JFP, **??** pages, 2023. © Cambridge University Press 2023
doi:10.1017/xxxxx

Alice or Bob?: Process Polymorphism in Choreographies

EVA GRAVERSEN

Department of Mathematics and Computer Science, University of Southern Denmark e-mail: efgraversen@imada.sdu.dk

ANDREW K. HIRSCH

Department of Computer Science and Engineering, University at Buffalo, SUNY e-mail: akhirsch@buffalo.edu

FABRIZIO MONTESI

Department of Mathematics and Computer Science, University of Southern Denmark e-mail: fmontesi@imada.sdu.dk

Abstract

We present PolyChorλ, a language for higher-order functional *choreographic programming*—an emerging paradigm for concurrent programming. In choreographic programming, programmers write the desired cooperative behaviour of a system of processes and then compile it into an implementation for each process, a translation called *endpoint projection*. Unlike its predecessor, Chorλ, PolyChorλ has both type and *process* polymorphism inspired by System F_ω. That is, PolyChorλ is the first (higher-order) functional choreographic language which gives programmers the ability to write generic choreographies and determine the participants at runtime. This novel combination of features also allows PolyChorλ processes to communicate *distributed values*, leading to a new and intuitive way to write delegation. While some of the functional features of PolyChorλ give it a weaker correspondence between the semantics of choreographies and their endpoint-projected concurrent systems than some other choreographic languages, we still get the hallmark end result of choreographic programming: projected programs are deadlock-free by design.

Key Words: Choreographic Programming, Concurrency, λ-calculus, Type Systems, Polymorphism

1 Introduction

Distributed systems involve interacting processes. Usually, programmers write one program per process, and then compose those programs in parallel. These programs contain *send* and *receive* expressions which transmit data between processes. Predicting how the composition of programs based on this method is challenging, so it is easy to write code that *deadlocks*, or gets stuck because patterns of sends and receives do not match. *Session types* (Honda, 1993; Honda et al., 1998) can be used to describe the patterns of sends and receives in a program, offering a foundation for static analyses aimed at preventing communication mismatches and deadlocks (Scalas and Yoshida, 2019; Caires and Pfenning, 2010; Wadler, 2012; DeYoung et al., 2012; Honda et al., 2016; Dardha et al., 2012). Working with session types enables the programmer to ensure the communications in their system follow compatible send/receive patterns.

Alternatively, developers can use a *choreographic* language to program the interactions that they wish to take place in the system directly from a global viewpoint (Montesi, 2023). *Choreographic programming* (Montesi, 2013) is a programming paradigm based on this idea with particularly well-explored foundations (Cruz-Filipe and Montesi, 2020; Montesi, 2023) and promising developments (see, e.g., Dalla Preda et al., 2017; Giallorenzo et al., 2021; Carbone and Montesi, 2013; Cruz-Filipe et al., 2022; Hirsch and Garg, 2022; Jongmans and van den Bos, 2022; López et al., 2016). In this paradigm a programmer writes one program as a choreography, which is then compiled to a program for each process that is guaranteed to be correct by construction. Unlike session types, which only allow local code to be checked against them, choreographies compile to the local code itself. The syntax of choreographic programming languages is typically inspired by security protocol notation (Needham and Schroeder, 1978), where send and receive commands are written together as part of atomic instructions for expressing communication. This has two key advantages. First, it gives programmers the power to express the desired communication flow among processes, but without the burden of manually coding send and receive actions. Second, it ensures that there is no mismatch which can cause deadlock, a property that has become known as *deadlock-freedom by design* (Carbone and Montesi, 2013).

To see the power of this, consider the (in)famous bookseller example—a recurring example in the literature of choreographic programming and session types (Carbone and Montesi, 2013; Honda et al., 2016; Montesi, 2023). Buyer wants to buy a book from Seller. To this end, Buyer sends the title of the book—say, "The Importance of Being Earnest"—to Seller, who then sends back the price. Buyer then can compare the price with its budget and based on the result informs Seller that they want to buy the book if it is within their budget, or informs them that they do not want to buy the book otherwise. We can describe this via the following choreography:

$$\begin{array}{l}
\mathsf{let}\ x = \mathsf{com}_{\mathsf{Buyer},\mathsf{Seller}}\ (\text{“The Importance of Being Earnest”}\ @\ \mathsf{Buyer}) \\
\mathsf{in\ let}\ y = \mathsf{com}_{\mathsf{Seller},\mathsf{Buyer}}\ (\mathsf{price_lookup}\ x) \\
\quad \mathsf{in\ if}\ y < \mathsf{budget} \\
\qquad \mathsf{then\ select}_{\mathsf{Buyer},\mathsf{Seller}}\ \mathsf{Buy}\ \ ((\,)\ @\ \mathsf{Seller}) \\
\qquad \mathsf{else\ select}_{\mathsf{Buyer},\mathsf{Seller}}\ \mathsf{Quit}\ \ ((\,)\ @\ \mathsf{Seller})
\end{array} \tag{1.1}$$

In Listing (1.1), as in all choreographic programs, computation takes place among multiple *processes* communicating via message passing. Values are located at processes; for example, in the first line of the choreography, the title of the book is initially located at Buyer. The function $\mathsf{com}_{\mathsf{P},\mathsf{Q}}$ communicates a value from the process P to the process Q. It takes a local value at P and returns a local value at Q.[1] Thus, x represents the string "The Importance of Being Earnest" at the process Seller, while y represents the price at the process Buyer. Finally, we check locally if the book's price is in Buyer's budget. Either way, we use function select to send a label from Buyer to Seller representing Buyer's choice to either proceed with the purchase or not. Either way, the choreography returns the dummy value $()$ at Seller.

[1] Formally, we require a type annotation on $\mathsf{com}_{\mathsf{P},\mathsf{Q}}$ (see Section 3). We elide this here for clarity.

While most of the early work on choreographies focused on simple lower-order imperative programming like in the example above, recent work has shown how to develop higher-order choreographic programming languages. These languages allow a programmer to write deadlock-free code using the usual abstractions of higher-order programming, such as objects (Giallorenzo et al., 2020) and higher-order functions (Hirsch and Garg, 2022; Cruz-Filipe et al., 2022).

For instance, Listing (1.1) bakes in the title and the value of the book. However, we may want to use this code whenever Buyer wants to buy any book, and let Buyer use any local function to decide whether to buy the book at a price.

$$
\begin{array}{l}
\lambda\, \mathsf{title} : \mathsf{String} \mathbin{@} \mathsf{Buyer}. \\
\quad \lambda\, \mathsf{buyAtPrice?} : \mathsf{Int} \mathbin{@} \mathsf{Buyer} \to_{\varnothing} \mathsf{Bool} \mathbin{@} \mathsf{Buyer}. \\
\quad\quad \mathsf{let}\ x = \mathsf{com}_{\mathsf{Buyer},\mathsf{Seller}}\ \mathsf{title} \\
\quad\quad \mathsf{in\ let}\ y = \mathsf{com}_{\mathsf{Seller},\mathsf{Buyer}}\ (\mathsf{price\_lookup}\ x) \\
\quad\quad\quad \mathsf{in\ if}\ \mathsf{buyAtPrice?}\ y \\
\quad\quad\quad\quad \mathsf{then\ select}_{\mathsf{Buyer},\mathsf{Seller}}\ \mathsf{Buy}\ \ (() \mathbin{@} \mathsf{Seller}) \\
\quad\quad\quad\quad \mathsf{else\ select}_{\mathsf{Buyer},\mathsf{Seller}}\ \mathsf{Quit}\ \ (() \mathbin{@} \mathsf{Seller})
\end{array}
\tag{1.2}
$$

Note the type of the function buyAtPrice?: it takes as input not just an integer, but an integer at Buyer; similarly, it returns a Boolean at Buyer. Moreover, the arrow is annotated with a set of processes, which in this case is empty ($\varnothing$). Other than those processes named in the input and output types of the function, these are the only processes who may participate in the computation of that function. Since that set is empty here, *no* other process may participate in the function—i.e., buyAtPrice? is local to Buyer. (Sometimes we wish for other processes to participate in the computation of a function, as we will see in Example 3.)

However, not every function with an $\varnothing$ annotation is local. For instance, $\mathsf{com}_{\mathsf{P},\mathsf{Q}}$ is a function compatible with type $\tau \to_{\varnothing} \tau$ for any type τ. Despite the fact that $\mathsf{com}_{\mathsf{P},\mathsf{Q}}$ is clearly not local, only P and Q are involved in the communication, leading to the $\varnothing$ annotation. Similarly, just because the input and output of a function are at different locations does not mean that the function involves communication: for instance, it might be a constant function. The choreography $\lambda\, x : \mathsf{Int} \mathbin{@} \mathsf{P}.\ 5 \mathbin{@} \mathsf{Q}$ has the same type as a communication of an integer from P to Q: $\mathsf{Int} \mathbin{@} \mathsf{P} \to_{\varnothing} \mathsf{Int} \mathbin{@} \mathsf{Q}$.

A programmer using a higher-order choreographic language, like a programmer using any higher-order programming language, can write a program once and use it in a large number of situations. For instance, by supplying different values of title and buyAtPrice?, the choreography in Listing (1.2) can be used to buy several different titles and Buyer can determine if they are willing to buy the book at the price using any method they desire.

While the move from first-order programming to higher-order programming is significant, previous work on the theoretical foundations of higher-order choreographic programming still did not account for other forms of abstraction (Hirsch and Garg, 2022; Cruz-Filipe et al., 2022). In particular, they did not allow for *polymorphism*, where programs can abstract over types as well as data, allowing them to operate in many more settings; nor did they allow for *delegation*, where one process can ask another process to act in its stead.

These forms of abstraction are relatively standard: delegation is an important operation in concurrent calculi, and polymorphism is vital to modern programming. In choreographic programming, however, another form of abstraction becomes natural: abstraction over processes. Current higher-order choreographic languages require that code mention concrete process names. However, we often want to write more-generic code, allowing the same code to run on many processes. For example, Listing (1.2) allows Buyer to decide whether to buy a book from Seller using any local function buyAtPrice?. It would be more natural to write Seller as a book-selling *service* which different clients could interact with in the same way to buy a book.

In this paper, we tackle three new features for choreographic languages. Firstly, we show that abstraction over processes is a type of polymorphism, which we refer to as *process polymorphism*. Secondly, we extend Chorλ—a simply-typed functional choreographic language—with polymorphism, including process polymorphism, and call this new language PolyChorλ. Thirdly, we add the ability to communicate distributed values such as functions. This gives us the ability to *delegate* (that is, to send code to another process, which that process is then expected to run), giving a clean language to study all three forms of abstraction.

Let us examine the bookseller *service* in our extended language:

$$\begin{array}{l}
\Lambda B :: \mathsf{Proc}.\\
\quad \lambda\, \mathsf{title} : \mathsf{String} \mathbin{@} B.\\
\quad\quad \lambda\, \mathsf{buyAtPrice?} : \mathsf{Int} \mathbin{@} B \rightarrow_{\varnothing} \mathsf{Bool} \mathbin{@} B.\\
\quad\quad\quad \mathsf{let}\ x = \mathsf{com}_{B,\mathsf{Seller}}\ \mathsf{title}\\
\quad\quad\quad \mathsf{in\ let}\ y = \mathsf{com}_{\mathsf{Seller},B}\ (\mathsf{price_lookup}\ x)\\
\quad\quad\quad\quad \mathsf{in\ if}\ \mathsf{buyAtPrice?}\ y\\
\quad\quad\quad\quad\quad \mathsf{then\ select}_{B,\mathsf{Seller}}\ \mathsf{Buy}\ \ ((\,) \mathbin{@} \mathsf{Seller})\\
\quad\quad\quad\quad\quad \mathsf{else\ select}_{B,\mathsf{Seller}}\ \mathsf{Quit}\ \ ((\,) \mathbin{@} \mathsf{Seller})
\end{array} \tag{1.3}$$

This program allows a process named B to connect with Seller to buy a book. B then provides a string title and a decision function buyAtPrice?. Thus, we no longer have to write a separate function for every process which may want to buy a book from Seller.

While this addition may appear simple, it poses some unique theoretical challenges. First, the goal of a choreographic language is to compile a global program to one local program per process. However, since B does not represent any particular process, it is unclear how to compile the polymorphic code above. We solve this problem via a simple principle: each process knows its identity. With this principle in place, we can compile the code to a conditional in each process: one option to run if they take the role of B, and the other to run if they do not.

Notably, each process chooses dynamically which interpretation of the code to run. This flexibility is important, since we may want to allow different processes to occupy B's place dynamically. For instance, we can imagine a situation where Buyer_1 and Buyer_2 work together to buy a particularly expensive book: perhaps they compare bank accounts, and whoever has more money buys the book for them to share. This can be achieved in our system with Listing 1.4, where seller_service is the name of the choreography from

Listing 1.3:

$$
\begin{array}{l}
\lambda\, \mathsf{title} : \mathsf{String} \mathbin{@} \mathsf{Buyer}_1 . \\
\quad \mathsf{let}\ x = \mathsf{com}_{\mathsf{Buyer}_1,\mathsf{Buyer}_2}\ \mathsf{bank\_balance}_1 \\
\quad \mathsf{in\ if}\ x < \mathsf{bank\_balance}_2 \\
\qquad \mathsf{then\ select}_{\mathsf{Buyer}_2,\mathsf{Buyer}_1}\ \mathsf{Me}\ \mathsf{select}_{\mathsf{Buyer}_2,\mathsf{Seller}}\ \mathsf{Me} \\
\qquad\qquad \left(\begin{array}{ll} \mathsf{seller\_service} & \mathsf{Buyer}_2\ (\mathsf{com}_{\mathsf{Buyer}_1,\mathsf{Buyer}_2}\ \mathsf{title}) \\ & (\lambda\, z.\, z < \mathsf{bank\_balance}_2) \end{array} \right) \\
\qquad \mathsf{else\ select}_{\mathsf{Buyer}_2,\mathsf{Buyer}_1}\ \mathsf{You}\ \mathsf{select}_{\mathsf{Buyer}_2,\mathsf{Seller}}\ \mathsf{Them} \\
\qquad\qquad (\mathsf{seller\_service}\ \mathsf{Buyer}_1\ \mathsf{title}\ (\lambda\, z.\, z < \mathsf{bank\_balance}_1))
\end{array}
\tag{1.4}
$$

Here Buyer_1 sends its bank balance, $\mathsf{bank\_balance}_1$ to Buyer_2, who compares the received value with its own balance, $\mathsf{bank\_balance}_2$. If Buyer_2 has the larger balance, then it informs Buyer_1 and Seller that Buyer_2 will be buying the book by means of the label "Me". Buyer_1 then sends the book title to Buyer_2, which allows Buyer_2 and Seller to initiate the seller_service choreography using a buyAtPrice? function that checks whether the price is less than Buyer_2's bank balance. If Buyer_1 has the larger balance then Buyer_2 again informs Buyer_1 and Seller of who will be performing the role of buyer for the rest of the protocol, "You" and "Them" respectively. Then Buyer_1 enters the seller_service choreography with similar input to the first case, except the title and buyAtPrice? are now located at Buyer_1.

A related challenge shows up in the operational semantics of our extended language. Languages like PolyChorλ generally have operational semantics which match the semantics of the compiled code by allowing *out-of-order execution*: redices in different processes might be reduced in any order. However, care must be taken with process polymorphism, since it may not be clear whether two redices are in the same or different processes.

In addition to type and process polymorphism, PolyChorλ is the first choreographic language to allow the communication of distributed values: values not located entirely at the sender. These values include full choreographies described by distributed functions, which can be used to model delegation. To see how process polymorphism and communication of distributed values enables delegation, consider Figure 1. Here, when a buyer asks for a book, the seller first checks whether it is in stock. If it is, the sale continues as normal. If not, the seller delegates to a second seller, which may sell the book to the buyer.

In more detail, after ascertaining that the book is not in stock, Seller informs B and Seller_2 that the rest of the choreography will be executed by Seller_2 in the place of Seller using two selections with label "Delegate". Then, Seller sends first the rest of the choreography to Seller_2, followed the title of the requested book. Seller_2 uses its own lookup function to execute the code in Listing 1.2. Both Seller_2 and B need to be informed that the delegation is happening, since B needs to know that it should interact with Seller_2 rather than Seller.

