathsf{Int}@P. \\
\quad \lambda qsg : \mathsf{Int}@Q.\ \lambda psp : \mathsf{Int}@P.\ \lambda qsp : \mathsf{Int}@Q. \\
\qquad \textbf{pair}\ (\mathsf{modPow}(P)\ psg\ (\textbf{com}_{Q,P}\ (\mathsf{modPow}(Q)\ qsg\ qsk\ qsp))\ psp) \\
\qquad\qquad (\mathsf{modPow}(Q)\ qsg\ (\textbf{com}_{P,Q}\ (\mathsf{modPow}(P)\ psg\ psk\ psp))\ qsp)
\end{array}$$

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

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

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

Using the key exchange protocol, we can now build a reusable utility that allows us to achieve secure bidirectional communication between the parties, by encrypting and decrypting messages with the shared key at the appropriate endpoints. For this we assume two functions that allow us to encrypt and decrypt a String message with a given Int key:

$$\begin{aligned}\mathsf{enc}(R) &: \mathsf{Int}@R \to \mathsf{String}@R \to \mathsf{String}@R\\ \mathsf{dec}(R) &: \mathsf{Int}@R \to \mathsf{String}@R \to \mathsf{String}@R\end{aligned}$$

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

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

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

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

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

4.2 EAP

Finally, we turn to implementing the core of the Extensible Authentication Protocol (EAP) [31]. EAP is a widely-employed link-layer protocol that defines an authentication framework allowing a peer P to authenticate with a backend authentication server S, with the communication passing through an authenticator A that acts as an access point for the network.

The framework provides a core protocol parametrised over a set of authentication methods (either predefined or custom vendor-specific ones), modelled as individual choreographies with type $\mathsf{AuthMethod}@(P, A, S) = \mathsf{String}@S \to_{\{P,A\}} \mathsf{Bool}@S$. For this reason, it is desirable that the core of the protocol be written in a way that doesn't assume any particular authentication method.

The $\mathsf{eap}(P, A, S)$ choreography does exactly that by leveraging higher-order composition of choreographies:

```
$\mathsf{eap}(P, A, S) = \lambda methods : [\mathsf{AuthMethod}]@(P, A, S).$
  $\mathsf{eapAuth}(P, A, S)\ (\mathsf{eapIdentity}\ \text{"auth request"}@S)\ methods$
$\mathsf{eapAuth}(P, A, S) = \lambda id : \mathsf{String}@S.\ \lambda methods : [\mathsf{AuthMethod}]@(P, A, S).$
  $\mathbf{if}\ \mathsf{empty}(P, A, S)\ methods\ \mathbf{then}$
    $\mathsf{eapFailure}(P, A, S)\ \text{"try again later"}@S$
  $\mathbf{else}$
    $\mathbf{if}\ (\mathbf{fst}\ methods)\ id\ \mathbf{then}$
    $\textsf{\textbf{select}}_{S,P}$ ok ($\textsf{\textbf{select}}_{S,A}$ ok (eapSuccess($P, A, S$) "welcome"@$S$))
else
    $\textsf{\textbf{select}}_{S,P}$ ko ($\textsf{\textbf{select}}_{S,A}$ ko (eapAuth($P, A, S$) $id$ ($\textsf{\textbf{snd}}$ $methods$)))
```

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

$$\begin{aligned}
\mathsf{eapIdentity}(P, A, S) &: \mathsf{String}@S \rightarrow_{\{P,A\}} \mathsf{String}@S \\
\mathsf{eapSuccess}(P, A, S) &: \mathsf{String}@S \rightarrow (\mathsf{String}@P \times \mathsf{String}@A) \\
\mathsf{eapFailure}(P, A, S) &: \mathsf{String}@S \rightarrow (\mathsf{String}@P \times \mathsf{String}@A)
\end{aligned}$$

$\mathsf{eap}(P, A, S)$ fetches the client's identity using $\mathsf{eapIdentity}(P, A, S)$, a function which exchanges the EAP packets and delivers the client's identity to the server.

Once the identity is known, $\mathsf{eapAuth}(P, A, S)$ is invoked in order to try the list of authentication methods until one succeeds, or the list is exhausted and authentication fails. EAP is parametric on a choreography, or in this case a list of choreographies, *methods*. We use the notation for lists in $[\mathsf{AuthMethod}]@(P, A, S)$ as described in Example 3, while the function $\mathsf{empty}(P, A, S)$ allows us to determine whether the list of methods is empty. Note that each authentication method can be an arbitrarily-complex choreography with its own communication structures that can involve all three involved roles, and implements a particular authentication method on top of EAP.

Finally, depending on the outcome of the authentication, an appropriate EAP packet is sent with either $\mathsf{eapSuccess}(P, A, S)$ or $\mathsf{eapFailure}(P, A, S)$ to indicate the result to the client.

5 Related Work

We already discussed much of the previous and related work on choreographic languages and choreographic programming in Section 1. In this section, we discuss relevant technical aspects in related work more in detail.

Chorλ is inspired by Choral [15], the first higher-order choreographic programming language. As we discussed in Section 1, Choral comes with no formal explanation of its semantics, typing, and guarantees. We have covered these in the present article, showing that it is possible to formulate semantics and types of higher-order choreographies that satisfy the expected properties (out-of-order execution with progress).