In general, delegation poses a challenge: the third-party processes involved in a communicated value (processes that are neither the sender nor the receiver, such as B above) might need to change who they are going to interact with by swapping names (for instance, swapping Seller_2 and Seller above). As we will see, this challenge is relevant for both the

$$
\begin{array}{l}
\Lambda B :: \mathsf{Proc}. \\
\quad \lambda\, \mathsf{title} : \mathsf{String} \ @\ B. \\
\quad\quad \lambda\, \mathsf{buyAtPrice?} : \mathsf{Int} \ @\ B \to_{\varnothing} \mathsf{Bool} \ @\ B. \\
\quad\quad\quad \mathsf{let}\ x = \mathsf{com}_{B,\mathsf{Seller}}\ \mathsf{title} \\
\quad\quad\quad \mathsf{in\ if}\ \mathsf{found}(\mathsf{price\_lookup}\ x\ ) \\
\quad\quad\quad\quad \mathsf{then\ select}_{\mathsf{Seller},B}\ \mathsf{Continue} \\
\quad\quad\quad\quad\quad\quad \mathsf{select}_{\mathsf{Seller},\mathsf{Seller}_2}\ \mathsf{Disconnect} \\
\quad\quad\quad\quad\quad\quad \mathsf{let}\ y = \mathsf{com}_{\mathsf{Seller},B}\ (\mathsf{price}(\mathsf{price\_lookup}\ x)\ ) \\
\quad\quad\quad\quad\quad\quad \mathsf{in\ if}\ \mathsf{buyAtPrice?}\ y \\
\quad\quad\quad\quad\quad\quad\quad \mathsf{then\ select}_{B,\mathsf{Seller}}\ \mathsf{Buy}\ \ (()\ @\ B) \\
\quad\quad\quad\quad\quad\quad\quad \mathsf{else\ select}_{B,\mathsf{Seller}}\ \mathsf{Quit}\ \ (()\ @\ B) \\
\quad\quad\quad\quad \mathsf{else\ select}_{\mathsf{Seller},B}\ \mathsf{Delegate} \\
\quad\quad\quad\quad\quad\quad \mathsf{select}_{\mathsf{Seller},\mathsf{Seller}_2}\ \mathsf{Delegate} \\
\quad\quad\quad\quad\quad\quad \mathsf{let}\ F = \mathsf{com}_{\mathsf{Seller},\mathsf{Seller}_2} \left(\begin{array}{l}
\lambda\, \mathsf{title}_2 : \mathsf{String}\ @\ \mathsf{Seller}. \\
\quad \mathsf{if}\ \mathsf{found}(\mathsf{price\_lookup}_2\ \mathsf{title}_2\ ) \\
\quad \mathsf{then\ select}_{\mathsf{Seller},B}\ \mathsf{Continue} \\
\quad\quad\quad \mathsf{let}\ y' = \mathsf{price}(\mathsf{price\_lookup}_2\ \mathsf{title}_2) \\
\quad\quad\quad \mathsf{in\ let}\ y = \mathsf{com}_{\mathsf{Seller},B}\ y' \\
\quad\quad\quad\quad \mathsf{in\ if}\ \mathsf{buyAtPrice?}\ y \\
\quad\quad\quad\quad\quad \mathsf{then\ select}_{B,\mathsf{Seller}}\ \mathsf{Buy}\ \ (()\ @\ B) \\
\quad\quad\quad\quad\quad \mathsf{else\ select}_{B,\mathsf{Seller}}\ \mathsf{Quit}\ \ (()\ @\ B) \\
\quad \mathsf{else\ select}_{\mathsf{Seller},B}\ \mathsf{Quit}\ \ (()\ @\ B)
\end{array} \right) \\
\quad\quad\quad\quad\quad\quad \mathsf{in\ let}\ \mathsf{title}_2 = \mathsf{com}_{\mathsf{Seller},\mathsf{Seller}_2}\ x \\
\quad\quad\quad\quad\quad\quad\quad \mathsf{in}\ F\ \mathsf{title}_2
\end{array}
\tag{1.5}
$$

Fig. 1. Example of Delegation

type system and projection operation of PolyChorλ. For typing, the combination of process polymorphism and distributed value communication can make it difficult to statically determine where data is located. For projection, we need to ensure that the third-party processes involved in a communicated value perform the required changes to process names in the right places during execution.

Structure of the Paper. We begin in Section 2 by examining the system model of PolyChorλ. We then proceed with the following contributions:

- In Section 3, we describe the PolyChorλ language in detail. This language includes both type polymorphism and process polymorphism. We develop both a type system and kind system and an operational semantics for PolyChorλ.
- In Section 4, we describe the local *network language* used to describe the distributed implementation. We also detail how to obtain this implementation via *endpoint projection*, which compiles PolyChorλ programs to a program for each process.
- In Section 5, we describe the main theorem of this paper, the correctness of endpoint projection with respect to our operational semantics. Because of the dynamic nature

of process polymorphism, this requires significant reasoning compared to previous works on choreographies.

- In Section 6, we demonstrate how our theory can be used to model an extended example where an edge computer can delegate tasks to an external server.

Finally, we discuss related work in Section 7 and conclude in Section 8.

2 System Model

We begin by discussing the assumptions we make about how PolyChorλ programs will be run. These assumptions are as light as possible, allowing for PolyChorλ to be run in many different scenarios. In particular, we assume that we have a fixed set of processes, which can communicate via messages. These processes can each be described by a polymorphic λ-calculus, similar to System Fω, but with the addition of communication primitives.

2.1 Processes

We assume that there is a fixed set N of process names P, Q, Alice, et cetera. These processes can represent nodes in a distributed system, system processes, threads, or more. Process polymorphism allows us to refer to processes using type variables, which may go in or out of scope. Despite this, the set of physically-running processes remains the same.

We assume every process knows its identity. Thus, every process can choose what code to run on the basis of its identity. This assumption is reasonable for many practical settings, for instance it is common for nodes in distributed systems to know their identity. This capability is essential to our strategy for enabling process polymorphism.

2.2 Communication

We assume that processes communicate via synchronous message passing. Thus, if P sends a message to Q, then P does not continue until Q has received the message. Moreover, we assume that message passing is instantaneous and certain, so messages do not get lost.

Processes can receive two kinds of messages: values of local programs (described below) and *labels* describing choices made during a computation. These are used to ensure that different processes stay in lock-step with each other.

2.3 Local Programs

We assume that processes run a *local* language described in Section 4. This is a functional language extended with communication features, similar to the language GV (Gay and Vasconcelos, 2010; Wadler, 2012; Lindley and Morris, 2015). Even more related to our work is FST (System F with Session Types) Lindley and Morris (2017), an extension of GV with polymorphism. As it does not have our communication of distributed values, they can base their types on System F rather that System Fω.

Variables	$x, y, \ldots$		
Type Variables	$X, Y, \ldots$		
Integers	n		
Labels	ℓ		
Process Names	P		
Process-Name Sets	ρ	$\in$	$2^{\text{Type Values}}$
Kinds	K	$::=$	$* \mid \mathsf{K}_1 \Rightarrow \mathsf{K}_2 \mid \mathsf{Proc} \mid \mathsf{K} \setminus \rho$
Types	τ	$::=$	$\nu \mid \tau_1\ \tau_2 \mid \tau_1 \to_\rho \tau_2$
		$\mid$	$\tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \mid \forall X :: \mathsf{K}.\ \tau \mid \lambda X :: \mathsf{K}.\ \tau$
Type Values	ν	$::=$	$X \mid () \mathbin{@} \nu \mid \mathsf{Int} \mathbin{@} \nu \mid \nu_1 \to_\rho \nu_2 \mid \mathsf{P}$
		$\mid$	$\nu_1 + \nu_2 \mid \nu_1 \times \nu_2 \mid \forall X :: \mathsf{K}.\ \nu \mid \lambda X :: \mathsf{K}.\ \nu$
Expressions	$M, N, \ldots$	$::=$	$x \mid () \mathbin{@} \nu \mid n \mathbin{@} \nu \mid \lambda\, x : \tau.\, M \mid \Lambda\, X :: \mathsf{K}.\, M$
		$\mid$	$M\ N \mid M\ \tau \mid \mathsf{inl}_\tau\ M \mid \mathsf{inr}_\tau\ M$
		$\mid$	$\mathsf{case}\ M\ \mathsf{of\ inl}\ \ x \Rightarrow N_1;\ \mathsf{inr}\ \ y \Rightarrow N_2$
		$\mid$	$(M, N) \mid \mathsf{fst}\ M \mid \mathsf{snd}\ M$
		$\mid$	$\mathsf{com}^{\tau}_{\nu_1,\nu_2} \mid \mathsf{select}_{\nu_1,\nu_2}\ \ell\ M \mid f$
Values	V	$::=$	$x \mid () \mathbin{@} \nu \mid n \mathbin{@} \nu \mid \lambda\, x : \tau.\, M \mid \Lambda\, x :: \mathsf{K}.\, M$
		$\mid$	$\mathsf{inl}_\tau\ V \mid \mathsf{inr}_\tau\ V \mid (V_1, V_2)$
		$\mid$	$\mathsf{com}^{\tau}_{\nu_1,\nu_2}$

Fig. 2. PolyChorλ Syntax

Endpoint projection translates PolyChorλ into this "Network Process" language. We have thus further extended GV with features required for our endpoint-projection mechanism. For instance, in the local language described in Section 4 we provide an AmI expression form, which allows a process to choose which code to run based on its identity. Despite these extensions, the language should feel familiar to any reader familiar with polymorphic λ-calculi.

3 The Polymorphic Chorλ Language

We now turn to our first major contribution: the design of the polymorphic, choreographic λ-calculus, PolyChorλ. This calculus extends the choreographic λ-calculus Chorλ of Cruz-Filipe et al. (2022) with both type and, more importantly, process polymorphism. We begin by describing the features that PolyChorλ shares with the base Chorλ before describing the new features. The syntax of PolyChorλ can be found in Figure 2.

Syntax Inherited from Chorλ. Since choreographic programs describe the behavior of an entire communicating network of processes, we need to reason about where terms are located. In other words, we need to know which processes store the data denoted by a term. Terms of base type, like integers, are stored by exactly one process. This is represented in our type system by matching base types with a process name. For example, integers stored by the process Alice are represented by the type $\mathsf{Int} \mathbin{@} \mathsf{Alice}$. Values of this type also mark the process which stores them, so a value $5 \mathbin{@} \mathsf{Alice}$ (read "the integer 5 at Alice") has type $\mathsf{Int} @ \mathsf{Alice}$. In Figure 2, the only base types are $() @ \mathsf{P}$ and $\mathsf{Int} @ \mathsf{P}$, but it is easy to extend the language with other base types, such as the types $\mathsf{String} @ \mathsf{P}$ or $\mathsf{Bool} @ \mathsf{P}$ used in the introduction. We will continue to freely use other base types in our examples.

While base types are located on just one process, data of more-complex types may involve multiple processes. For instance, the term $(5 @ \mathsf{Alice}, 42 @ \mathsf{Bob})$ involves both data stored by Alice and Bob. This is still recorded in the type: the term above has type $\mathsf{Int} @ \mathsf{Alice} \times \mathsf{Int} @ \mathsf{Bob}$. In addition to base types and product types, PolyChorλ also has sum types (written $\tau_1 + \tau_2$), along with their normal introduction and elimination forms. Note that products and coproducts in PolyChorλ may not represent a product or coproduct at the local level, since each component may be at a different process. For instance, we can represent distributed booleans as $\mathsf{Bool} @ \mathsf{Alice} \times \mathsf{Bool} @ \mathsf{Bob} + \mathsf{Bool} @ \mathsf{Alice} \times \mathsf{Bool} @ \mathsf{Bob}$. Matching on a value with this type will cause both Alice and Bob to make the same choice.

Functions are treated more unusually: while we have standard λ and application forms, we also allow functions to be defined mutually-recursively with each other. In order to do so, any PolyChorλ choreography is associated with a list, D, of bindings of functions to *function variables* f, which are also expressions. A function variable can then during execution be instantiated with its definition according to this list. As we will see in Section 3.3, PolyChorλ terms are evaluated in a context which associates each function variable with a term. Note that, while in the original Chorλ types were mutually recursive in a similar way, in PolyChorλ we do not support recursive types. To see why, note that we syntactically restrict many types to *type values*. This prevents us having to reason about processes denoted by arbitrary terms—e.g., we cannot send to the "process" $(\lambda X :: \mathsf{Proc}. X)\ \mathsf{P}$ but we can write $(\Lambda Y :: \mathsf{Proc}. \mathsf{com}^{\tau}_{\mathsf{Q},Y})\ ((\lambda X :: \mathsf{Proc}. X)\ \mathsf{P})$ which, due to our call-by-value semantics, will force the type to reduce to P before Y gets instantiated. As we will see in Section 4, allowing communication between arbitrary types would make endpoint projection difficult. However, since recursive types cannot necessarily reduce to a type value, they cannot be used in many parts of the type system.

Function types are also more specific than their usual construction in λ-calculus: they are written $\tau_1 \to_\rho \tau_2$. Here, ρ is a set of process names and type variables denoting additional participants in the function which do not have either the input or output. Thus, if Alice wants to communicate an integer to Bob directly (without intermediaries), then she should use a function of type $\mathsf{Int} @ \mathsf{Alice} \to_\varnothing \mathsf{Int} @ \mathsf{Bob}$. However, if she is willing to use the process Proxy as an intermediary, then she should use a function of type $\mathsf{Int} @ \mathsf{Alice} \to_{\{\mathsf{Proxy}\}} \mathsf{Int} @ \mathsf{Bob}$. We will use ρ when projecting to determine that the function in question and any uses thereof must be part of the local code of Proxy.

In order to allow values to be communicated between processes, we provide the primitive communication function $\mathsf{com}^{\tau}_{\mathsf{P},\mathsf{Q}}$. This function takes a value of type τ at P and returns the corresponding value at Q. As mentioned in the introduction, most choreographic languages provide a communication term modelled after the "Alice-and-Bob" notation of cryptographic protocols. For instance, $\mathsf{Alice} \texttt{->} \mathsf{Bob}: 5$ might represent Alice sending 5 to Bob. This is easily recovered by applying the function $\mathsf{com}^{\tau}_{\mathsf{Alice},\mathsf{Bob}}$. For example, the term $\mathsf{com}^{\mathsf{Int} @ \mathsf{Alice}}_{\mathsf{Alice},\mathsf{Bob}}\ (5 @ \mathsf{Alice})$ represents Alice sending a message containing 5 to Bob: it evaluates to $5 @ \mathsf{Bob}$ and has type $\mathsf{Int} @ \mathsf{Bob}$.

Finally, consider the following, where M has type $\mathsf{Int} @ \mathsf{Alice} + \mathsf{Int} @ \mathsf{Alice}$:

$$\begin{array}{l}\mathsf{case}\ M\ \mathsf{of}\\ \quad \mathsf{inl}\ \ x \Rightarrow 3 @ \mathsf{Bob};\\ \quad \mathsf{inr}\ \ y \Rightarrow 4 @ \mathsf{Bob}\end{array}$$

Clearly, Bob needs to know which branch is taken, since he needs to store a different return value in each branch. However, only Alice knows which whether M evaluates to $\mathsf{inr}_{\mathsf{Int@Alice}}\ V$ or $\mathsf{inl}_{\mathsf{Int@Alice}}\ V$ (here inl and inr are used to denote that a value is either the right or left part of a sum and annotated with the type of the other part of the sum to ensure type principality). Thus, this choreography cannot correspond to any network program. Using the terminology found in the literature of choreographic languages, we might say that the choreography is *unrealisable* because there is insufficient *knowledge of choice* (Castagna et al., 2012; Montesi, 2023).

In order to enable programs where a process's behaviour differs depending on other processes data, such as how Bob behaved differently depending on Alice's data, we provide select terms. These allow one process to tell another which branch has been taken, preventing knowledge from "appearing out of nowhere." For instance, we can extend the program above to:

$$\begin{array}{l}\mathsf{case}\ M\ \mathsf{of}\\ \quad \mathsf{inl}\ \ x \Rightarrow \mathsf{select}_{\mathsf{Alice},\mathsf{Bob}}\ \mathsf{Left}\ (3 @ \mathsf{Bob});\\ \quad \mathsf{inr}\ \ y \Rightarrow \mathsf{select}_{\mathsf{Alice},\mathsf{Bob}}\ \mathsf{Right}\ (4 @ \mathsf{Bob})\end{array}$$

This represents the same program as above, except Alice tells Bob whether the left or the right branch has been taken. Unlike the previous version of this example, it *does* represent a (deadlock-free) network program. In general, we allow arbitrary labels to be sent by select terms, so semantically-meaningful labels can be chosen.

While com and select both transfer information between two processes, they differ in what information they transfer. com moves a value, e.g., as an integer or a function, from the sender to the receiver. select on the other hand uses a label to inform the receiver of a choice made by the sender. Some choreographic languages combine the two, so both a label and a value is communicated at the same time, but like most choreographic languages PolyChorλ keeps the two separate.

Syntax Additions over Chorλ. In order to achieve (both type and process) polymorphism in PolyChorλ, we add several features based on System Fω (Girard, 1972). In particular, we add kinds and universal types $\forall X :: \mathsf{K}.\ \tau$ along with type abstraction and application. From System Fω we inherit the kind $*$, which is the kind of types. We additionally inherit the kind $\mathsf{K}_1 \Rightarrow \mathsf{K}_2$ which represents functions from types to types.