There exist theories of choreographies that support some form of higher-order composition, which are more restrictive than Choral and Chorλ [13,18]. In particular, they fall short of capturing distribution, both in terms of independent execution and data structures. In [13], the authors present a choreographic language for writing abstract specifications of system behaviour (as in multiparty session types [19]) that supports higher-order composition. Compared to Chorλ, the design of the language hampers decentralisation: entering a choreography requires that the programmer picks a role as central coordinator, which then orchestrates the other roles with multicasts. This coordination effectively acts as a barrier, so processes cannot really perform their own local computations independently of each other when higher-order composition is involved. After [13] and Choral [15], a theory of higher-order choreographic programming was proposed in [18]. While this theory supports computation at roles, it is even more centralised than [13]: every function application in a choreography requires that all processes go through a global barrier that involves the entire system. The global barrier is modelled as a middleware in the semantics of the language, and involves even processes that do not contribute at all to the function or its arguments.

Previous theories of choreographies organised their syntax in two layers: one for local computation and one for communication [1,2,10,11,8,12,18,22]. Chorλ has a very different and novel design, whereby a unified language addresses both areas. An important consequence of our unified approach is that Chorλ can express distributed data structures (e.g., pairs with elements located at different roles), which can be manipulated by independent local computations or in coordination by performing appropriate communications. This feature is crucial for our examples in Section 4 (and several examples in the original presentation of Choral in [15]).

Another related line of work is that on multitier programming and its progenitor calculus, Lambda 5 [26]. Similarly to Chorλ, Lambda 5 and multitier languages have data types with locations [33]. 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 contrast, 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 [16], which presents algorithms for translating choreographies to multitier programs and vice versa.

6 Conclusion and Future Work

We have presented Chorλ, a new theory of choreographic programming that supports higher-order, modular choreographies. Chorλ is equipped with a type system that guarantees progress (Theorems 1 and 2). Unlike previous choreographic programming languages, Chorλ is based on the λ-calculus. It therefore inherits the simple syntax of λ terms and it is the first purely functional choreographic programming language. The semantics of Chorλ makes it the first theory of higher-order choreographies that is truly decentralised: processes can proceed independently unless the choreography specifies explicitly that they should interact.

We have demonstrated the usefulness of higher-order choreographies in Chorλ by modelling common protocols in Section 4. The examples on single sign-on with encrypted channels and EAP, in particular, are parametrised on choreographies and cannot be expressed in previous theories, either because of lack of higher-order composition or because the semantics is not satisfactory due to global synchronisations—which the original protocol specifications do not expect.

Future Work An obvious extension of Chorλ would be to add generic data types, which we did not include to keep the focus on choreographies. Since we use λ-calculus as foundation, we believe that this would be a straightforward import of known methods.

Other features that are interesting for Chorλ have been investigated in the context of first-order choreographic languages and represent future work. These include: channel-based communication [2], dynamic creation of roles [7], internal threads [1], group communication [9], availability-awareness [23], and runtime adaptation [12].

A more sophisticated extension would be to reify roles, that is, extending the syntax such that values can be roles that can be acted upon. This could, for example, enable dynamic topologies: choreographies where a process receives at runtime a role that it needs to interact with at a later time.

Another interesting line of future work would be to extend existing formalisations of choreographic languages with the features explored in this work [11,10,18,30].

References

1. Carbone, M., Honda, K., Yoshida, N.: Structured communication-centered programming for web services. ACM Trans. Program. Lang. Syst. **34**(2), 8:1–8:78 (2012). `https://doi.org/10.1145/2220365.2220367`, `https://doi.org/10.1145/2220365.2220367`
2. Carbone, M., Montesi, F.: Deadlock-freedom-by-design: multiparty asynchronous global programming. In: Giacobazzi, R., Cousot, R. (eds.) Procs. POPL. pp. 263–274. ACM (2013). `https://doi.org/10.1145/2429069.2429101`
3. Cardelli, L., Gordon, A.D.: Mobile ambients. Theor. Comput. Sci. **240**(1), 177–213 (2000). `https://doi.org/10.1016/S0304-3975(99)00231-5`, `https://doi.org/10.1016/S0304-3975(99)00231-5`
4. Castagna, G., Dezani-Ciancaglini, M., Padovani, L.: On global types and multiparty sessions. In: Formal Techniques for Distributed Systems, pp. 1–28. Springer (2011)
5. Church, A.: A set of postulates for the foundation of logic. Annals of Mathematics **33**(2), 346–366 (1932), `http://www.jstor.org/stable/1968337`
6. Cruz-Filipe, L., Graversen, E., Lugovic, L., Montesi, F., Peressotti, M.: Functional choreographic programming. CoRR **abs/2111.03701** (2021), `https://arxiv.org/abs/2111.03701`
7. Cruz-Filipe, L., Montesi, F.: Procedural choreographic programming. In: FORTE. Lecture Notes in Computer Science, vol. 10321, pp. 92–107. Springer (2017)
8. Cruz-Filipe, L., Montesi, F.: A core model for choreographic programming. Theor. Comput. Sci. **802**, 38–66 (2020). `https://doi.org/10.1016/j.tcs.2019.07.005`, `https://doi.org/10.1016/j.tcs.2019.07.005`
9. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Communications in choreographies, revisited. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018. pp. 1248–1255.