Moreover, we inherit type-level functions $\lambda X :: \mathsf{K}.\ \tau$ from System Fω. These represent the definition of type constructors. We also have type-level function application $\tau_1\ \tau_2$. Since types contain computation, we also define type *values*, which are simply types without application.

We use type-level functions for two primary purposes. First, we can use it to denote types which depend on process names, such as $\lambda X :: \mathsf{Proc}.\ \mathsf{Int} @ X$ and $\lambda X :: \mathsf{Proc} \Rightarrow *.\ X\ \mathsf{P}$. Second, we use type level functions to type communications, as we will see in Section 3.1.

Note that the base types $() \mathbin{@} v$ and $\mathsf{Int} \mathbin{@} v$, like local values, are *syntactically* restricted to only allow type values as subterms. This allows us to use a type variable to compute the location of a value dynamically, but not arbitrary terms, which would make it much harder to tell at time of projection where the value is located. Thus, we can write $(\lambda X :: \mathsf{Proc}.\, \mathsf{Int} \mathbin{@} X)\ (Y\ \mathsf{P})$ to compute the location of an integer dynamically ($Y\ \mathsf{P}$ has to reduce to a type value before X can be instantiated), but we cannot write $\mathsf{Int} \mathbin{@} (Y\ \mathsf{P})$ directly. This way, our projected calculus can tell when instantiating X (at runtime) whether it gets instantiated as P. It would be more complicated to create runtime checks for whether Y gets instantiated as a function type that outputs P or not.

In addition to the kinds $*$ and $\mathsf{K}_1 \Rightarrow \mathsf{K}_2$ of System Fω, we also have the kind Proc of *process names*. Thus, process names are types, but they cannot be used to type any terms.

Additionally, we have *Without kinds* $\mathsf{K} \setminus \rho$, which represents types of kind K which do not mention any of the processes in the set ρ. We also refer to this kind as having a restriction of the processes in ρ. Since we restrict the types that can be communicated based on which processes they contain, as we will see soon, the Without kind can be used to define polymorphic functions which contain communication. For instance, the term

$$\Lambda X :: \mathsf{Proc}.\, \Lambda Y :: \mathsf{Proc} \setminus \{X\}.\, \mathsf{com}_{X,Y}^{\mathsf{Int} @ X}\ (5 \mathbin{@} X)$$

defines a function which, given *distinct* processes X and Y, causes X to send 5 to Y. As we will see in Section 3.2, restricting the processes involved in a type (and therefore the term being typed) is essential for typing communications. In particular, we need to ensure that a sender never tries to send something located at the receiver. Moreover, we need to ensure that every part of the communicated value located at the sender actually gets moved to the receiver, even if its location is an uninstantiated type variable.

In the rest of this section, we explore the semantics of PolyChorλ. First, we look at its static semantics, both in the form of typing and kinding. Second, we describe its operational semantics. Throughout, we will continue to give intuitions based on the concurrent interpretation of PolyChorλ, though the semantics we give here does not correspond directly to that interpretation.

3.1 Typing

We now turn to the type system for PolyChorλ. As before, our type system builds on that for Chorλ. Here, we focus on the rules that are new in this work. Thus, we focus on rules related to polymorphism, and those that have had to change due to polymorphism.

Typing judgements for PolyChorλ have the form $\Theta; \Gamma \vdash M : \tau$, where Θ is the set of process names—either names in N or type variables with kind Proc—used in M or the type of M. The *typing environment* Γ is a list associating variables and function names to their types and type variables and process names to their kinds. We sometimes refer to the pair $\Theta; \Gamma$ as a *typing context*.

Selected rules for our type system can be found in Figure 3. The full collection of rules are given in Appendix 1. Again, many of the rules are inherited directly from Chorλ (Cruz-Filipe et al., 2022); we thus focus on the rules that have changed due to our additions. Many, if not most, of these rules are inspired by System Fω. However, the addition of the kind of processes and Without kinds—i.e., kinds of the form $\mathsf{K} \setminus \rho$—also lead to some changes.

$$[\textsc{Tunit}]\ \frac{\Theta;\Gamma \vdash v :: \mathsf{Proc}}{\Theta;\Gamma \vdash ()\ @\ v : ()\ @\ v} \qquad [\textsc{Tint}]\ \frac{\Theta;\Gamma \vdash v :: \mathsf{Proc}}{\Theta;\Gamma \vdash n\ @\ v : \mathsf{Int}\ @\ v}$$

$$[\textsc{Tapp}]\ \frac{\Theta;\Gamma \vdash N : \tau_1 \to_\rho \tau_2 \qquad \Theta;\Gamma \vdash M : \tau_1}{\Theta;\Gamma \vdash N\ M : \tau_2}$$

$$[\textsc{Tabs}]\ \frac{\begin{array}{c}\Theta;\Gamma \vdash \tau_1 :: * \qquad \Theta;\Gamma' \vdash v :: \mathsf{Proc} \text{ for all } v \in \rho \\ \Theta \cap (\rho \cup \mathsf{ip}(\tau_1) \cup \mathsf{ip}(\tau_2) \cup \mathrm{ftv}(\tau_1) \cup \mathrm{ftv}(\tau_2)); \Gamma, x : \tau_1 \vdash M : \tau_2\end{array}}{\Theta;\Gamma \vdash \lambda\, x : \tau_1.\, M : \tau_1 \to_\rho \tau_2}$$

$$[\textsc{Tsel}]\ \frac{\Theta;\Gamma \vdash v_1 :: \mathsf{Proc} \qquad \Theta;\Gamma \vdash v_2 :: \mathsf{Proc} \qquad \Theta;\Gamma \vdash M : \tau}{\Theta;\Gamma \vdash \mathsf{select}_{v_1,v_2}\ \ell\ M : \tau}$$

$$[\textsc{Tcom}]\ \frac{\begin{array}{c}\Theta;\Gamma \vdash \tau :: \mathsf{Proc} \Rightarrow * \\ \Theta;\Gamma \vdash v_1 :: \mathsf{Proc} \setminus (\mathsf{mp}(\tau) \cup \mathrm{ftv}(\tau)) \qquad \Theta;\Gamma \vdash v_2 :: \mathsf{Proc} \setminus (\mathsf{mp}(\tau) \cup \mathrm{ftv}(\tau))\end{array}}{\Theta;\Gamma \vdash \mathsf{com}^{\tau}_{v_1,v_2} : (\tau\ v_1 \to_\varnothing \tau\ v_2)}$$

$$[\textsc{TappT}]\ \frac{\Theta;\Gamma \vdash M : \forall X :: \mathsf{K}.\, \tau_1 \qquad \Theta;\Gamma \vdash \tau_2 :: \mathsf{K}}{\Theta;\Gamma \vdash M\ \tau_2 : \tau_1[X \mapsto \tau_2]}$$

$$[\textsc{TabsT1}]\ \frac{\Theta, X; \Gamma + X\ \&\ \rho \setminus \{X\}, X :: \mathsf{Proc} \setminus \rho \vdash M : \tau}{\Theta;\Gamma \vdash \Lambda X :: \mathsf{Proc} \setminus \rho.\, M : \forall X :: \mathsf{Proc} \setminus \rho.\, \tau}$$

$$[\textsc{TabsT2}]\ \frac{\Theta, X; \Gamma + X, X :: \mathsf{Proc} \vdash M : \tau}{\Theta;\Gamma \vdash \Lambda X :: \mathsf{Proc}.\, M : \forall X :: \mathsf{Proc}.\, \tau}$$

$$[\textsc{TabsT3}]\ \frac{\Theta;\Gamma + X\ \&\ \rho \setminus \{X\}, X :: \mathsf{K} \setminus \rho \vdash M : \tau \qquad \mathsf{K} \neq \mathsf{Proc}}{\Theta;\Gamma \vdash \Lambda X :: \mathsf{K} \setminus \rho.\, M : \forall X :: \mathsf{K} \setminus \rho.\, \tau}$$

$$[\textsc{TabsT4}]\ \frac{\Theta, X; \Gamma + X, X :: \mathsf{K} \vdash M : \tau \qquad \mathsf{K} \neq \mathsf{Proc} \qquad \nexists \mathsf{K}', \rho.\, \mathsf{K} = \mathsf{K}' \setminus \rho}{\Theta;\Gamma \vdash \Lambda X :: \mathsf{K}.\, M : \forall X :: \mathsf{K}.\, \tau}$$

$$[\textsc{Teq}]\ \frac{\Theta;\Gamma \vdash M : \tau_1 \qquad \tau_1 \equiv \tau_2 \qquad \Theta;\Gamma \vdash \tau_2 :: *}{\Theta;\Gamma \vdash M : \tau_2}$$

Fig. 3. Typing Rules (Selected)

The rules [Tunit] and [Tint] give types to values of base types. Here, we have to ensure that the location of the term is a process. Intuitively, then, we want the location to have kind Proc. However, it might be a Without kind—that is, it might be of the form $\mathsf{Proc} \setminus \rho$. In this case, our subkinding system (which you can find details about in Section 3.2) still allows us to apply the rule.

We express function application and abstraction via the [Tapp] and [Tabs] rules, respectively. The application rule [Tapp] is largely standard—the only addition is the addition of a set ρ on the function type, as discussed earlier. The abstraction rule [Tabs], on the other hand, is more complicated. First, it ensures that the argument type, τ_1, has kind $*$. Then, it ensures that every element in the set decorating the arrow is a process name—i.e., that it has kind Proc. Finally, it checks that, in an extended environment, the body of the function has the output type τ_2. As is usual, this extended environment gives a type to the argument. However, it restricts the available process names to those in the set ρ and those mentioned in the types τ_1 and τ_2.

There are two ways that a type τ can mention a process: it can either name it directly, or it can name it via a type variable. Thus, in the rule [Tabs] we allow the free variables of τ_1 and τ_2 to remain in the process context, computing them using the (standard) free-type-variable function where $\forall X :: \mathsf{K}.\, M$ and $\lambda X :: \mathsf{K}.\, M$ both bind X. However, we must also identify the *involved processes* in a type, which we write $\mathsf{ip}(\tau)$ and compute as follows:

$$\mathsf{ip}(X) = \varnothing \qquad \mathsf{ip}(\mathsf{P}) = \mathsf{P} \qquad \mathsf{ip}(() \mathbin{@} \nu) = \mathsf{ip}(\mathsf{Int} \mathbin{@} \nu) = \mathsf{ip}(\nu)$$

$$\mathsf{ip}(\nu_1 \to_\rho \nu_2) = \mathsf{ip}(\nu_1) \cup \{\mathsf{P} \mid \mathsf{P} \in \rho\} \cup \mathsf{ip}(\nu_2)$$

$$\mathsf{ip}(\forall X :: \mathsf{K} \setminus \rho.\, \tau) = \mathsf{ip}(\lambda X :: \mathsf{K} \setminus \rho.\, \tau) = \mathsf{ip}(\tau) \cup (\mathbf{N} \setminus \rho)$$

$$\mathsf{ip}(\forall X :: \mathsf{K}.\, \tau) = \mathsf{ip}(\lambda X :: \mathsf{K}.\, \tau) = \mathbf{N} \text{ if } \nexists \mathsf{K}', \rho.\, \mathsf{K} = \mathsf{K}' \setminus \rho$$

The involved processes of other types are defined homomorphically.

The communication primitives select and com are typed with [Tsel] and [Tcom], respectively. A term $\mathsf{select}_{\nu_1,\nu_2}\ \ell\ M$ behaves as M, where the process ν_1 informs the process ν_2 that the ℓ branch has been taken, as we saw earlier. Thus, the entire term has type τ if M does. Moreover, ν_1 and ν_2 must be processes.

The rule [Tcom] types com terms. So far we have been simplifying the type used in $\mathsf{com}^{\tau}_{\mathsf{P},\mathsf{Q}}$ for readability. We have been using τ to denote the input type, but as it turns out to type $\mathsf{com}^{\tau}_{\mathsf{P},\mathsf{Q}}$ correctly, we have to complicate things a little. Intuitively, a term $\mathsf{com}^{\tau}_{\nu_1,\nu_2}\ M$ represents ν_1 communicating the parts of M on ν_1 to ν_2. Thus, we require that τ be a type *transformer* requiring a process. Moreover, ν_1 and ν_2 cannot be mentioned in τ; otherwise not every part of the type of M on ν_1 in our example above would transfer to ν_2. For this we use the following notion of *mentioned processes*:

$$\mathsf{mp}(X) = \varnothing \qquad \mathsf{mp}(\mathsf{P}) = \mathsf{P} \qquad \mathsf{mp}(() \mathbin{@} \nu) = \mathsf{mp}(\mathsf{Int} \mathbin{@} \nu) = \mathsf{mp}(\nu)$$

$$\mathsf{mp}(\nu_1 \to_\rho \nu_2) = \mathsf{mp}(\nu_1) \cup \{\mathsf{P} \mid \mathsf{P} \in \rho\} \cup \mathsf{mp}(\nu_2)$$

$$\mathsf{mp}(\forall X :: \mathsf{K} \setminus \rho.\, \tau) = \mathsf{mp}(\lambda X :: \mathsf{K} \setminus \rho.\, \tau) = \mathsf{mp}(\tau) \cup \rho$$

$$\mathsf{mp}(\forall X :: \mathsf{K}.\, \tau) = \mathsf{mp}(\lambda X :: \mathsf{K}.\, \tau) = \mathsf{mp}(\tau) \text{ if } \nexists \mathsf{K}', \rho.\, \mathsf{K} = \mathsf{K}' \setminus \rho$$

Again, with other types being defined homomorphically. The difference between involved and mentioned processes is subtle. If there is no polymorphism, they are the same, but when dealing with polymorphism with restriction they are opposites: involved processes includes every process not in the restriction (the variable could be instantiated as something involving those processes and thus they may be involved), while mentioned names includes the processes mentioned in the restriction. Mentioned names is used only when typing com. If we have such a type-level function, τ, and two type values v_1 and v_2 which are not and will not be instantiated to anything mentioned in τ then we can type $\mathsf{com}^{\tau}_{v_1,v_2}$ as a function from $\tau\ v_1$ to $\tau\ v_2$. Since this is direct communication, no intermediaries are necessary and we can associate this arrow with the empty set $\varnothing$.

It is worth noting at this point that the communication rule inspired our use of System Fω rather than plain System F, which lacks type-level computation. In Chorλ and other previous choreographic languages, communicated values must be local to the sender. In PolyChorλ, this would mean not allowing the communicated type to include type variables or processes other than the sender. Since we are introducing the idea of using communication as a means of delegation, we have slackened that restriction. This means that PolyChorλ programs can communicate larger choreographies whose type may involve other processes, and importantly other type variables. We see this in the delegation example Listing (1.5), where we have the communication $\mathsf{com}_{\mathsf{Seller},\mathsf{Seller}_2}$. Adding in the required type annotation (which we had suppressed in the introduction), this becomes $\mathsf{com}^{\lambda X::\mathsf{Proc}.\,\mathsf{String}@X\to_{\varnothing}()@B}_{\mathsf{Seller},\mathsf{Seller}_2}$. Note that this still leaves us with a free type variable B, representing the unknown process that Seller is telling Seller_2 to interact with! Since we cannot ban free type variables in communicated types, we must create a typing system that can handle them, and this requires type level computation.

To see why this led us to type-level computation, consider the alternative. In Chorλ and other choreographic works, we would have a type communication using process *substitution* instead of communication. The annotated program would then be $\mathsf{com}^{\mathsf{String}@\mathsf{Seller}\to_{\varnothing}()@B}_{\mathsf{Seller},\mathsf{Seller}_2}$. When applied to a program of appropriate type, the result would have type

$$(\mathsf{String}\ @\ \mathsf{Seller} \to_{\varnothing} ()\ @\ B)[\mathsf{Seller} \mapsto \mathsf{Seller}_2] = \mathsf{String}\ @\ \mathsf{Seller}_2 \to_{\varnothing} ()\ @\ B$$

Note that, because B is a type variable, it was ignored by the substitution. If B is later instantiated as Seller, then we must substitute B with Seller_2 in the output type. Thus, we need some mechanism to delay this substitution; rather than use a mechanism like explicit substitutions, we instead reached for the standard tool of System Fω. The communication winds up instead being written as $\mathsf{com}^{\lambda X::\mathsf{Proc}.\,\mathsf{String}@X\to_{\varnothing}()@B}_{\mathsf{Seller},\mathsf{Seller}_2}$ with X being instantiated as Seller in the input type and Seller_2 in the output type. This seemed more elegant and less ad-hoc; moreover, it adds features which a real-world implementation of PolyChorλ would want anyway. To ensure that B does not get instantiated incorrectly, we use our Without kinds. Rule [Tcom] requires that both Seller and Seller_2 are restricted on B, which, thanks to our restrictions being symmetric, means that B cannot be instantiated as either of them. The Without kinds here prevent nonsensical typings of com where in the type, part of the output does not get moved from the sender to the receiver. This can happen if a type variable present in the type of the communicated value will in the execution of the choreography get instantiated before the communication takes place, but has not yet been instantiated when we type the choreography. Were it not for the restrictions imposed by Without kinds, we would allow the choreography

$$(\Lambda B :: \mathsf{Proc}.\ \lambda f : \mathsf{String}\ @\ \mathsf{Seller} \to_{\varnothing} ()\ @\ B.\ (\mathsf{com}^{\lambda X::\mathsf{Proc}.\,\mathsf{String}@X\to_{\varnothing}()@B}_{\mathsf{Seller},\mathsf{Seller}_2}\ f))\ \mathsf{Seller}$$

to get typed as $(\mathsf{String}\ @\ \mathsf{Seller} \to_{\varnothing} ()\ @\ \mathsf{Seller}) \to_{\varnothing} (\mathsf{String}\ @\ \mathsf{Seller_2} \to_{\varnothing} ()\ @\ \mathsf{Seller})$, which implies that part of the function is still at Seller after the communication is executed. This is not what will happen when actually executing the choreography, so the type is wrong. The Without kinds ensure that the choreography cannot be typed, as the kind of B must be $\mathsf{Proc} \setminus \{\mathsf{Seller}, \mathsf{Seller_2}\}$, and it therefore cannot be instantiated as Seller.

Returning now to the typing rules of Figure 3, we next have the [TappT], [TabsT1], [TabsT2], [TabsT3] and [TabsT4] rules, which type universal quantification. The [TappT] rule is completely standard, while the others are 4 cases of what to do with a type abstraction. Each of these rules have a different definition for the typing context of M, depending on the kind of X. As is standard, we check if the body of the function has the right type when the parameter X has kind K. But first, if X is a process as in [TabsT1] and [TabsT2], then we need to extend Θ with X. In addition we must further manipulate the context in order to ensure that the types whose kinds are restricted on X correspond to the restriction on the kind of X.

First, the new type variable X may shadow a previously-defined X. Thus, we need to remove X from any Without kinds already in the context. We do this using the following operation $\mathsf{K} + v$:

$$(\mathsf{K} \setminus \rho) + v = (\mathsf{K} + v) \setminus (\rho \setminus \{v\})$$

We define $+$ on other kinds homomorphically, and extend this to contexts as usual:

$$\Gamma + v = \{x : \tau \mid x : \tau \in \Gamma\} \cup \{X : \mathsf{K} + v \mid X : \mathsf{K} \in \Gamma\}$$

Furthermore, in [TabsT1] and [TabsT4] if X itself has a Without kind—that is, X's kind tells us it cannot be any of the processes in ρ—then we need to symmetrically add a restriction on X to every type in ρ. Otherwise, we would not be able to use the roles in ρ in any place where we cannot use X, even though we know X will not be instantiated with them. We do this with the operation $\Gamma \,\&\, \rho \setminus X$, which we define as follows:

$$\begin{aligned} \Gamma \,\&\, \rho \setminus X = \quad & \{x : \tau \mid x : \tau \in \Gamma\} \cup \{\tau :: \mathsf{K} \mid \tau :: \mathsf{K} \in \Gamma \text{ and } \tau \notin \rho\} \\ & \cup \{\tau :: \mathsf{K} \setminus (\rho_2 \cup \{X\}) \mid \tau :: \mathsf{K} \setminus \rho_2 \in \Gamma \text{ and } \tau \in \rho\} \\ & \cup \{\tau :: \mathsf{K} \setminus \{X\} \mid \tau :: \mathsf{K} \in \Gamma,\ \mathsf{K} \neq \mathsf{K_2} \setminus \rho_2, \text{ and } \tau \in \rho\} \end{aligned}$$

With these operations in place, we can now fully understand how to type the type abstractions. When K is actually a Without kind, then we must handle both shadowing and symmetrical restrictions. However, when it is not a Without kind, we must only handle shadowing. We show an example where every possible complication

Example 1 (Typing complex type abstractions)**.** Consider the following choreography, which takes a process A and sends an integer communication with A from P to Q:

$$M = \Lambda A :: \mathsf{Proc} \setminus \{\mathsf{P}, \mathsf{Q}\}.\ \mathsf{com}_{\mathsf{P},\mathsf{Q}}^{\Lambda X :: \mathsf{Proc}.\ \mathsf{Int}@X \to_{\varnothing} \mathsf{Int}@A}\ \mathsf{com}_{\mathsf{P},A}^{\Lambda Y :: \mathsf{Proc}.\ \mathsf{Int}@Y}$$

That A has a Without kind and the fact that A is a process means that we will need to use Rule [TabsT1] when typing M. In order to illustrate the necessity of shadowing, we will include an unnecessary process $\mathsf{P_2}$ in our environment. Setting $\Theta = \{\mathsf{P}, \mathsf{Q}, \mathsf{P_2}\}$, we start with the following judgment:

$$\Theta; \mathsf{P} : \mathsf{Proc}, \mathsf{Q} : \mathsf{Proc}, \mathsf{P}_2 : \mathsf{Proc} \setminus \{A\} \vdash M : \forall A :: \mathsf{Proc} \setminus \{\mathsf{P}, \mathsf{Q}\}. \mathsf{Int} \mathbin{@} \mathsf{Q} \to_{\varnothing} \mathsf{Int} \mathbin{@} A$$

We need to take into account both that A is a process and that it has a Without kind in order to make the choreography typeable. First, we shadow, obtaining the following:

$$(\mathsf{P} : \mathsf{Proc}, \mathsf{Q} : \mathsf{Proc}, \mathsf{P}_2 : \mathsf{Proc} \setminus \{A\}) + A = \mathsf{P} : \mathsf{Proc}, \mathsf{Q} : \mathsf{Proc}, \mathsf{P}_2 : \mathsf{Proc}$$

so we get rid of any restrictions on previous variables called A. We then add the new symmetric restrictions necessary for typing the communication, as follows:

$$(\mathsf{P} : \mathsf{Proc}, \mathsf{Q} : \mathsf{Proc}, \mathsf{P}_2 : \mathsf{Proc}) \mathbin{\&} \{\mathsf{P}, \mathsf{Q}\} \setminus A = \mathsf{P} : \mathsf{Proc} \setminus \{A\}, \mathsf{Q} : \mathsf{Proc} \setminus \{A\}, \mathsf{P}_2 : \mathsf{Proc}$$

Continuing on, we can abbreviate $\mathsf{K} = \mathsf{Proc} \setminus \{A\}$. Finally, we add A to the environment and Θ (writing $\Theta' = \Theta \cup \{A\}$), giving:

$$\Theta'; \mathsf{P} : \mathsf{K}, \mathsf{Q} : \mathsf{K}, \mathsf{P}_2 : \mathsf{Proc}, A : \mathsf{Proc} \setminus \{\mathsf{P}, \mathsf{Q}\} \vdash N : \mathsf{Int} \mathbin{@} \mathsf{Q} \to_{\varnothing} \mathsf{Int} \mathbin{@} A$$

where $M = \Lambda A :: \mathsf{Proc} \setminus \{\mathsf{P}, \mathsf{Q}\}. N$. Because of the restrictions in Rule [Tcom], N would not be typable if we had not made sure to add the symmetric restrictions. We will furthermore see in Section 3.2 that adding A to the set process names is also necessary when kinding it with the Proc kind.

On the other hand, although the rule looks bigger at first glance, it is much simpler to use Rule [TabsT4].

Example 2 (Typing simple type abstractions)**.** Consider the following type abstraction, which takes a type A and applies a variable of that type to a function which also returns something of the same type:

$$\Lambda A :: *. \lambda x : A. \lambda f : A \to_{\varnothing} A. f\ x$$

We can type this as

$$\emptyset; \emptyset \vdash \Lambda A :: *. \lambda x : A. \lambda f : A \to_{\varnothing} A. f\ x : \forall A :: *. A \to_{\varnothing} A \to_{\varnothing} A \to_{\varnothing} A$$

Since we have no shadowing, the only way we have to manipulate our environment when entering the type abstraction is to add $A : *$ to the environment, giving us

$$\emptyset; A : * \vdash \lambda x : A. \lambda f : A \to_{\varnothing} A. f\ x : A \to_{\varnothing} A \to_{\varnothing} A \to_{\varnothing} A$$

Rules [TabsT2] and [TabsT3] are for cases of middling complexity. In Rule [TabsT2], we have to add the type variable to Θ, as in [TabsT1]. However, since we have no restrictions, we do not need to consider symmetric conflict. In Rule [TabsT3], we do consider symmetric conflicts, but do not add to Θ (since we are not dealing with a process).

The final addition to our type system is the rule [Teq]. This is another standard rule from System Fω; it tells us that we are allowed to compute in types. More specifically, it tells us that we can replace a type with an equivalent type, using the following equivalence:

$$\frac{}{\tau \equiv \tau} \qquad \frac{\tau_1 \equiv \tau_2}{\tau_2 \equiv \tau_1} \qquad \frac{\tau_1 \equiv \tau_2 \quad \tau_2 \equiv \tau_3}{\tau_1 \equiv \tau_3}$$

$$\frac{\tau_1 \equiv \tau_1{}' \quad \tau_2 \equiv \tau_2{}'}{\tau_1 \to_\rho \tau_2 \equiv \tau_1{}' \to_\rho \tau_2{}'} \qquad \frac{\tau_1 \equiv \tau_1{}' \quad \tau_2 \equiv \tau_2{}'}{\tau_1 + \tau_2 \equiv \tau_1{}' + \tau_2{}'} \qquad \frac{\tau_1 \equiv \tau_1{}' \quad \tau_2 \equiv \tau_2{}'}{\tau_1 \times \tau_2 \equiv \tau_1{}' \times \tau_2{}'}$$

$$\frac{\tau \equiv \tau'}{\lambda X :: \mathsf{K}.\ \tau \equiv \lambda X :: \mathsf{K}.\ \tau'} \qquad \frac{}{(\lambda X :: \mathsf{K}.\ \tau_1)\ \tau_2 \equiv \tau_1[X \mapsto \tau_2]} \qquad \frac{\tau_1 \equiv \tau_1{}' \quad \tau_2 \equiv \tau_2{}'}{\tau_1\ \tau_2 \equiv \tau_1{}'\ \tau_2{}'}$$

$$\frac{\tau \equiv \tau'}{\forall X :: \mathsf{K}.\ \tau \equiv \forall X :: \mathsf{K}.\ \tau'}$$

In addition to the rules in Figure 3 for typing choreographies, our type system needs one more rule for typing the definitions of our recursive functions. We also add an extra judgement of the form $\Theta;\Gamma \vdash D$ where $\Theta;\Gamma$ is a typing context as before, and D is a set of definitions for function variables—i.e., $D = \{f_1 = M_1, \ldots f_n = M_n\}$. We write $D(f)$ for the term associated with f in D. The only rule for this judgement is [Tdefs], which says that a set of definitions is well-formed if every variable in D is associated with a type τ in Γ, and the body of f in D can be given be given type τ in the context $\emptyset;\Gamma$. We require that the body of f can be typed with an empty set of roles because they are global predefined functions, and as such they should not be local to any one process.

$$[\textsc{Tdefs}]\ \frac{\forall f \in \mathsf{domain}(D).\ f : \tau \in \Gamma \wedge \emptyset;\Gamma \vdash D(f) : \tau}{\Theta;\Gamma \vdash D}$$

3.2 Kinding

We finish our discussion of the static semantics of PolyChorλ by looking at our kinding system. Our kinding system uses only one judgement, $\Theta;\Gamma \vdash \tau :: \mathsf{K}$, which says that in the typing context $\Theta;\Gamma$, the type τ has kind K. You can find the rules of our kinding system in Figure 4. These are mostly directly inherited from System Fω. However, we must account for Proc and Without kinds.

For instance, the rules [Kunit] and [Kint] check that the type representing which process is storing the data indeed has the kind Proc. Similarly, [Kfun] ensures that all of the types in the set of possible intermediaries are processes. The rule for type variables, [Kvar], ensures that if a type variable X is assigned kind Proc, then X must also be in Θ.

One of the biggest differences between our kinding system and that of System Fω, however, is the rule [Ksub] which tells us that our system enjoys *subkinding*. The subkinding rules come from the subset ordering on Without kinds. We also consider any kind equivalent to the same kind restricted on the empty set due to [SKEmpty] and [SKWithoutL].

$$[\textsc{KVar}]\ \frac{X :: \mathsf{K} \in \Gamma \qquad \text{if } \mathsf{K} \in \{\mathsf{Proc}, \mathsf{Proc} \setminus \rho\} \text{ then } X \in \Theta}{\Theta; \Gamma \vdash X :: \mathsf{K}}$$

$$[\textsc{KRole}]\ \frac{\mathsf{K} \in \{\mathsf{Proc}, \mathsf{Proc} \setminus \rho\} \qquad \begin{array}{c}\mathsf{P} :: \mathsf{K} \in \Gamma \\ \mathsf{P} \in \Theta\end{array} \qquad \text{if } \mathsf{K} = \mathsf{Proc} \setminus \rho \text{ then } \mathsf{P} \notin \rho}{\Theta; \Gamma \vdash \mathsf{P} :: \mathsf{K}}$$

$$[\textsc{KUnit}]\ \frac{\Theta; \Gamma \vdash \tau :: \mathsf{Proc} \setminus \rho}{\Theta; \Gamma \vdash () \mathbin{@} \tau :: * \setminus \rho} \qquad [\textsc{KInt}]\ \frac{\Theta; \Gamma \vdash \tau :: \mathsf{Proc} \setminus \rho}{\Theta; \Gamma \vdash \mathsf{Int} \mathbin{@} \tau :: * \setminus \rho}$$

$$[\textsc{KFun}]\ \frac{\Theta; \Gamma \vdash \tau_1 :: * \setminus \rho_2 \qquad \Theta; \Gamma \vdash \tau_2 :: * \setminus \rho_2 \qquad \forall \nu \in \rho_1 .\, \Theta; \Gamma \vdash \nu :: \mathsf{Proc} \setminus \rho_2}{\Theta; \Gamma \vdash \tau_1 \rightarrow_{\rho_1} \tau_2 :: * \setminus \rho_2}$$

$$[\textsc{KAbs}]\ \frac{\Theta; \Gamma, X :: \mathsf{K}_1 \vdash \tau :: \mathsf{K}_2}{\Theta; \Gamma \vdash \lambda X :: \mathsf{K}_1 .\, \tau :: \mathsf{K}_1 \Rightarrow \mathsf{K}_2} \qquad [\textsc{KAll}]\ \frac{\Theta; \Gamma, X :: \mathsf{K} \vdash \tau :: * \setminus \rho}{\Theta; \Gamma \vdash \forall X :: \mathsf{K} .\, \tau :: * \setminus \rho}$$

$$[\textsc{KArr}]\ \frac{\Theta; \Gamma \vdash \tau :: (\mathsf{K}_1 \setminus \rho) \Rightarrow (\mathsf{K}_2 \setminus \rho)}{\Theta; \Gamma \vdash \tau :: (\mathsf{K}_1 \Rightarrow \mathsf{K}_2) \setminus \rho} \qquad [\textsc{KSub}]\ \frac{\Theta; \Gamma \vdash \tau :: \mathsf{K}_1 \qquad \mathsf{K}_1 <: \mathsf{K}_2}{\Theta; \Gamma \vdash \tau :: \mathsf{K}_2}$$

$$[\textsc{KSum}]\ \frac{\Theta; \Gamma \vdash \tau_1 :: * \setminus \rho \qquad \Theta; \Gamma \vdash \tau_2 :: * \setminus \rho}{\Theta; \Gamma \vdash \tau_1 + \tau_2 :: * \setminus \rho}$$

$$[\textsc{KProd}]\ \frac{\Theta; \Gamma \vdash \tau_1 :: * \setminus \rho \qquad \Theta; \Gamma \vdash \tau_2 :: * \setminus \rho}{\Theta; \Gamma \vdash \tau_1 \times \tau_2 :: * \setminus \rho}$$

Fig. 4. Kinding Rules

The rules for subkinding are as follows:

$$[\textsc{SKRefl}]\ \frac{}{\mathsf{K} <: \mathsf{K}} \qquad [\textsc{SKTrans}]\ \frac{\mathsf{K}_1 <: \mathsf{K}_2 \qquad \mathsf{K}_2 <: \mathsf{K}_3}{\mathsf{K}_1 <: \mathsf{K}_3}$$

$$[\textsc{SKArr}]\ \frac{\mathsf{K}'_1 <: \mathsf{K}_1 \qquad \mathsf{K}_2 <: \mathsf{K}'_2}{\mathsf{K}_1 \Rightarrow \mathsf{K}_2 <: \mathsf{K}'_1 \Rightarrow \mathsf{K}'_2} \qquad [\textsc{SKEmpty}]\ \frac{}{\mathsf{K} <: \mathsf{K} \setminus \varnothing}$$

$$[\textsc{SKWithoutL}]\ \frac{\mathsf{K}_1 <: \mathsf{K}_2}{\mathsf{K}_1 \setminus \rho <: \mathsf{K}_2} \qquad [\textsc{SKWithoutUnion}]\ \frac{\mathsf{K}_1 <: \mathsf{K}_2}{\mathsf{K}_1 \setminus (\rho_1 \cup \rho_2) <: \mathsf{K}_2 \setminus \rho_1}$$