ACM (2018). https://doi.org/10.1145/3167132.3167267, https://doi.org/10.1145/3167132.3167267

10. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Certifying choreography compilation. In: Cerone, A., Ölveczky, P.C. (eds.) Theoretical Aspects of Computing - ICTAC 2021 - 18th International Colloquium, Virtual Event, Nur-Sultan, Kazakhstan, September 8-10, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12819, pp. 115–133. Springer (2021). https://doi.org/10.1007/978-3-030-85315-0_8
11. Cruz-Filipe, L., Montesi, F., Peressotti, M.: Formalising a turing-complete choreographic language in coq. In: Cohen, L., Kaliszyk, C. (eds.) 12th International Conference on Interactive Theorem Proving, ITP 2021, June 29 to July 1, 2021, Rome, Italy (Virtual Conference). LIPIcs, vol. 193, pp. 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2021). https://doi.org/10.4230/LIPIcs.ITP.2021.15
12. Dalla Preda, M., Gabbrielli, M., Giallorenzo, S., Lanese, I., Mauro, J.: Dynamic choreographies: Theory and implementation. Log. Methods Comput. Sci. **13**(2) (2017). https://doi.org/10.23638/LMCS-13(2:1)2017
13. Demangeon, R., Honda, K.: Nested protocols in session types. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7454, pp. 272–286. Springer (2012). https://doi.org/10.1007/978-3-642-32940-1_20, https://doi.org/10.1007/978-3-642-32940-1_20
14. Diffie, W., Hellman, M.E.: New directions in cryptography. IEEE Trans. Inf. Theory **22**(6), 644–654 (1976). https://doi.org/10.1109/TIT.1976.1055638, https://doi.org/10.1109/TIT.1976.1055638
15. Giallorenzo, S., Montesi, F., Peressotti, M.: Choreographies as objects. CoRR **abs/2005.09520** (2020), https://arxiv.org/abs/2005.09520
16. Giallorenzo, S., Montesi, F., Peressotti, M., Richter, D., Salvaneschi, G., Weisenburger, P.: Multiparty Languages: The Choreographic and Multitier Cases. In: 35th European Conference on Object-Oriented Programming, ECOOP 2021, July 12-17, 2021, Aarhus, Denmark (Virtual Conference). LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2021), to appear. Pre-print available at https://fabriziomontesi.com/files/gmprsw21.pdf
17. Object Management Group: Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0/ (2011)
18. Hirsch, A.K., Garg, D.: Pirouette: higher-order typed functional choreographies. Proc. ACM Program. Lang. **6**(POPL), 1–27 (2022). https://doi.org/10.1145/3498684, https://doi.org/10.1145/3498684
19. Honda, K., Yoshida, N., Carbone, M.: Multiparty asynchronous session types. J. ACM **63**(1), 9 (2016). https://doi.org/10.1145/2827695, also: POPL, pages 273–284, 2008
20. Hüttel, H., Lanese, I., Vasconcelos, V.T., Caires, L., Carbone, M., Deniélou, P., Mostrous, D., Padovani, L., Ravara, A., Tuosto, E., Vieira, H.T., Zavattaro, G.: Foundations of session types and behavioural contracts. ACM Comput. Surv. **49**(1), 3:1–3:36 (2016). https://doi.org/10.1145/2873052
21. Intl. Telecommunication Union: Recommendation Z.120: Message Sequence Chart (1996)
22. Jongmans, S., van den Bos, P.: A predicate transformer for choreographies—computing preconditions in choreographic programming. In: Müller, P. (ed.) Programming Languages and Systems - 31st European Symposium on Programming,

ESOP 2022, Proceedings. Lecture Notes in Computer Science, vol. To appear. Springer (2022)
23. López, H.A., Nielson, F., Nielson, H.R.: Enforcing availability in failure-aware communicating systems. In: Albert, E., Lanese, I. (eds.) Procs. FORTE. Lecture Notes in Computer Science, vol. 9688, pp. 195–211. Springer (2016). `https://doi.org/10.1007/978-3-319-39570-8_13`
24. Montesi, F.: Choreographic Programming. Ph.D. Thesis, IT University of Copenhagen (2013)
25. Montesi, F.: Introduction to Choreographies. Cambridge University Press (2022), to appear
26. Murphy VII, T., Crary, K., Harper, R., Pfenning, F.: A symmetric modal lambda calculus for distributed computing. In: 19th IEEE Symposium on Logic in Computer Science (LICS 2004), 14-17 July 2004, Turku, Finland, Proceedings. pp. 286–295. IEEE Computer Society (2004). `https://doi.org/10.1109/LICS.2004.1319623`, `https://doi.org/10.1109/LICS.2004.1319623`
27. Needham, R.M., Schroeder, M.D.: Using encryption for authentication in large networks of computers. Commun. ACM **21**(12), 993–999 (1978). `https://doi.org/10.1145/359657.359659`
28. OpenID Foundation: OpenID Specification. https://openid.net/developers/specs/ (2014)
29. Pierce, B.C.: Types and programming languages. MIT Press (2002)
30. Pohjola, J.Å., Gómez-Londoño, A., Shaker, J., Norrish, M.: Kalas: A verified, end-to-end compiler for a choreographic language. In: Andronick, J., de Moura, L. (eds.) 13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel. LIPIcs, vol. 237, pp. 27:1–27:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2022). `https://doi.org/10.4230/LIPIcs.ITP.2022.27`, `https://doi.org/10.4230/LIPIcs.ITP.2022.27`
31. Vollbrecht, J., Carlson, J.D., Blunk, L., Aboba, D.B.D., Levkowetz, H.: Extensible Authentication Protocol (EAP). RFC 3748 (Jun 2004). `https://doi.org/10.17487/RFC3748`, `https://rfc-editor.org/rfc/rfc3748.txt`
32. W3C: WS Choreography Description Language. http://www.w3.org/TR/ws-cdl-10/ (2004)
33. Weisenburger, P., Wirth, J., Salvaneschi, G.: A survey of multitier programming. ACM Comput. Surv. **53**(4), 81:1–81:35 (2020). `https://doi.org/10.1145/3397495`, `https://doi.org/10.1145/3397495`