Lemma 1. *Let τ be a type. If there exists a typing context $\Theta; \Gamma$ such that $\Theta; \Gamma \vdash \tau :: K$ then there exists a unique type value ν such that $\tau \equiv \nu$.*

Proof The existence of ν follows from induction on $\Theta; \Gamma \vdash \tau :: \mathsf{K}$ and its uniqueness from induction on $\tau \equiv \nu$. ■

$$\text{[AppTAbs]}\ \frac{\tau \equiv v}{(\Lambda X :: \mathsf{K}.\ M)\ \tau \rightarrow_D M[X \mapsto v]} \qquad \text{[MTApp1]}\ \frac{M_1 \rightarrow_D M_2}{M_1\ \tau \rightarrow_D M_2\ \tau}$$

$$\text{[Def]}\ \frac{}{f \rightarrow_D D(f)}$$

$$\text{[Sel]}\ \frac{}{\mathsf{select}_{\mathsf{P},\mathsf{Q}}\ \ell\ M \rightarrow_D M} \qquad \text{[Com]}\ \frac{}{\mathsf{com}^{\tau}_{\mathsf{P},\mathsf{Q}}\ V \rightarrow_D V[\mathsf{P} \mapsto \mathsf{Q}]}$$

Fig. 5. Semantics of PolyChorλ (Selected Rules)

Lemma 2 (Type restriction)**.** *Let τ be a type. If there exists a typing context $\Theta;\Gamma$ such that $\Theta;\Gamma \vdash \tau :: K \setminus \rho$ then $(ip(\tau) \cup ftv(\tau)) \cap \rho = \emptyset$.*

Proof Follows from kinding rules. ∎

Theorem 1 (Kindable types)**.** *Let M be a choreography and τ be a type such that $\Theta;\Gamma \vdash M : \tau$. Then $\Theta;\Gamma \vdash \tau :: *$.*

Proof Follows from induction on the derivation of $\Theta;\Gamma \vdash M : \tau$ and the kinding rules. ∎

We also find that types have the same kinds as their equivalent type values. Due to β-expansion, a kindable type can be equivalent to an unkindable type, but not an unkindable type value.

Theorem 2 (Kind Preservation)**.** *Let τ be a type. If there exists a typing context $\Theta;\Gamma$ such that $\Theta;\Gamma \vdash \tau :: K$, then $\Theta;\Gamma \vdash v :: K$ for any type value v such that $\tau \equiv v$.*

Proof Follows from the kinding and type equivalence rules. The only way that a kindable type τ can be equivalent to a type which is not kindable is when we have types τ_1 and τ_2 such that $\tau = \tau_1[X \mapsto \tau_2]$. In that case, if we use the rule $(\lambda X :: \mathsf{K}.\ \tau_1)\ \tau_2 \equiv \tau_1[X \mapsto \tau_2]$ to create an unkindable $\tau' \equiv \tau$ with an extra application. However, this unkindable type is not a type value, and in fact we must also use the same rule to remove this new type application before we get to a type value. ∎

Example 3. We return to the delegation example (Listing (1.5)) and try to type it. As B appears free in the type of a value, F, being communicated between Seller and Seller_2, B must actually have the Without kind $\mathsf{Proc} \setminus \{\mathsf{Seller}, \mathsf{Seller}_2\}$. The choreography therefore gets the type

$$\forall B :: \mathsf{Proc} \setminus \{\mathsf{Seller}, \mathsf{Seller}_2\}.$$
$$\mathsf{String} @ B \rightarrow_{\{\mathsf{Seller},\mathsf{Seller}_2\}} ((\mathsf{Int} @ B \rightarrow_{\emptyset} \mathsf{Bool} @ B) \rightarrow_{\{\mathsf{Seller},\mathsf{Seller}_2\}} () @ B)$$

This type shows both the input, output, and involved roles of the choreography.

3.3 Operational Semantics

Finally, we consider the operational semantics of PolyChorλ. In practice, the semantics of a choreographic language can be used to simulate a choreography and check if it specifies the expected collective behaviour. Its key role, however, is to prove properties about the projected local code. Specifically, we are going to prove that the projected code is compliant to the choreography (an operational correspondence result) and that as a result it is deadlock-free. The semantics of PolyChorλ are mostly a standard call-by-value reduction semantics for a typed λ calculus. However, the reduction semantics must also carry a set D of function definitions. Only a few rules are unusual or must be modified; those can be found in Figure 5. You can find the rest of the rules in Appendix 2.

The rules [AppTAbs] and [MTApp1] come from System Fω. The rule [AppTAbs] is similar to ordinary CBV β reduction, but tells us how to reduce a *type* abstraction applied to a *type* value, but with the caveat that if we do not have a type value we must use type equivalence to get one before reducing. The rule [MTApp1] tells us that we can reduce a type function applied to any argument.

The rule [Def] allows us to reduce function names by looking up their definition in the set D.

Finally, we have the rules for communication. The rule [Sel] says that select acts as a no-op, as we stated earlier. While this may seem redundant, such terms are vital for projection, as we will see in the next section. More importantly, the [Com] rule tells us how we represent communication at the choreography level: via substitution of roles. This also helps explain some of the restrictions in [Tcom]. Since we replace *all* mentions of P with Q in V, we cannot allow other mentions of P in the type transformer of V. Otherwise, there could be some mentions of P which should not be replaced during communication, which we do not model. Unlike when typing $\mathsf{com}^{\tau}_{\mathsf{P},\mathsf{Q}}\ V$, when executing a communication we know (since we only consider choreographies without free variables) that any type variables in τ or V have already been instantiated and as such do we do not need to consider how to substitute variables which may later be instantiated to P or Q.

It may be surprising to learn that our semantics are simply call-by-value reduction semantics, especially for those readers familiar with choreographies. After all, choreographies are supposed to represent concurrent programs, and so multiple redices should be available at any time. Indeed, previous works on choreographic programming (e.g. Hirsch and Garg, 2022; Cruz-Filipe and Montesi, 2020; Carbone and Montesi, 2013) provided a semantics with *out-of-order execution*, so that the operational semantics of the choreographies matched with all possible reductions in the concurrent interpretation. We use these simpler semantics, without out-of-order execution, instead. In exchange, our result in Section 5 will be weaker: we only promise that any value which the choreography can reduce to, so can the concurrent interpretation.

To see why we chose to obtain this weaker result, consider the choreography

$$f\ ((\mathsf{com}^{\lambda X::\mathsf{Proc}.\ \mathsf{Int}@X}_{\mathsf{Q}_1,\mathsf{Q}_2}\ (3 \mathbin{@} \mathsf{Q}_1)), (4 \mathbin{@} \mathsf{P}))$$

Here we have a function f which needs to be instantiated with a distributed pair. P is ready to feed its part of the argument into f and start computing the result, while Q_1 and Q_2 are still working on computing their part of the argument. There are two ways we could interpret PolyChorλ concurrently: we can synchronize when all processes enter a function *or* we can allow P to enter the function early. We take the second, more practical, route. However, this means it is not possible to reflect at least one evaluation order into the semantics of the choreography without banning distributed values or allowing us to somehow call a single value in multiple steps. This insight led to us adopting the weaker guarantee discussed above.

As is standard for call-by-value λ-calculi, we are able to show that our type system is *sound* with respect to our operational semantics, as expressed in the following two theorems:

Theorem 3 (Type Preservation)**.** *Let M be a choreography and D a function mapping containing every function in M. If there exists a typing context $\Theta;\Gamma$ such that $\Theta;\Gamma \vdash M : \tau$ and $\Theta;\Gamma \vdash D$, then $\Theta;\Gamma \vdash M' : \tau$ for any M' such that $M \rightarrow_D M'$.*

Proof Follows from the typing and semantic rules and Theorem 2. ■

Theorem 4 (Progress)**.** *Let M be a closed choreography and D a function mapping containing every function in M. If there exists a typing context $\Theta;\Gamma$ such that $\Theta;\Gamma \vdash M : \tau$ and $\Theta;\Gamma \vdash D$, then either $M = V$ or there exists M' such that $M \rightarrow_D M'$.*

Proof Follows from the typing and semantic rules. ■

4 Endpoint Projection

We now proceed to the most important result for any choreographic programming language: *endpoint projection*. Endpoint projection gives a concurrent interpretation to our language PolyChorλ by translating it to a parallel composition of programs, one for each process. In order to define endpoint projection, though, we must define our process language, which we refer to as a *local* language. The syntax of the local language can be found in Figure 6. There you can also find the syntax of local transition labels and network transition labels, both of which will be described when we describe the operational semantics of networks.

As in PolyChorλ, our local language inherits much of its structure from System Fω. In particular, we have products, sums, functions, universal quantification, and λ types, along with their corresponding terms. In fact, some types look more like standard System Fω than PolyChorλ: function types do not need a set of processes which may participate in the function, and base types no longer need a location.

However, not everything is familiar; we have introduced new terms and new types. The terms send_v and recv_v allow terms to send and receive values, respectively. We also split select terms into two terms: an *offer* term $\&_v\ \{\ell_1 : L_1, \ldots, \ell_n : L_n\}$ which allows v to choose how this term will evolve. We represent such choices using *choice* terms of the form $\oplus_v\ \ell\ L$. This term informs the process represented by v that it should reduce to its subterm labeled by ℓ, and then itself reduces to the term L. While these are unusual pieces of a polymorphic language like System Fω, they are familiar from process languages like π calculus. We also add undefined types and terms, written $\perp$ and $\perp$, respectively. These

Variables	$x, y, \ldots$		
Type Variables	$X, Y, \ldots$		
Process Names	P		
Local Transition Labels	μ	$::=$	$\tau \mid \mathsf{P} \mid \mathsf{send}_{\mathsf{P}}\ L\ L' \mid \mathsf{recv}_{\mathsf{P}}\ L'\ L$
		$\mid$	$\oplus_{\mathsf{P}}\ \ell \mid \&_{\mathsf{P}}\ \ell$
Network Transition Labels	μ	$::=$	$\tau_{\mathscr{P}}$
Process labels	$\mathscr{P}$	$::=$	$\mathsf{P} \mid \mathsf{P}, \mathsf{Q}$
Local Types	t	$::=$	$v \mid t_1\ t_2 \mid \mathsf{AmI}\ v\ ?\ t_1\ \&\ t_2 \mid t_1 \to t_2$
		$\mid$	$t_1 + t_2 \mid t_1 \times t_2 \mid \forall X.\, t \mid \lambda X.\, t$
Local Type Values	v	$::=$	$X \mid () \mid \mathsf{Int} \mid v_1 \to v_2 \mid \mathsf{P} \mid \bot$
		$\mid$	$v_1 + v_2 \mid v_1 \times v_2 \mid \forall X.\, v \mid \lambda X.\, v$
Local Expressions	B	$::=$	$x \mid () \mid n \mid \lambda\, x : t.\, B \mid \Lambda X.\, B$
		$\mid$	$B_1\ B_2 \mid B\ t \mid \mathsf{inl}_t\ B \mid \mathsf{inr}_t\ B$
		$\mid$	$\mathsf{case}\ B\ \mathsf{of}\ \mathsf{inl}\ \ x \Rightarrow B_1;\ \mathsf{inr}\ \ y \Rightarrow B_2$
		$\mid$	$(B_1, B_2) \mid \mathsf{fst}\ B \mid \mathsf{snd}\ B$
		$\mid$	$\mathsf{send}_v \mid \mathsf{recv}_v$
		$\mid$	$\&_v\ \{\ell_1 : B_1, \ldots, \ell_n : B_n\} \mid \oplus_v\ \ell\ B$
		$\mid$	$\mathsf{sub}[v_1 \mapsto v_2] \mid f \mid \mathsf{AmI}\ v\ ?\ B_1\ \&\ B_2$
Local Values	L	$::=$	$x \mid () \mid n \mid \bot \mid \lambda\, x : t.\, B \mid \Lambda X.\, B$
		$\mid$	$\mathsf{inl}_t\ L \mid \mathsf{inr}_t\ L \mid (L_1, L_2)$
		$\mid$	$\mathsf{send}_v \mid \mathsf{recv}_v \mid \mathsf{sub}[v_1 \mapsto v_2]$

Fig. 6. Local Language Syntax

represent terms which are ill-defined; we use them to represent data which does not exist on some process P, but which needs to be written structurally in P's program. For instance, $\bot$ is the result of sending a value without process polymorphism. We also use it as the input of recv, since both send and recv are functions which require an input. More generally, if a process P participates in a function but the input and/or output is located elsewhere, we will use $\bot$ to represent that input and/or output. The type $\bot$ is only used for the term $\bot$.

We also include a more unusual feature: *explicit substitutions* of processes. The term $\mathsf{sub}[v_1 \mapsto v_2]$ is a function which, when applied, replaces the role denoted by v_1 with that denoted by v_2 in its argument. This function allows us to represent the view of communication according to third parties: the roles simply change, without any mechanism necessary. For instance, imagine that Alice wants to tell Bob to communicate an integer to Cathy. She can do this by sending Bob the function $\mathsf{com}^{\lambda X::\mathsf{Proc}.\ \mathsf{Int}@X}_{\mathsf{Alice},\mathsf{Cathy}}$. In PolyChor$\lambda$, this corresponds to the choreography

$$\mathsf{com}^{\lambda X::\mathsf{Proc}.\ \mathsf{Int}@X \to_{\varnothing} \mathsf{Int}@\mathsf{Cathy}}_{\mathsf{Alice},\mathsf{Bob}}\ \left(\mathsf{com}^{\lambda X::\mathsf{Proc}.\ \mathsf{Int}@X}_{\mathsf{Alice},\mathsf{Cathy}}\right)$$

In order to project this choreography, we need to be able to project the communication function above even when it is not applied to any arguments. This is where we use explicit substitutions: we project the communication function to $\mathsf{sub}[\mathsf{Alice} \mapsto \mathsf{Bob}]$.

Finally, we introduce our unique feature: AmI terms and their corresponding type. These represent the ability of a process to know its own identity, and to take actions based on that knowledge. Process polymorphism requires an instantiation of a process variable at process P to be accompanied by a conditional determining whether the variable has been instantiated as P or as some other process P may interact with. In particular, the term $\mathsf{AmI}\ v\,?\,B_1\ \&\ B_2$ reduces to B_1 if the term is run by the process denoted by v, and B_2 otherwise. Since B_1 and B_2 may have different types, we provide types of the form $\mathsf{AmI}\ v\,?\,t_1\ \&\ t_2$, which represent either the type t_1 (if typing a term on the process denoted by v) or t_2 (otherwise). These terms form a backbone of endpoint projection for PolyChorλ: every Λ term binding a process gets translated to include an AmI term. For instance, consider projecting the choreography

$$\Lambda X :: \mathsf{Proc}.\ \mathsf{com}_{\mathsf{Q},X}^{\lambda X' :: \mathsf{Proc}.\ \mathsf{Int} @ X'}\ 4 @ \mathsf{Q}$$

to some process P. Depending on the argument to which this function is applied, P should behave very differently: if it is applied to P itself, it should receive something from Q. However, if it's applied to any other term, it should do nothing. We therefore project the choreography above to the following program for P:

$$\Lambda X.\ \mathsf{AmI}\ X\ ?\ \mathsf{recv}_{\mathsf{Q}}\ \bot\ \&\ \bot$$

Note that the AmI construct is necessary for process polymorphism in general, unless process variables cannot be instantiated to the process they are located at. It, and the combinatorial explosion caused by having multiple process abstractions, is not caused by the choreographic language but instead the choreographic language hides it and lets programmers avoid explicitly describing both sides of the AmI separately.

Note that we do not have a kinding system for local programs. In fact, we do not check the types of local programs at all. However, because types have *computational* content, we need to project them as well. In order to preserve that computational content, we again use an equivalence of types which corresponds to β,η-equivalence. However, in order to accommodate AmI types, we must index that equivalence with a process. Then, we have two rules regarding AmI types:

$$[\textsc{IAm}]\ \frac{}{\mathsf{AmI}\ \mathsf{P}\ ?\ t_1\ \&\ t_2 \equiv_{\mathsf{P}} t_1} \qquad [\textsc{IAmNot}]\ \frac{\mathsf{P} \neq \mathsf{Q}}{\mathsf{AmI}\ \mathsf{Q}\ ?\ t_1\ \&\ t_2 \equiv_{\mathsf{P}} t_2}$$

We use these equivalence rules with process annotation to ensure that processes only use equivalences indexed with their own name and do not pick the wrong branch of an AmI type. This way we project the type $(\lambda X :: \mathsf{Proc}.\ \mathsf{Int} @ X)\ \mathsf{P}$ as $(\lambda X.\ \mathsf{AmI}\ X\ ?\ \mathsf{Int}\ \&\ \bot)\ \mathsf{P}$ which is equivalent to Int and P but $\bot$ everywhere else.

Now that we have seen the syntax of the programs which run on each process, we can look at whole networks:

Definition 1. A network $\mathcal{N}$ is a finite map from a set of processes to local programs. We often write $\mathsf{P}_1[L_1] \mid \cdots \mid \mathsf{P}_n[L_n]$ for the network where process P_i has behaviour L_i.

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, as suggested by the syntax above.

We now consider the operational semantics of local programs and networks. These are given via labelled-transition systems; the syntax of both sorts of label can be found in Figure 6. The network transitions are labelled with $\tau_{\mathscr{P}}$ where $\mathscr{P}$ is the set of involved processes (either one for a local action or two for a synchronisation). The local transitions have more options for labels. The label τ denotes a normal local computation. We use the process name P as a label for an action which can only take place at P. The label $\mathsf{send}_{\mathsf{P}}\ L\ L'$ denotes sending the value L to P, leaving L' after the send—we will explain what a label left behind after the send does when we discuss the semantics of local communication in detail. The label $\mathsf{recv}_{\mathsf{P}}\ L'\ L$ is the dual: it denotes receiving L' from P, with L being the value the receiver had before receiving. Again, we explain the semantics of receiving in detail later. Finally, the label $\oplus_{\mathsf{P}}\ \ell$ denotes sending a label ℓ to P, while the label $\&_{\mathsf{P}}\ \ell$ denotes receiving the label ℓ from P.

Selected rules for both operational semantics can be found in Figures 7 and 8. As before, transitions are indexed by a set d of function definitions. Function variables reduce by looking up their definition in d. Since this transition involves no communication, it is labelled with the empty transition, τ.

Perhaps surprisingly, undefined arguments to functions do not immediately cause the application to be undefined. To see why, think about choreographies of the form $(\lambda\, x : \mathsf{Int} @ \mathsf{P}.\, M)\ N$ where some process $\mathsf{Q} \neq \mathsf{P}$ is involved in both M and N. We project this to an application on Q of the form $(\lambda\, x : \bot.\ [\![M]\!]_{\mathsf{Q}})\ [\![N]\!]_{\mathsf{Q}}$. Note that because we know that N has type $\mathsf{Int} @ \mathsf{P}$, the projection $[\![N]\!]_{\mathsf{Q}}$ has type $\bot$ and eventually evaluates to $\bot$. Thus, if $(\lambda\, x : \bot.\ [\![M]\!]_{\mathsf{Q}})\ \bot$ immediately evaluated to $\bot$, the process Q could not participate in M, as they need to do! We therefore allow this to evaluate to $[\![M]\!]_{\mathsf{Q}}$. However, when the function is also undefined, we evaluate this to $\bot$ with the empty label τ, as you can see in the rules [NBot] and [NBott]

As mentioned earlier, the explicit substitutions $\mathtt{sub}[\mathsf{P} \mapsto \mathsf{Q}]$ are functions which, when applied, perform the requested substitution in the value to which they are applied. This is implemented in the rule [NSub].

The AmI terms are given meaning via the rules [NAmIR] and [NAmIL]. The rule [NAmIR] says that the term $\mathsf{AmI}\ \mathsf{P}\ ?\, L_1 \,\&\, L_2$ can evaluate to L_1 with label P, while the rule [NAmIL] says that it can instead reduce to L_2 with label Q where $\mathsf{Q} \neq \mathsf{P}$. We will see later that in the network semantics, we only allow transitions labeled with the process performing the transition.

Choice and offer terms reduce via the rules [NCho] and [Noff]. The first, [Ncho], tells us that a choice term simply reduces to its continuation with a transition label indicating the choice that has been made. The second, [Noff], tells us that an offer term can reduce to *any* continuation, with a transition label indicating the label of the continuation it reduced to. We will see later that the semantics of networks only allows the offer term to reduce to the continuation chosen by a matching choice term.

Finally, the send and recv terms are given meaning via [NSend] and [NRecv], respectively. However, these rules behave somewhat-differently than might be expected: rather than acting as a plain send and receive, they behave more like a swap of information.

In a plain send, the sender would not have any information after the send—perhaps the term would come with a continuation, but this would not be related to the send. Moreover, the receiver would not provide any information, but merely receive the information from

$$[\textsc{NDef}]\ f \xrightarrow{\tau}_d d(f) \qquad [\textsc{NAbsApp}]\ (\lambda\, x : t.\, B)\ L \xrightarrow{\tau}_d B[x \mapsto L]$$

$$[\textsc{NBAbs}]\ \frac{t \equiv_{\mathsf{P}} v}{(\Lambda X.\, B)\ t \xrightarrow{\mathsf{P}}_d B[X \mapsto v]} \qquad [\textsc{NBot}]\ \bot\ \bot \xrightarrow{\tau}_d \bot$$

$$[\textsc{NSub}]\ \mathsf{sub}[\mathsf{P} \mapsto \mathsf{Q}]\ L \xrightarrow{\tau}_d L[\mathsf{P} \mapsto \mathsf{Q}] \qquad [\textsc{NBotT}]\ \bot\ \bot \xrightarrow{\tau}_d \bot$$

$$[\textsc{NAmIR}]\ \mathsf{AmI}\ \mathsf{P}\ ?\ L_1\ \&\ L_2 \xrightarrow{\mathsf{P}}_d L_1 \qquad [\textsc{NAmIL}]\ \frac{\mathsf{Q} \neq \mathsf{P}}{\mathsf{AmI}\ \mathsf{P}\ ?\ L_1\ \&\ L_2 \xrightarrow{\mathsf{Q}}_d L_2}$$

$$[\textsc{NCho}]\ \oplus_{\mathsf{P}}\ \ell\ L \xrightarrow{\oplus_{\mathsf{P}}\ \ell}_d L \qquad [\textsc{NOff}]\ \&_{\mathsf{P}}\ \{\ell_1 : L_1, \ldots, \ell_n : L_n\} \xrightarrow{\&_{\mathsf{P}}\ \ell_i}_d L_i$$

$$[\textsc{NSend}]\ \mathsf{send}_{\mathsf{P}}\ L_1 \xrightarrow{\mathsf{send}_{\mathsf{P}}\ L_1\ L_2}_d L_2 \qquad [\textsc{NRecv}]\ \mathsf{recv}_{\mathsf{P}}\ L_1 \xrightarrow{\mathsf{recv}_{\mathsf{P}}\ L_2\ L_1}_d L_2$$

$$[\textsc{NApp1}]\ \frac{B_1 \xrightarrow{\mu}_d B_2}{B_1\ B' \xrightarrow{\mu}_d B_2\ B'} \qquad [\textsc{NApp2}]\ \frac{B \xrightarrow{\mu}_d B'}{L\ B \xrightarrow{\mu}_d L\ B'} \qquad [\textsc{NTApp1}]\ \frac{B \xrightarrow{\mu}_d B'}{B\ t \xrightarrow{\mu}_d B\ t}$$

$$[\textsc{NInL}]\ \frac{B \xrightarrow{\mu}_d B'}{\mathsf{inl}_t\ B \xrightarrow{\mu}_d \mathsf{inl}_t\ B'} \qquad [\textsc{NInR}]\ \frac{B \xrightarrow{\mu}_d B'}{\mathsf{inr}_t\ B \xrightarrow{\mu}_d \mathsf{inr}_t\ B'}$$

$$[\textsc{NCase}]\ \frac{B \xrightarrow{\mu}_d B'}{\mathsf{case}\ B\ \mathsf{of}\ \mathsf{inl}\ \ x \Rightarrow B_1;\ \mathsf{inr}\ \ y \Rightarrow B_2 \xrightarrow{\mu}_d \mathsf{case}\ B'\ \mathsf{of}\ \mathsf{inl}\ \ x \Rightarrow B_1;\ \mathsf{inr}\ \ y \Rightarrow B_2}$$

$$[\textsc{NCaseL}]\ \mathsf{case}\ \mathsf{inl}_t\ L\ \mathsf{of}\ \mathsf{inl}\ \ x \Rightarrow B_1;\ \mathsf{inr}\ \ y \Rightarrow B_2 \xrightarrow{\tau}_d B_1[x \mapsto L]$$

$$[\textsc{NCaseR}]\ \mathsf{case}\ \mathsf{inr}_t\ L\ \mathsf{of}\ \mathsf{inl}\ \ x \Rightarrow B_1;\ \mathsf{inr}\ \ y \Rightarrow B_2 \xrightarrow{\tau}_d B_2[x \mapsto L]$$

$$[\textsc{NPair1}]\ \frac{B_1 \xrightarrow{\mu}_d B_1'}{(B_1, B_2) \xrightarrow{\mu}_d (B_1', B_2)} \qquad [\textsc{NPair2}]\ \frac{B_2 \xrightarrow{\mu}_d B_2'}{(B_1, B_2) \xrightarrow{\mu}_d (B_1, B_2')}$$

$$[\textsc{Fst}]\ \frac{B_1 \rightarrow_D B_2}{\mathsf{fst}\ B_1 \rightarrow_D \mathsf{fst}\ B_2} \qquad [\textsc{Snd}]\ \frac{B_1 \rightarrow_D B_2}{\mathsf{snd}\ B_1 \rightarrow_D \mathsf{snd}\ B_2} \qquad [\textsc{NProj1}]\ \mathsf{fst}\ \ (L_1, L_2) \xrightarrow{\tau}_d L_1$$

$$[\textsc{NProj2}]\ \mathsf{snd}\ \ (L_1, L_2) \xrightarrow{\tau}_d L_2$$

Fig. 7. Semantics of Local Processes

the sender. However, when sending a choreography with process polymorphism, the sender may need to participate in the continuation, depending on how polymorphic functions are applied. For instance, consider the following choreography, where P sends a polymorphic

$$[\textsc{NCom}]\ \frac{L_1 \xrightarrow{\mathsf{send}_{\mathsf{P}}\ L\ (L'[\mathsf{Q}\mapsto\mathsf{P}])}_d L_1' \qquad L_2 \xrightarrow{\mathsf{recv}_{\mathsf{Q}}\ (L[\mathsf{Q}\mapsto\mathsf{P}])\ L'}_d L_2'}{\mathsf{Q}[L_1] \mid \mathsf{P}[L_2] \xrightarrow{\tau_{\mathsf{Q},\mathsf{P}}}_d \mathsf{Q}[L_1'] \mid \mathsf{P}[L_2']}$$

$$[\textsc{NSel}]\ \frac{L_1 \xrightarrow{\oplus_{\mathsf{P}}\ \ell}_d L_1' \qquad L_2 \xrightarrow{\&_{\mathsf{Q}}\ \ell}_d L_2'}{\mathsf{Q}[L_1] \mid \mathsf{P}[L_2] \xrightarrow{\tau_{\mathsf{Q},\mathsf{P}}}_d \mathsf{Q}[L_1'] \mid \mathsf{P}[L_2']}$$

$$[\textsc{NProAm}]\ \frac{L \xrightarrow{\mathsf{P}}_d L'}{\mathsf{P}[L] \xrightarrow{\tau_{\mathsf{P}}}_d \mathsf{P}[L']} \qquad [\textsc{NPro}]\ \frac{L \xrightarrow{\tau}_d L'}{\mathsf{P}[L] \xrightarrow{\tau_{\mathsf{P}}}_d \mathsf{P}[L']}$$

$$[\textsc{NPar}]\ \frac{\mathcal{N}_1 \xrightarrow{\tau_{\mathscr{P}}}_d \mathcal{N}_2}{\mathcal{N}_1 \mid \mathcal{N}' \xrightarrow{\tau_{\mathscr{P}}}_d \mathcal{N}_2 \mid \mathcal{N}'}$$

Fig. 8. Semantics of Networks

function to Q, and the resulting polymorphic function is applied to P:

$$(\mathsf{com}_{\mathsf{P},\mathsf{Q}}^{\lambda Y::\mathsf{Proc}.\,\forall X::\mathsf{Proc}.\,\mathsf{Int}@X}\ (\Lambda X :: \mathsf{Proc}.\ \mathsf{com}_{\mathsf{P},X}^{\lambda Y'::\mathsf{Proc}.\,\mathsf{Int}@Y'}\ (5 @ \mathsf{P})))\ \mathsf{P}$$

The polymorphic function that results from the com above is as follows:

$$\Lambda X :: \mathsf{Proc}.\ \left(\mathsf{com}_{\mathsf{Q},X}^{\lambda Y'::\mathsf{Proc}.\,\mathsf{Int}@Y'}\ (5 @ \mathsf{Q})\right)$$

Applying this to P leads to a program where P receives from Q. Since P needs to participate in this program, P must have a program remaining after sending the polymorphic function to Q.

While this explains why send terms cannot simply, for instance, return unit, it does not explain why send and recv terms *swap* results. To see this, consider what happens when a term is sent from a process P to another process Q. We know from our type system that Q is not mentioned in the type of the term being sent, and we know that after the send all mentions of P are changed to mentions of Q. Hence, after the send, P's version of the term should be the view of a process not involved in the term. This is exactly what Q's version of the term is *before* the send. Thus, sends and recvs behaving as swaps leads to the correct behaviour.

Example 4 (Send And Receive)**.** We now show the local projection (formalised in Section 4.1) and desired behaviour of

$$(\mathsf{com}_{\mathsf{P},\mathsf{Q}}^{\lambda Y::\mathsf{Proc}.\,\forall X::\mathsf{Proc}.\,\mathsf{Int}@X}\ (\Lambda X :: \mathsf{Proc}.\ \mathsf{com}_{\mathsf{P},X}^{\lambda Y'::\mathsf{Proc}.\,\mathsf{Int}@Y'}\ (5 @ \mathsf{P})))\ \mathsf{P}$$

This choreography generates the network:

$$\mathsf{P}[(\mathsf{send}_{\mathsf{Q}}\ (\Lambda X.\ \mathsf{AmI}\ X\ ?\ (\lambda x : \lambda Y'.\ \mathsf{AmI}\ Y'\ ?\ \mathsf{Int}\ \&\ \bot\ \mathsf{P}.\ x)\ \&\ (\mathsf{send}_X\ 5)))\ \mathsf{P}] \mid$$
$$\mathsf{Q}[(\mathsf{recv}_{\mathsf{P}}\ (\Lambda X.\ \mathsf{AmI}\ X\ ?\ (\mathsf{recv}_{\mathsf{P}}\ \bot)\ \&\ (\bot)))\ \mathsf{P}]$$

Using our semantics, we get the following reductions:

$$
\begin{array}{l}
\mathsf{P}[(\mathsf{send}_{\mathsf{Q}}\ (\Lambda X.\ \mathsf{AmI}\ X\ ?\ (\lambda\, x : \lambda Y'.\ \mathsf{AmI}\ Y'\ ?\ \mathsf{Int}\ \&\ \bot\ \mathsf{P}.\, x)\ \&\ (\mathsf{send}_X\ 5)))\ \mathsf{P}]\,\|\\
\mathsf{Q}[(\mathsf{recv}_{\mathsf{P}}\ (\Lambda X.\ \mathsf{AmI}\ X\ ?\ (\mathsf{recv}_{\mathsf{P}}\ \bot)\ \&\ (\bot)))\ \mathsf{P}]\\
\xrightarrow{\tau_{\mathsf{P},\mathsf{Q}}}_{\emptyset}\\
\mathsf{P}[(\Lambda X.\ \mathsf{AmI}\ X\ ?\ (\mathsf{recv}_{\mathsf{Q}}\ \bot)\ \&\ (\bot))\ \mathsf{P}]\,\|\\
\mathsf{Q}[(\Lambda X.\ \mathsf{AmI}\ X\ ?\ (\lambda\, x : \lambda Y'.\ \mathsf{AmI}\ Y'\ ?\ \mathsf{Int}\ \&\ \bot\ \mathsf{Q}.\, x)\ \&\ (\mathsf{send}_X\ 5))\ \mathsf{P}]\\
\xrightarrow{\tau_{\mathsf{P}}}_{\emptyset}\\
\mathsf{P}[(\mathsf{AmI}\ \mathsf{P}\ ?\ (\mathsf{recv}_{\mathsf{Q}}\ \bot)\ \&\ (\bot))]\,\|\\
\mathsf{Q}[(\Lambda X.\ \mathsf{AmI}\ X\ ?\ (\lambda\, x : \lambda Y'.\ \mathsf{AmI}\ Y'\ ?\ \mathsf{Int}\ \&\ \bot\ \mathsf{Q}.\, x)\ \&\ (\mathsf{send}_X\ 5))\ \mathsf{P}]\\
\xrightarrow{\tau_{\mathsf{P}}}_{\emptyset}\\
\mathsf{P}[\mathsf{recv}_{\mathsf{Q}}\ \bot]\,\|\\
\mathsf{Q}[(\Lambda X.\ \mathsf{AmI}\ X\ ?\ (\lambda\, x : \lambda Y'.\ \mathsf{AmI}\ Y'\ ?\ \mathsf{Int}\ \&\ \bot\ \mathsf{Q}.\, x)\ \&\ (\mathsf{send}_X\ 5))\ \mathsf{P}]\\
\xrightarrow{\tau_{\mathsf{Q}}}_{\emptyset}\\
\mathsf{P}[\mathsf{recv}_{\mathsf{Q}}\ \bot]\,\|\\
\mathsf{Q}[\mathsf{AmI}\ \mathsf{P}\ ?\ (\lambda\, x : \lambda Y'.\ \mathsf{AmI}\ Y'\ ?\ \mathsf{Int}\ \&\ \bot\ \mathsf{Q}.\, x)\ \&\ (\mathsf{send}_X\ 5)]\\
\xrightarrow{\tau_{\mathsf{Q}}}_{\emptyset}\\
\mathsf{P}[\mathsf{recv}_{\mathsf{Q}}\ \bot]\,\|\,\mathsf{Q}[(\mathsf{send}_X\ 5)]\\
\xrightarrow{\tau_{\mathsf{Q},\mathsf{P}}}_{\emptyset}\\
\mathsf{P}[5]\,\|\,\mathsf{Q}[\bot]
\end{array}
$$

Now that we have discussed the semantics of local programs, we discuss the semantics of networks. Each transition in the network semantics has a silent label indexed with the processes participating in that reduction: $\tau_{\mathscr{P}}$, where $\mathscr{P}$ consists of either one process name (for local actions at that process) or two process names (for interactions involving these two processes). We treat $\mathscr{P}$ as a set, implicitly allowing for exchange.