A Single Sign-on Authentication

We now implement the single sign-on authentication protocol inspired by the OpenID specification [28], the Choral implementation of which we described in Example 1. We first implement the choreography in a parametric way that allows us to specify the means of communication, and then combine it with the secure communication from the previous example.

The protocol involves three roles with the client C wanting to authenticate with the server S via a third party identity provider I. If authentication succeeds, the client and the server both get a unique token from the identity provider.

We use the following local functions for working with user credentials

$$\mathsf{username}(R) : \mathsf{Credentials}@R \to \mathsf{String}@R$$

$$\begin{aligned}&\mathsf{password}(R) : \mathsf{Credentials}@R \to \mathsf{String}@R\\&\mathsf{calcHash}(R) : \mathsf{String}@R \to \mathsf{String}@R \to \mathsf{String}@R\end{aligned}$$

computing the username and password from a local type $\mathsf{Credentials}@R$ (which can be a pair, for example), and the hash of a string with a given salt. These are mainly used by the client.

In addition, we require functions for retrieving the salt, validating the hash, and creating a token for a given username, which are used by the identity provider:

$$\begin{aligned}\mathsf{getSalt}(R) &: \mathsf{String}@R \to \mathsf{String}@R\\ \mathsf{check}(R) &: \mathsf{String}@R \to \mathsf{String}@R \to \mathsf{Bool}@R\\ \mathsf{createToken}(R) &: \mathsf{String}@R \to \mathsf{String}@R.\end{aligned}$$

Given the above helper functions, the authentication protocol is as follows. Here we use if-then-else as syntactic sugar for **case**:

```
$\mathsf{authenticate}(S, C, I) = \lambda credentials : \mathsf{Credentials}@C.$
  $\lambda comcip : \mathsf{String}@C \to \mathsf{String}@I.\ \lambda comipc : \mathsf{String}@I \to \mathsf{String}@C.$
  $\lambda comips : \mathsf{String}@I \to \mathsf{String}@S.$
    $((\lambda user : \mathsf{String}@I.\ (\lambda salt : \mathsf{String}@C.\ (\lambda hash : \mathsf{String}@I.$
      **if** $\mathsf{check}(I)\ user\ hash$ **then**
        $\mathbf{select}_{I,C}$ ok ($\mathbf{select}_{I,S}$ ok
          $(\lambda token : \mathsf{String}@I.$ **inl** (**pair** $(comipc\ token)\ (comips\ token)))$
          $(\mathsf{createToken}(I)\ user))$
      **else**
        $\mathbf{select}_{I,C}$ ko ($\mathbf{select}_{I,S}$ ko **inr** $()@I))$
     $(comcip\ (\mathsf{calcHash}(C)\ salt\ (\mathsf{password}(C)\ credentials))))$
    $(comipc\ (\mathsf{getSalt}(I)\ user)))$
   $(comcip\ (\mathsf{username}(C)\ credentials)))$
```

As mentioned, the choreography is parametrised over three channels between the participants, allowing the communication to be customized (*comcip*, *comipc* and *comips*). The client first sends their username to the identity provider who replies with the appropriate salt. The client then calculates a salted hash of their password and sends it back to the identity provider. Finally, the identity provider validates the hash and either sends a token to both participants or returns a unit. The shared token is again represented using a pair of equal values, visible from the type of the choreography:

$$\begin{aligned}&\mathsf{Credentials}@C \to (\mathsf{String}@C \to \mathsf{String}@I) \to (\mathsf{String}@I \to \mathsf{String}@C)\\&\qquad \to (\mathsf{String}@I \to \mathsf{String}@S) \to ((\mathsf{String}@C \times \mathsf{String}@S) + ()@I)\end{aligned}$$

We can now combine $\mathsf{authenticate}(S, C, I)$ and $\mathsf{makeSecureChannels}(P, Q)$ (from Section 4.1) to can obtain a choreography $\mathsf{main}(S, C, I)$ that carries out the authentication securely. Using $\mathsf{makeSecureChannels}(P, Q)$, the participants first establish secure channels backed by encryption keys derived using $\mathsf{diffieHellman}(P, Q)$.

After the secure communication is in place, the participants can execute the authentication protocol specified by $\mathsf{authenticate}(S, C, I)$.

```
$\mathsf{main}(S, C, I) =$
  $(\lambda k1 : \mathsf{Int}@C \times \mathsf{Int}@I.\ \lambda k2 : \mathsf{Int}@I \times \mathsf{Int}@S.$
    $(\lambda c1 : (\mathsf{String}@C \to \mathsf{String}@I) \times (\mathsf{String}@I \to \mathsf{String}@C).$
     $\lambda c2 : (\mathsf{String}@I \to \mathsf{String}@S) \times (\mathsf{String}@S \to \mathsf{String}@I).$
      $(\lambda t : (\mathsf{String}@C \times \mathsf{String}@S) + ()@I.$
        $\textbf{case}\ t\ \textbf{of}$
          $\textbf{Inl}\ x \Rightarrow$ "Authentication successful"$@C$
          $\textbf{Inr}\ x \Rightarrow$ "Authentication failed"$@C)$
      $(\mathsf{authenticate}(S, C, I)\ (\textbf{fst}\ c1)\ (\textbf{snd}\ c1)\ (\textbf{fst}\ c2)))$
    $(\mathsf{makeSecureChannels}(C, I)\ k1)\ (\mathsf{makeSecureChannels}(I, S)\ k2))$
  $(\mathsf{diffieHellman}(C, I)\ csk\ ipsk\ csg\ ipsg\ csp\ ipsp)$
  $(\mathsf{diffieHellman}(I, S)\ ipsk\ ssk\ ipsg\ ssg\ ipsp\ ssp)$
```

In this example, the client simply reports whether the authentication has succeeded with a value, which can be checked in a larger context. Or, alternatively, we could parameterise main over choreographic continuations to be invoked in case of success or failure.

We denote by Γ the set of typings we have given so far in this section. Then we can type $\{S, C, I\}; \emptyset; \Gamma \vdash \mathsf{main}(S, C, I) : \mathsf{String}@C$.

B Full definitions and proofs

Definition 4 (Free Variables). *Given a choreography M, the free variables of M, $\mathrm{fv}(M)$ are defined as:*

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

Definition 5 (Merging). *Given two behaviours B and B', $B \sqcup B'$ is defined as follows.*

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

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

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

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

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

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

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

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

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

$$\frac{\Theta; \Sigma; \Gamma \vdash M : T \quad S, R \in \Theta}{\Theta; \Sigma; \Gamma \vdash \textsf{select}_{S,R}\ l\ M : T} \text{[TSel]}$$

$$\frac{f(\vec{R'}) : T \in \Gamma \quad \vec{R} \subseteq \Theta \quad ||\vec{R}|| = ||\vec{R'}|| \quad \text{distinct}(\vec{R})}{\Theta; \Sigma; \Gamma \vdash f(\vec{R}) : T[\vec{R'} := \vec{R}]} \text{[TDef]}$$

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

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

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

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

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

$$\frac{\forall f(\vec{R}) \in \textsf{domain}(D) : \quad f(\vec{R}) : T \in \Gamma \quad \vec{R}; \Sigma; \Gamma \vdash D(f(\vec{R})) : T \quad \text{distinct}(\vec{R}) \quad \vec{R} \subseteq \Theta}{\Theta; \Sigma; \Gamma \vdash D} \text{[TDefs]}$$

Figure 4. Full set of typing rules for Chorλ.

$$\frac{}{\lambda x : T.M\ V \xrightarrow{\tau,\emptyset}_D M[x := V]}\ [\textsc{AppAbs}] \qquad \frac{M \xrightarrow{\ell,\mathbf{R}}_D M'}{\lambda x : T.M \xrightarrow{\lambda,\mathbf{R}}_D \lambda x : T.M'}\ [\textsc{InAbs}]$$

$$\frac{M \xrightarrow{\ell,\mathbf{R}}_D M' \quad \ell = \lambda \Rightarrow \mathbf{R} \cap \mathrm{roles}(N) = \emptyset}{M\ N \xrightarrow{\tau,\mathbf{R}}_D M'\ N}\ [\textsc{App1}]$$

$$\frac{N \xrightarrow{\tau,\mathbf{R}}_D N'}{V\ N \xrightarrow{\tau,\mathbf{R}}_D V\ N'}\ [\textsc{App2}] \qquad \frac{N \xrightarrow{\tau,\mathbf{R}}_D N' \quad \mathbf{R} \cap \mathrm{roles}(M) = \emptyset}{M\ N \xrightarrow{\tau,\mathbf{R}}_D M\ N'}\ [\textsc{App3}]$$

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

$$\frac{M_1 \xrightarrow{\ell,\mathbf{R}}_D M_1' \quad M_2 \xrightarrow{\ell,\mathbf{R}}_D M_2' \quad \mathbf{R} \cap \mathrm{roles}(N) = \emptyset}{\textsf{case}\ N\ \textsf{of Inl}\ x \Rightarrow M_1\textsf{; Inr}\ x' \Rightarrow M_2 \xrightarrow{\ell,\mathbf{R}}_D \textsf{case}\ N\ \textsf{of Inl}\ x \Rightarrow M_1'\textsf{; Inr}\ x' \Rightarrow M_2'}\ [\textsc{InCase}]$$