For instance, the rule [NCom] describes communication. Here, one local term must reduce with a send label, while another reduces with a recv label. These labels must match, in the sense that the value received by the recv must be the value sent by the send—though with the receiver in place of the sender—and vice-versa. Then, a network in which the local terms are associated with the appropriate processes, Q and P, can reduce with the label $\tau_{\mathsf{Q},\mathsf{P}}$. Similarly, the rules [NSel] reduces matching choice and select terms, resulting in the label $\tau_{\mathsf{Q},\mathsf{P}}$.

While [NCom] and [NSel] describe communication, the rest of the rules describe how a single process's term can evolve over time in a network. Particularly interesting is [NProam], which says that a AmI term can reduce only according to the process it is associated with. We can see here that the resulting label is τ_{P}, indicating that this reduction step only involves P.

The rules [NPro] tells us how to lift steps with an empty label τ. Such steps make no assumptions about the network, and so such terms can be associated with any process P. When such a reduction takes place in a network, we label the resulting transition τ_{P}.

Finally, the rule [NPar] says that if one part of a network can reduce with a label $\tau_{\mathscr{P}}$, then the entire network can reduce with that same label. This allows the other rules, which assume minimal networks, to be applied in larger networks.

In the future we will use $\to^*$ and $\to^+$ to denote respectively a sequence and a sequence of at least one action with arbitrary labels.

4.1 Projection

We can now define the endpoint projection (EPP) of choreographies. This describes a single process's view of the choreography; the concurrent interpretation of a choreography is given by composing the projection to every process in parallel. Endpoint projection to a particular process P is defined as a recursive function over typing derivations $\Theta; \Gamma \vdash M : \tau$. For readability, however, we write it as a recursive function over the term M, and use the notation $\mathsf{typeof}(N)$ to refer to the types assigned to any term N in the implicit typing derivation. Similarly, we use $\mathsf{kindof}(\tau)$ to refer to the kind of a type τ in the implicit typing derivation. We write $[\![M]\!]_{\mathsf{P}}$ to denote the projection of the term M (implicitly a typing derivation for M, proving that it has *some* type) to the process P.

Intuitively, projection translates a choreography term to its corresponding local behavior. For example, a communication action projects to a send (for the sender), a receive (for the receiver), a substitution (for the other processes in the type of the value being communicated) or an empty process (for the remaining processes). However, this is more complicated for case statements. For instance, consider the following choreography, which matches on a sum type which is either an integer on Alice or a unit on Alice. If it is an integer, then Bob receives that integer from Alice and the choreography returns the integer now located at Bob. Otherwise, The choreography returns the default value 42 also located at Bob. Alice informs Bob of which branch she has taken using select terms.

$$\begin{pmatrix} \lambda z : (\mathsf{Int} \mathbin{@} \mathsf{Alice}) + (() \mathbin{@} \mathsf{Alice}). \\ \quad \mathsf{case}\ z\ \mathsf{of} \\ \qquad \mathsf{inl}\ \ x \Rightarrow \mathsf{select}_{\mathsf{Alice},\mathsf{Bob}}\ \mathsf{Just}\ (\mathsf{com}^{\lambda X.\, \mathsf{Int} @ X}_{\mathsf{Alice},\mathsf{Bob}}\ x); \\ \qquad \mathsf{inr}\ \ y \Rightarrow \mathsf{select}_{\mathsf{Alice},\mathsf{Bob}}\ \mathsf{Nothing}\ (42 \mathbin{@} \mathsf{Bob}) \end{pmatrix} \mathsf{inl}_{() @ \mathsf{Alice}}\ (3 \mathbin{@} \mathsf{Alice})$$

Imagine projecting this to Bob's point of view. He does not have any of the information in the sum, so he cannot participate in choosing which branch of the case expression gets evaluated. Instead, he has to wait for Alice to tell him which branch he is in. If we naïvely translate just the first branch of the case expression, Bob waits for Alice to send him the label Just and then waits for Alice to send him an integer. Similarly, in the second branch Bob waits for Alice to send him the label $\mathsf{Nothing}$ before returning the default value 42. Somehow, we need to combine these so that Bob waits for either label, and then takes the corresponding action.

We do this by *merging* Bob's local programs for each branch (Carbone et al., 2012; Cruz-Filipe and Montesi, 2020; Honda et al., 2016). Merging is a *partial* function which combines two compatible local programs, combining choice statements. In other words, the key property of merging is:

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

Merging is defined homomorphically on other terms, though it is undefined on incompatible terms. Thus, for example, $\mathsf{inl}_t\ B \sqcup \mathsf{inl}_t\ B' = \mathsf{inl}_t\ (B \sqcup B')$, but $\mathsf{inl}_{t_1}\ B \sqcup \mathsf{inr}_{t_2}\ B'$ is undefined.

We can then use this to project the choreography above to Bob as:

$$(\lambda\, z : \bot . \&_{\mathsf{Alice}}\ \{\mathsf{Just} : (\mathsf{recv}_{\mathsf{Alice}}\ \bot, \mathsf{Nothing} : 42\})\ \bot$$

Where $\bot$ represents a part of the choreography executed by Alice.

Definition 2. The EPP of a choreography M for process P is defined by the rules in Figures 9, 10 and 11.

To project a network from a choreography, we therefore project the choreography for each process and combine the results in parallel: $[\![M]\!] = \prod_{\mathsf{P} \in \mathsf{ip}(M)} \mathsf{P}\ [[\![M]\!]_{\mathsf{P}}]$.

Intuitively, projecting a choreography to a process that is not involved in it returns a $\bot$. More complex choreographies, though, may involve processes that are not shown in their type. This explains the first clause for projecting an application: even if P does not appear in the type of M, it may participate in interactions 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 other choreography calculi, where self-communications are not allowed—we do not want to impose this in PolyChorλ, since we have process polymorphism and therefore do not want to place unnecessary restrictions on which processes a choreography can be instantiated with.

Any process P must prepare two behaviours for a process abstraction $\Lambda X :: \mathsf{Proc}. M$: one for when X is instantiated with P itself, and one for when X is instantiated with another process. To do this, we use AmI terms, which allow P to use its knowledge of its identity to select which behaviour takes place. (This also holds when X has a Without kind, as long as the base kind is Proc, though if P is excluded from the type of X and P does not participate in M then we simply project to $\bot$.) However, type abstractions $\Lambda X :: \mathsf{K}. M$ do not use AmI terms if K is not a kind of processes, since P cannot instantiate X.

When projecting an application, we may project both the function and its argument, either one but not the other, or neither. While it may seem simple—just project both sides, and get rid of any $\bot$s or $\bot$s that come up—it turns out to be somewhat complicated. In order to ensure every process performs actions in the same order and avoid communication mismatches, we must project an abstraction for any process involved in the computation, even if they do not have the input (Cruz-Filipe et al., 2023, Example 6). To see why this causes complications, consider $M = \lambda\, x : \mathsf{Int} @ \mathsf{P}. 5 @ \mathsf{Q}$. When M gets projected to Q, it becomes $\lambda\, x. 5$. However, applying M to an argument—say, $M\ 2 @ \mathsf{P}$—needs to lead to a function application on Q! Thus, we project this to $(\lambda\, x. 5)\ \bot$, allowing Q to instantiate its function. We use the type system to identify the cases where we need to keep $\bot$ or $\bot$ and those where we should only project the function part of an (type) application.

Type applications work a bit differently. Since there is no chance of communication happening while computing a type, we can project only the body of a type abstraction without the actual abstraction to P when we know the argument is not located at P. In addition, we do not have a case for projecting only the argument, since the context surrounding a type abstraction will not expect a type.

In general, projecting a type yields $\perp$ at any process not used in that type. We use the restrictions on kinds to avoid projecting type variables and type abstractions when we know we do not need to and project all process names to themselves, but otherwise the projection of type constructs is similar to that of corresponding process terms.

Finally, to execute a projected choreography, we need to project the set of definitions of choreographic functions to a set of definitions of local functions. Since these functions are all parametrised over every involved process, this is as simple as projecting the definitions onto an arbitrarily chosen process name.

$$[\![D]\!] = \{f \mapsto [\![D(f)]\!]_{\mathsf{P}} \mid f \in \mathsf{domain}(D)\}$$

Note that function names always get projected everywhere. This means that if we have a function which does not terminate when applied to some value in any process, then it diverges when applied to that value in the choreography and in every other process.

Example 5. We will now show how to project the bookseller service example Eq. (1.3). As in that example we use $\mathsf{let}\ x = B\ \mathsf{in}\ B'$ as syntactic sugar for $\lambda\, x : t.\, B'\ B$ for some t and $\mathsf{if}\ B_1\ \mathsf{then}\ B_2\ \mathsf{else}\ B_3$ as syntactic sugar for $\mathsf{case}\ B_1\ \mathsf{of}\ \mathsf{inl}\ \ x \Rightarrow B_2;\ \mathsf{inr}\ \ x \Rightarrow B_3$ for some $x \notin (\mathrm{fv}(B_2) \cup \mathrm{fv}(B_3))$. We project for Seller and get the following process:

$$\begin{array}{l}
\Lambda B.\\
\quad \mathsf{AmI}\ B\\
\quad\quad ?\,\lambda\,\mathsf{title}.\\
\quad\quad\quad\quad \lambda\,\mathsf{buyAtPrice?}.\\
\quad\quad\quad\quad\quad \mathsf{let}\ x = (\lambda\, y.\, y)\ \mathsf{title}\\
\quad\quad\quad\quad\quad \mathsf{in}\ \mathsf{let}\ y = (\lambda\, z.\, z)\ (\mathsf{price_lookup}\ x)\\
\quad\quad\quad\quad\quad\quad \mathsf{in}\ \mathsf{if}\ \mathsf{buyAtPrice?}\ y\\
\quad\quad\quad\quad\quad\quad\quad \mathsf{then}\ ()\\
\quad\quad\quad\quad\quad\quad\quad \mathsf{else}\ ()\\
\quad\quad \&\ \ \lambda\,\mathsf{title}.\\
\quad\quad\quad\quad\quad \lambda\,\mathsf{buyAtPrice?}.\\
\quad\quad\quad\quad\quad\quad \mathsf{let}\ x = \mathsf{recv}_B\ \perp\\
\quad\quad\quad\quad\quad\quad \mathsf{in}\ \mathsf{let}\ y = \mathsf{send}_B\ (\mathsf{price_lookup}\ x)\\
\quad\quad\quad\quad\quad\quad\quad \mathsf{in}\ \&_B\ \{\mathsf{Buy} : (), \mathsf{Quit} : ()\}
\end{array}$$

Here we can see that if the buyer B turns out to be Seller itself, then all the communications become identity functions, and the seller does not inform itself of its choice. Otherwise, we get a function which, after being instantiated with a buyer, also needs to be instantiated with two $\perp$s representing values existing at B. It then waits for B to send a title, returns the price of this title, and waits for B to decide whether to buy or not. It might seem strange to have a function parametric on two values that are located at B and will therefore here be instantiated with $\perp$s, but this example actually illustrates why when projecting we cannot in cases like this remove the first two λs from the process without causing a deadlock. Consider that $\mathsf{let}\ y =$

send_B ($\mathsf{price\_lookup}\ x$)$\mathsf{in}\ \&_B\ \{\mathsf{Buy}: (), \mathsf{Quit}: ()\}$ is syntactic sugar for $(\lambda\, y.\ \&_B\ \{\mathsf{Buy}: (), \mathsf{Quit}: ()\})\ (\mathsf{send}_B\ (\mathsf{price\_lookup}\ x))$. Here we need to have the abstraction on y even though it gets instantiated as $\bot$ after Seller sends the result of $\mathsf{price\_lookup}\ x$ to B. If instead we only had $(\&_B\ \{\mathsf{Buy}: (), \mathsf{Quit}: ()\})\ (\mathsf{send}_B\ (\mathsf{price\_lookup}\ x))$, then the first part of the application would not be a value, and would be waiting for B to choose between Buy and Quit while B has the abstraction on y and therefore considers the first part of the application a function which must wait to be instantiated. B therefore expects to receive the result of $\mathsf{price\_lookup}\ x$, and we get a deadlock in our system. This is why we never want to project a value to a non-value term, and need to keep any abstractions guarding a part of the choreography involving Seller.

$$[\![x]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![\mathsf{typeof}(x)]\!]_{\mathsf{P}} = \bot \\ x & \text{otherwise} \end{cases} \qquad [\![f]\!]_{\mathsf{P}} = f$$

$$[\![() \ @ \ v]\!]_{\mathsf{P}} = \begin{cases} () & \text{if } [\![v]\!]_{\mathsf{P}} = \mathsf{P} \\ \bot & \text{otherwise} \end{cases} \qquad [\![n \ @ \ v]\!]_{\mathsf{P}} = \begin{cases} n & \text{if } [\![v]\!]_{\mathsf{P}} = \mathsf{P} \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\lambda\, x : \tau .\, M]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \text{ and } [\![\tau]\!]_{\mathsf{P}} = \bot \\ \lambda\, x : [\![\tau]\!]_{\mathsf{P}} .\ [\![M]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![M\ N]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = [\![N]\!]_{\mathsf{P}} = \bot \\ [\![M]\!]_{\mathsf{P}}\ \ [\![N]\!]_{\mathsf{P}} & \text{if } \mathsf{P} \in \mathsf{ip}(\mathsf{typeof}(M)) \\ & \quad \text{or } [\![M]\!]_{\mathsf{P}} \neq \bot \neq [\![N]\!]_{\mathsf{P}} \\ [\![M]\!]_{\mathsf{P}} & \text{if } [\![N]\!]_{\mathsf{P}} = \bot \\ [\![N]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\Lambda X :: \mathsf{K}.\, M]\!]_{\mathsf{P}} = \begin{cases} \Lambda X.\, \mathsf{AmI}\ X & \text{if } \mathsf{K} \in \{\mathsf{Proc},\ \mathsf{Proc} \setminus \rho\} \\ \qquad ?\ [\![M[X \mapsto \mathsf{P}]]\!]_{\mathsf{P}} & \\ \qquad \&\ \ [\![M]\!]_{\mathsf{P}} & \\ [\![M]\!]_{\mathsf{P}} & \text{if } \mathsf{K} = \mathsf{K}' \setminus \{\mathsf{P}\} \cup \rho \\ \Lambda X.\ [\![M]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![M\ \tau]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = [\![\tau]\!]_{\mathsf{P}} = \bot \\ [\![M]\!]_{\mathsf{P}} & \text{if } [\![\tau]\!]_{\mathsf{P}} = \bot \\ & \quad \text{and } \mathsf{P} \notin \mathsf{ip}(\mathsf{typeof}(M)) \\ [\![M]\!]_{\mathsf{P}}\ \ [\![\tau]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{inl}_{\tau}\ M]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \text{ and } \\ & \quad \mathsf{kindof}(\tau) = \mathsf{K} \setminus \rho \\ [\![M_1]\!]_{\mathsf{P}} & \text{if } [\![\mathsf{typeof}(M)]\!]_{\mathsf{P}} = \bot \\ \mathsf{inl}_{[\![\tau]\!]_{\mathsf{P}}}\ \ [\![M_1]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{inr}_{\tau}\ M]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \text{ and } \\ & \quad \mathsf{kindof}(\tau) = \mathsf{K} \setminus \rho \\ [\![M_1]\!]_{\mathsf{P}} & \text{if } [\![\mathsf{typeof}(M)]\!]_{\mathsf{P}} = \bot \\ \mathsf{inr}_{[\![\tau]\!]_{\mathsf{P}}}\ \ [\![M_1]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