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

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

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

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

$$\frac{\mathrm{fv}(V) = \emptyset}{\textsf{com}_{S,R}\ V \xrightarrow{\tau,\{S,R\}}_D V[S := R]}\ [\textsc{Com}] \qquad \frac{}{\textsf{select}_{S,R}\ l\ M \xrightarrow{\tau,\{S,R\}}_D M}\ [\textsc{Sel}]$$

$$\frac{M \xrightarrow{\ell,\mathbf{R}}_D M' \quad \mathbf{R} \cap \{S,R\} = \emptyset}{\textsf{select}_{S,R}\ \ell\ M \xrightarrow{\ell,\mathbf{R}}_D \textsf{select}_{S,R}\ \ell\ M'}\ [\textsc{InSel}] \qquad \frac{M \rightsquigarrow^* N \quad N \xrightarrow{\tau,\mathbf{R}} N'}{M \xrightarrow{\tau,\mathbf{R}}_D M'}\ [\textsc{Str}]$$

Figure 5. Semantics of Chorλ

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

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

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

Choreographies:

$$[\![M\ N]\!]_R = \begin{cases} [\![M]\!]_R\ \ [\![N]\!]_R & \text{if } R \in \text{roles}(\text{type}(M)) \text{ or } R \in \text{roles}(M) \cap \text{roles}(N) \\ \bot & \text{if } [\![M]\!]_R = [\![N]\!]_R = \bot \\ [\![M]\!]_R & \text{if } [\![N]\!]_R = \bot \\ [\![N]\!]_R & \text{otherwise} \end{cases}$$

$$[\![\lambda x : T.M]\!]_R = \begin{cases} \lambda x.\ [\![M]\!]_R & \text{if } R \in \text{roles}(\text{type}(x : T.M)) \\ \bot & \text{otherwise} \end{cases}$$

$$\left[\!\!\left[\textsf{case}\ M\ \textsf{of Inl}\ x \Rightarrow N;\ \textsf{Inr}\ x' \Rightarrow N'\right]\!\!\right]_R =$$

$$\begin{cases} \textsf{case}\ [\![M]\!]_R\ \textsf{of Inl}\ x \Rightarrow [\![N]\!]_R;\ \textsf{Inr}\ x' \Rightarrow \left[\!\!\left[N'\right]\!\!\right]_R & \text{if } R \in \text{roles}(\text{type}(M)) \\ \bot & \text{if } [\![M]\!]_R = [\![N]\!]_R = \left[\!\!\left[N'\right]\!\!\right]_R = \bot \\ [\![M]\!]_R & \text{if } [\![N]\!]_R = \left[\!\!\left[N'\right]\!\!\right]_R = \bot \\ [\![N]\!]_R \sqcup \left[\!\!\left[N'\right]\!\!\right]_R & \text{if } [\![M]\!]_R = \bot \\ (\lambda x'' : \bot.\ [\![N]\!]_R \sqcup \left[\!\!\left[N'\right]\!\!\right]_R)\ \ [\![M]\!]_R & \text{for some } x'' \notin \text{fv}(N) \cup \text{fv}(N') \\ & \text{otherwise} \end{cases}$$

$$\left[\!\!\left[\textsf{select}_{S,S'}\ \ell\ M\right]\!\!\right]_R = \begin{cases} \oplus_{S'}\ \ell\ \ [\![M]\!]_R & \text{if } R = S \neq S' \\ \&_S\{\ell : [\![M]\!]_R\} & \text{if } R = S' \neq S \\ [\![M]\!]_R & \text{otherwise} \end{cases}$$

$$\left[\!\!\left[\textsf{com}_{S,S'}\right]\!\!\right]_R = \begin{cases} \lambda x.x & \text{if } R = S = S' \\ \textsf{send}_{S'} & \text{if } R = S \neq S' \\ \textsf{recv}_S & \text{if } R = S' \neq S \\ \bot & \text{otherwise} \end{cases}$$

$$[\![()@S]\!]_R = \begin{cases} () & \text{if } S = R \\ \bot & \text{otherwise} \end{cases} \qquad [\![x]\!]_R = \begin{cases} x & \text{if } R \in \text{roles}(\text{type}(x)) \\ \bot & \text{otherwise} \end{cases}$$

$$\left[\!\!\left[f(\vec{R})\right]\!\!\right]_R = \begin{cases} f_i(R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_n) & \text{if } \vec{R} = R_1, \ldots, R_{i-1}, R, R_{i+1}, \ldots, R_n \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\textsf{Pair}\ V\ \ V']\!]_R = \begin{cases} \textsf{Pair}\ [\![V]\!]_R\ \ [\![V']\!]_R & \text{if } R \in \text{roles}(\text{type}(V) \times \text{type}(V')) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\textsf{fst}]\!]_R = \begin{cases} \textsf{fst} & \text{if } R \in \text{roles}(\text{type}(\textsf{fst})) \\ \bot & \text{otherwise} \end{cases} \qquad [\![\textsf{snd}]\!]_R = \begin{cases} \textsf{snd} & \text{if } R \in \text{roles}(\text{type}(\textsf{snd})) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\textsf{Inl}\ V]\!]_R = \begin{cases} \textsf{Inl}\ [\![V]\!]_R & \text{if } R \in \text{roles}(\text{type}(\textsf{Inl}\ V)) \\ \bot & \text{otherwise} \end{cases} \qquad [\![\textsf{Inr}\ V]\!]_R = \begin{cases} \textsf{Inr}\ [\![V]\!]_R & \text{if } r \in \text{roles}(\text{type}(\textsf{Inr}\ V)) \\ \bot & \text{otherwise} \end{cases}$$

Types:

$$[\![T \to_\rho T']\!]_R = \begin{cases} [\![T]\!]_R \to [\![T']\!]_R & \text{if } R \in \rho \cup \text{roles}(T) \cup \text{roles}(T') \\ \bot & \text{otherwise} \end{cases} \qquad [\![()@S]\!]_R = \begin{cases} () & \text{if } S = R \\ \bot & \text{otherwise} \end{cases}$$

$$[\![T \times T']\!]_R = \begin{cases} [\![T]\!]_R \times [\![T']\!]_R & \text{if } R \in \text{roles}(T \times T') \\ \bot & \text{otherwise} \end{cases} \qquad [\![T + T']\!]_R = \begin{cases} [\![T]\!]_R + [\![T']\!]_R & \text{if } R \in \text{roles}(T + T') \\ \bot & \text{otherwise} \end{cases}$$

$$\left[\!\!\left[t@\vec{R}\right]\!\!\right]_R = \begin{cases} t_i & \text{if } \vec{R} = R_1, \ldots, R_{i-1}, R, R_{i+1}, \ldots, R_n \\ \bot & \text{otherwise} \end{cases}$$

Definitions:

$$[\![D]\!] = \{f_i(R_1, \ldots, R_{i-1}, R_{i+1}, \ldots, R_n) \mapsto [\![D(f(R_1, \ldots, R_n))]\!]_{R_i} \mid f(R_1, \ldots, R_n) \in \textsf{domain}(D)\}\}$$

Figure 6. Projecting Chorλ onto a role

C Proof of Theorem 1

Proof (Proof of Theorem 1). We prove this by induction on the typing derivation of $\Theta; \Sigma; \Gamma \vdash M : T$. Most cases either M is a value, or the result follows from simple induction, we go through the rest.

- Assume we use rule TApp, so $M = N_1\ N_2$, $\Theta; \Sigma; \Gamma \vdash N_1 : T' \rightarrow_\rho T$, and $\Theta; \Sigma; \Gamma \vdash N_2 : T'$. If N_1 or N_2 is not a value then the result follows from induction and using rule App1 or rule App2. Otherwise, we have four cases:
  - Assume $\Theta; \Sigma; \Gamma \vdash N_1 : T' \rightarrow_\rho T$ uses rule TAbs. Then the result follows using rule AppAbs.
  - Assume $\Theta; \Sigma; \Gamma \vdash N_1 : T' \rightarrow_\rho T$ uses rule TCom. Then, since M is closed, the result follows by rule Com.
  - Assume $\Theta; \Sigma; \Gamma \vdash N_1 : T' \rightarrow_\rho T$ uses rule TProj1. Then, since M is closed and N_2 is a value, $N_2 = \textsf{Pair}\ V\ V'$, and consequently the result follows using rule Proj1.
  - Assume $\Theta; \Sigma; \Gamma \vdash N_1 : T' \rightarrow_\rho T$ uses rule TProj2. Then, since M is closed and N_2 is a value, $N_2 = \textsf{Pair}\ V\ V'$, and consequently the result follows using rule Proj2.
- Assume we use rule TCase, so $M = \textsf{case}\ N_1\ \textsf{of Inl}\ x \Rightarrow N_2\textsf{; Inr}\ x' \Rightarrow N_3$, $\Theta; \Sigma; \Gamma \vdash N_1 : T_1 + T_2$, $\Theta; \Sigma; \Gamma, x : T_1 \vdash N_2 : T$, and $\Theta; \Sigma; \Gamma, x' : T_2 \vdash N_3 : T$. Then if N_1 is not a value the result follows from induction and using rule Case. If N_1 is a value then, since M is closed, either $N_1 = \textsf{Inl}\ V$ or $N_1 = \textsf{Inr}\ V$, and the result follows by rule CaseL or rule CaseR respectively.
- Assume we use rule TSel so $M = \textsf{select}_{S,R}\ l\ N : T$, $\Theta; \Sigma; \Gamma \vdash N : T$, and $S, R \in \Theta$. Then the result follows from using rule Sel.
- Assume we use rule TDef and $M = f(\vec{R})$, $f(\vec{R'} : T' \in \Gamma$, $\vec{R} \subseteq \Theta$, $||\vec{R}|| = ||\vec{R'}||$, distinct$(\vec{R})$, and $T = T'[\vec{R'} := \vec{R}]$. Then the result follows from D containing f, $\Theta; \Sigma; \Gamma \vdash D$ and rule Def.

D Proof of Theorem 2

Lemma 1. *Given a choreography, M, if $\Theta; \Sigma; \Gamma \vdash M$ then $\Theta \cup \Theta'; \Sigma; \Gamma \vdash M$*

Proof. Follows from the typing rules only ever discussing subsets of Θ.

Lemma 2 (Type preservation under rewriting). *Let M be a choreography. If there exists a typing context $\Theta; \Sigma; \Gamma$ such that $\Theta; \Sigma; \Gamma \vdash M : T$, then $\Theta; \Sigma; \Gamma \vdash M' : T$ for any M' such that $M \rightsquigarrow M'$.*

Proof. We prove this by case analysis of the the rewriting rules:

rule R-AbsR Then $M = ((\lambda x : T_1.N_1)\ N_2)\ N_3$, and from the typing rules we get that there exist T_2 and ρ such that $\Theta; \Sigma; \Gamma, x : T_1 \vdash N_1 : T_2 \rightarrow_\rho T$, $\Theta; \Sigma; \Gamma \vdash N_2 : T_1$, and $\Theta; \Sigma; \Gamma \vdash N_3 : T_2$, and $M' = (\lambda x : T_1.(N_1\ N_3))\ N_2$. The result follows from using rules TApp and TAbs and $x \notin \text{fv}(N_3)$.

rule R-AbsL Then $M = N_1\ ((\lambda x : T_1.N_3)\ N_2)$, $\Theta; \Sigma; \Gamma, x : T_1 \vdash N_1 : T_2 \to_\rho T$, $\Theta; \Sigma; \Gamma \vdash N_2 : T_1$, and $\Theta; \Sigma; \Gamma \vdash N_3 : T_2$, and $M' = (\lambda x : T_1.(N_1\ N_3))\ N_2$. The result follows from using rules TApp and TAbs and $x \notin \mathrm{fv}(N_1)$.

rule R-CaseR Then $M = \textsf{case}\ N_1\ \textsf{of Inl}\ x \Rightarrow N_2\textsf{; Inr}\ x' \Rightarrow N_3)\ N_4$, and from the typing rules we get that there exist T_1, T_2, T_3, and ρ such that $\Theta; \Sigma; \Gamma \vdash N_1 : T_1 + T_2$, $\Theta; \Sigma; \Gamma, x : T_1 \vdash N_2 : T_3 \to_\rho T$, $\Theta; \Sigma; \Gamma, x' : T_2 \vdash N_3 : T_3 \to_\rho T$, and $\Theta; \Sigma; \Gamma \vdash N_4 : T_3$, and $M' = \textsf{case}\ N_1\ \textsf{of Inl}\ x \Rightarrow N_2\ N_4\textsf{; Inr}\ x' \Rightarrow N_3\ N_4)$. The result follows from using rules TAbs and TCase and $x, x' \notin \mathrm{fv}(N_4)$.

rule R-CaseL This case is similar to the previous.

rule R-SelR Then $M = N_1\ (\textsf{select}_{S,R}\ l\ N_2)$, and from the typing rules we get that there exist T', and ρ such that $\Theta; \Sigma; \Gamma \vdash N_1 : T' \to_\rho T$ and $\Theta; \Sigma; \Gamma \vdash N_2 : T'$, and $M' = \textsf{select}_{S,R}\ l\ (N_1\ N_2)$. The result follows from using rules TAbs and TSel.

rule R-SelL This case is similar to the previous.

Proof (Proof of Theorem 2). We prove this by induction on the derivation of $M \xrightarrow{\tau,\mathbf{R}}_D M'$. The cases for rules AppAbs, App1 and App2 are standard for simply-typed λ-calculus. And the cases for rules InAbs, App3, Case, InCase and InSel follow from simple induction. We go through the rest.

- Assume we use rule CaseL. Then we know that $M = \textsf{case Inl}\ V\ \textsf{of Inl}\ x \Rightarrow N_1\textsf{; Inr}\ x' \Rightarrow N_2$, and from the typing rules we get that there exists T' such that $\Theta; \Sigma; \Gamma \vdash V : T'$ and $\Theta; \Sigma; \Gamma, x : T' \vdash N_1 : T$. Therefore, $\Theta; \Sigma; \Gamma \vdash N_1[x := V] : T$.
- Assume we use rule CaseR. This is similar to the previous case.
- Assume we use rule Proj1. Then we know $M = \textsf{fst Pair}\ V\ V'$, and from the typing rules we get that $\Theta; \Sigma; \Gamma \vdash V : T$.
- Assume we use rule Proj2. This is similar to the previous case.
- Assume we use rule Def. From the typing of M we get that there exists $f(\vec{R'}) : T \in \Gamma$ such that $||\vec{R}|| = ||\vec{R'}||$, $\vec{R} \subseteq \Theta$, and $\mathrm{distinct}(\vec{R'})$. From the typing of D we get that $\mathrm{distinct}(\vec{R'})$ and $\vec{R'}; \Sigma; \Gamma \vdash D(f(\vec{R'}))$. Therefore, by Lemma 1, we get $\Theta; \Sigma; \Gamma \vdash D(f(\vec{R'}))$.
- Assume we use rule Com. Then we know that $M = \textsf{com}_{S,R}\ V$, $\mathrm{fv}(V) = \emptyset$, and there exists T' such that $\Theta; \Sigma; \Gamma \vdash V : T'$, $\mathrm{roles}(T') = \{S\}$, and $T = T'[S := R]$. We see from our typing rules that the only time we use roles not mentioned in the choreography in typing is when handling free variables. Therefore we get that $\Theta; \Sigma; \Gamma \vdash V[S := R] : T$.
- Assume we use rule Sel. Then we know that $M = \textsf{select}_{S,R}\ l\ N$ and $\Theta; \Sigma; \Gamma \vdash N : T$. The result follows.
- Assume we use rule Str. Then the result follows from Lemma 2 and induction.