Fig. 9. Projection of PolyChorλ Programs

$$[\![\mathsf{case}\ M\ \mathsf{of\ inl}\ \ x \Rightarrow N_1;\ \ \mathsf{inr}\ \ y \Rightarrow N_2]\!]_{\mathsf{P}} =$$

$$\begin{cases} \mathsf{case}\ \ [\![M]\!]_{\mathsf{P}}\ \ \mathsf{of\ inl}\ \ x \Rightarrow [\![N_1]\!]_{\mathsf{P}}\,;\ \ \mathsf{inr}\ \ y \Rightarrow [\![N_2]\!]_{\mathsf{P}} & \text{if } P \in \mathsf{ip}(\mathsf{typeof}(M)) \\ \bot & \text{if } [\![M]\!]_{\mathsf{P}} = [\![N_1]\!]_{\mathsf{P}} = [\![N_2]\!]_{\mathsf{P}} = \bot \\ [\![M]\!]_{\mathsf{P}} & \text{if } [\![N_1]\!]_{\mathsf{P}} = [\![N_2]\!]_{\mathsf{P}} = \bot \\ [\![N_1]\!]_{\mathsf{P}} \sqcup [\![N_2]\!]_{\mathsf{P}} & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \\ (\lambda\, z : \bot.\ ([\![N_1]\!]_{\mathsf{P}} \sqcup [\![N_2]\!]_{\mathsf{P}}))\ \ [\![M]\!]_{\mathsf{P}}\ \ (z\ \mathit{fresh}) & \text{otherwise} \end{cases}$$

$$[\![(M_1, M_2)]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M_1]\!]_{\mathsf{P}} = [\![M_2]\!]_{\mathsf{P}} = \bot \\ ([\![M_1]\!]_{\mathsf{P}}\,,\,[\![M_2]\!]_{\mathsf{P}}) & \text{otherwise} \end{cases}$$

$$[\![\mathsf{fst}\ M]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \\ [\![M_1]\!]_{\mathsf{P}} & \text{if } [\![\mathsf{typeof}(M)]\!]_{\mathsf{P}} = \bot \\ \mathsf{fst}\ \ [\![M_1]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{snd}\ M]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![M]\!]_{\mathsf{P}} = \bot \\ [\![M_1]\!]_{\mathsf{P}} & \text{if } [\![\mathsf{typeof}(M)]\!]_{\mathsf{P}} = \bot \\ \mathsf{snd}\ \ [\![M_1]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$\left[\!\left[\mathsf{select}_{\mathsf{Q}_1,\mathsf{Q}_2}\ \ell\ M\right]\!\right]_{\mathsf{P}} =$$

$$\begin{cases} \oplus_{\mathsf{Q}'}\ \ell\ \ [\![M]\!]_{\mathsf{P}} & \text{if } \mathsf{P} = \mathsf{Q}_1 \neq \mathsf{Q}_2 \\ \&_S\ \{\ell : [\![M]\!]_{\mathsf{P}}\} & \text{if } \mathsf{P} = \mathsf{Q}_2 \neq \mathsf{Q}_1 \\ [\![M]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$\left[\!\left[\mathsf{com}^{\tau}_{\mathsf{Q}_1,\mathsf{Q}_2}\right]\!\right]_{\mathsf{P}} =$$

$$\begin{cases} \lambda\, x : [\![\tau\ \mathsf{P}]\!]_{\mathsf{P}}\,.\,x & \text{if } \mathsf{P} = \mathsf{Q}_1 = \mathsf{Q}_2 \\ \mathsf{send}_{\mathsf{Q}_2} & \text{if } \mathsf{P} = \mathsf{Q}_1 \neq \mathsf{Q}_2 \\ \mathsf{recv}_{\mathsf{Q}_1} & \text{if } \mathsf{P} = \mathsf{Q}_2 \neq \mathsf{Q}_1 \\ \mathsf{sub}[\mathsf{Q}_1 \mapsto \mathsf{Q}_2] & \text{if } [\![\tau]\!]_{\mathsf{P}} \neq \bot \\ \bot & \text{otherwise} \end{cases}$$

Fig. 10. Projection of PolyChorλ Programs (ctd.)

$$[\![X]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } \mathsf{kindof}(X) = \mathsf{K} \setminus (\{\mathsf{P}\} \cup \rho) \text{ for } \mathsf{K} \neq \mathsf{Proc} \\ X & \text{otherwise} \end{cases} \qquad [\![\mathsf{Q}]\!]_{\mathsf{P}} = \mathsf{Q}$$

$$[\![() \mathbin{@} \mathsf{Q}]\!]_{\mathsf{P}} = \begin{cases} () & \text{if } \mathsf{P} = \mathsf{Q} \\ \bot & \text{otherwise} \end{cases} \qquad [\![\mathsf{Int} \mathbin{@} \mathsf{Q}]\!]_{\mathsf{P}} = \begin{cases} \mathsf{Int} & \text{if } \mathsf{P} = \mathsf{Q} \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\tau_1 \times \tau_2]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![\tau_1]\!]_{\mathsf{P}} = [\![\tau_2]\!]_{\mathsf{P}} = \bot \\ [\![\tau_1]\!]_{\mathsf{P}} \times [\![\tau_2]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\tau_1 + \tau_2]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![\tau_1]\!]_{\mathsf{P}} = [\![\tau_2]\!]_{\mathsf{P}} = \bot \\ [\![\tau_1]\!]_{\mathsf{P}} + [\![\tau_2]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\tau_1 \to_\rho \tau_2]\!]_{\mathsf{P}} = \begin{cases} [\![\tau_1]\!]_{\mathsf{P}} \to [\![\tau_2]\!]_{\mathsf{P}} & \text{if } \mathsf{P} \in \rho \text{ or } [\![\tau_1]\!]_{\mathsf{P}} \neq \bot \neq [\![\tau_2]\!]_{\mathsf{P}} \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\forall X :: \mathsf{K}.\ \tau]\!]_{\mathsf{P}} =$$

$$\begin{cases} \bot & \text{if } [\![\tau]\!]_{\mathsf{P}} = \bot \text{ and } \mathsf{K} = \mathsf{K}' \setminus (\{\mathsf{P}\} \cup \rho) \\ \forall X.\ \mathsf{AmI}\ X\ ?\ [\![\tau[X \mapsto \mathsf{P}]]\!]_{\mathsf{P}}\ \&\ [\![\tau]\!]_{\mathsf{P}} & \text{if } \mathsf{K} \in \{\mathsf{Proc},\ \mathsf{Proc} \setminus \rho\} \\ \forall X.\ [\![\tau]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\tau_1\ \tau_2]\!]_{\mathsf{P}} = \begin{cases} \bot & \text{if } [\![\tau_1]\!]_{\mathsf{P}} = [\![\tau_2]\!]_{\mathsf{P}} = \bot \\ [\![\tau_1]\!]_{\mathsf{P}} & \text{if } [\![\tau_2]\!]_{\mathsf{P}} = \bot \text{ and } \mathsf{kindof}(\tau_2) = \mathsf{K} \setminus (\{\mathsf{P}\} \cup \rho) \\ [\![\tau_2]\!]_{\mathsf{P}} & \text{if } [\![\tau_1]\!]_{\mathsf{P}} = \bot \\ [\![\tau_1]\!]_{\mathsf{P}}\ [\![\tau_2]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

$$[\![\lambda X :: \mathsf{K}.\ \tau]\!]_{\mathsf{P}} =$$

$$\begin{cases} \bot & \text{if } [\![\tau]\!]_{\mathsf{P}} = \bot \text{ and } \mathsf{K} = \mathsf{K}' \setminus (\{\mathsf{P}\} \cup \rho) \\ \lambda X.\ \mathsf{AmI}\ X\ ?\ [\![\tau[X \mapsto \mathsf{P}]]\!]_{\mathsf{P}}\ \&\ [\![\tau]\!]_{\mathsf{P}} & \text{if } \mathsf{K} \in \{\mathsf{Proc},\ \mathsf{Proc} \setminus \rho\} \\ \lambda X.\ [\![\tau]\!]_{\mathsf{P}} & \text{otherwise} \end{cases}$$

Fig. 11. Projection of PolyChorλ Types

5 The Correctness of Endpoint Projection

We now show that there is a close correspondence between the executions of choreographies and of their projections. Intuitively, this correspondence states that a choreography can execute an action if, and only if, its projection can execute the same action, and both transition to new terms in the same relation. However, this is not completely true: if a choreography M reduces by rule [CaseL], 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 branching relation (Montesi, 2023; Cruz-Filipe and Montesi, 2020), defined by $M \sqsupseteq N$ iff $M \sqcup N = M$. Intuitively, if $M \sqsupseteq N$, then M offers the same and possibly more behaviours than N. This notion extends to networks by defining $\mathcal{N} \sqsupseteq \mathcal{N}'$ to mean that, for any role P, $\mathcal{N}(\mathsf{P}) \sqsupseteq \mathcal{N}'(\mathsf{P})$.

Using this, we can show that the EPP of a choreography can do all that (completeness) and only what (soundness) the original choreography does. For traditional imperative choreographic languages, this correspondence takes the form of one action in the choreography corresponding to one action in the projected network. We instead have a correspondence between one action in the choreography and multiple actions in the network due to allowing choreographies to manipulate distributed values in one action such as in $\lambda x : \mathsf{Int} \mathbin{@} \mathsf{Bob} \times \mathsf{Int} \mathbin{@} \mathsf{Alice}.\, M\ (3 \mathbin{@} \mathsf{Bob}, 3 \mathbin{@} \mathsf{Alice})$ where both Bob and Alice independently take the first part of the pair.

Theorem 5 (Completeness)**.** *Given a closed choreography M, if $M \to_D M'$, $\Theta;\Gamma \vdash D$, $\Theta;\Gamma \vdash M : \tau$, and $[\![M]\!]$ is defined, then there exists network $\mathcal{N}$ and choreography M'' such that: $[\![M]\!] \to^{+}_{[\![D]\!]} \mathcal{N}$ and $\mathcal{N} \sqsupseteq [\![M']\!]$.*

Proof We prove this by structural induction on $M \to_D M'$. We take advantage of the fact that type values project to $\bot$ at processes not involved in them, while choreographic values correspondingly project to $\bot$ at processes not involved in their type. See Appendix 3 for full details. ■

Theorem 6 (Soundness)**.** *Given a closed choreography M and a function mapping D, if $\Theta;\Gamma \vdash M : \tau$, $\Theta;\Gamma \vdash D$, and $[\![M]\!] \to^{*}_{[\![D]\!]} \mathcal{N}$ for some network $\mathcal{N}$, then there exist a choreography M' and a network $\mathcal{N}'$ such that: $M \to^{*}_{D} M'$, $\mathcal{N} \to^{*}_{[\![D]\!]} \mathcal{N}'$, and $\mathcal{N}' \sqsupseteq [\![M']\!]$.*

Proof We prove this by structural induction on M in the accompanying technical report, taking advantage of the fact that thanks to projecting function names everywhere, a choreography that diverges at one role diverges at every role. See Appendix 4 for full details. ■

From Theorems 3 to 6, 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 1. *Given a closed choreography M and a function environment D containing all the function names of M, if $\Theta;\Gamma \vdash M : T$ and $\Theta;\Gamma \vdash D$, then: whenever $[\![M]\!] \to^{*}_{[\![D]\!]} \mathcal{N}$ for some network $\mathcal{N}$, either there exists $\mathscr{P}$ such that $\mathcal{N} \xrightarrow{\tau_{\mathscr{P}}}_{[\![D]\!]} \mathcal{N}'$ or $\mathcal{N} = \prod_{P \in ip(M)} P[V_P]$.*

$$
\begin{array}{l}
\lambda\,\mathsf{HandleHere} : (\mathsf{Int} \mathbin{@} \mathsf{E} \to_{\varnothing} \mathsf{Int} \mathbin{@} \mathsf{E}) \to_{\varnothing} \mathsf{Bool} \mathbin{@} \mathsf{E}.\\
\Bigg(\lambda\, x : \mathsf{Int} \mathbin{@} \mathsf{E} \to_{\varnothing} \mathsf{Int} \mathbin{@} \mathsf{E}.\\
\quad \mathsf{LogRequest}\ \mathsf{E}\ \mathsf{S}\ \mathsf{C}\ x\\
\quad \Bigg(\mathsf{case}\ \mathsf{HandleHere}\ x\ \mathsf{of}\\
\qquad \mathsf{inl}\ \ y \Rightarrow \left(\begin{array}{l} \mathsf{select}_{\mathsf{E},\mathsf{C}}\ \mathsf{here}\\ \mathsf{select}_{\mathsf{E},\mathsf{S}}\ \mathsf{here}\\ \mathsf{com}_{\mathsf{E},\mathsf{C}}^{\lambda X::\mathsf{Proc}\setminus\{\mathsf{S}\}.\ \mathsf{Int}@X}\ (\mathsf{Compute}\ x) \end{array}\right);\\
\qquad \mathsf{inr}\ \ y \Rightarrow \mathsf{select}_{\mathsf{E},\mathsf{C}}\ \mathsf{atS}\\
\qquad\qquad \mathsf{select}_{\mathsf{E},\mathsf{S}}\ \mathsf{you}\\
\qquad\qquad \mathsf{case}\ \left(\left(\begin{array}{l} \mathsf{com}_{\mathsf{E},\mathsf{C}}^{\lambda Y::\mathsf{Proc}\setminus\{\mathsf{S}\}.\ \mathsf{Int}@Y \to_{\varnothing} \mathsf{Bool}@\mathsf{S}}\\ \left(\begin{array}{l} \mathsf{com}_{\mathsf{S},\mathsf{E}}^{\lambda Y::\mathsf{Proc}\setminus\{\mathsf{C}\}.\ \forall X::\mathsf{Proc}.\ \mathsf{Int}@Y \to_{\varnothing} \mathsf{Bool}@X}\\ \quad \Lambda\,\mathsf{Z} :: \mathsf{Proc}.\ \lambda\, z : \mathsf{Int} \mathbin{@} \mathsf{S}.\ \mathsf{Authenticate}\\ \qquad\qquad \mathsf{com}_{\mathsf{S},\mathsf{Z}}^{\lambda X::\mathsf{Proc}.\ \mathsf{Int}@X}\ z \end{array}\right)\\ \mathsf{S} \end{array}\right)\\ \mathsf{AuthKey} \end{array}\right)\ \mathsf{of}\\
\qquad\qquad\quad \mathsf{inl}\ \ y \Rightarrow \left(\begin{array}{l} \mathsf{select}_{\mathsf{S},\mathsf{E}}\ \mathsf{go}\\ \mathsf{select}_{\mathsf{S},\mathsf{C}}\ \mathsf{go}\\ \mathsf{com}_{\mathsf{S},\mathsf{C}}^{\lambda X::\mathsf{Proc}\setminus\{\mathsf{E}\}.\ \mathsf{Int}@X}\\ \quad \mathsf{Compute}\ (\mathsf{com}_{\mathsf{E},\mathsf{S}}^{\lambda X::\mathsf{Proc}\setminus\{\mathsf{C}\}.\ \mathsf{Int}@X \to_{\varnothing} \mathsf{Int}@X}\ x) \end{array}\right);\\
\qquad\qquad\quad \mathsf{inr}\ \ y \Rightarrow \mathsf{select}_{\mathsf{S},\mathsf{E}}\ \mathsf{no}\\
\qquad\qquad\qquad\qquad \mathsf{select}_{\mathsf{S},\mathsf{C}}\ \mathsf{no}\\
\qquad\qquad\qquad\qquad 0 \mathbin{@} \mathsf{C} \Bigg)\Bigg)\\
\quad \mathsf{com}_{\mathsf{C},\mathsf{E}}^{\lambda X::\mathsf{Proc}\setminus\{\mathsf{C}\}.\ \mathsf{Int}@X \to_{\varnothing} \mathsf{Int}@X}\ \mathsf{task}
\end{array}
$$

Fig. 12. Case Study Code

6 Case Study

We now show how our language can be used in an extended example (Figure 12). This example involves three processes: a client C, an edge computer E, and a server S. Intuitively, C wants to request that E does some computation. However, E may not have the resources to perform the computation; in this case, it will forward the request to S. Whenever S receives a request, then C must first perform an authentication protocol. Whether or not the task is outsourced to S, S logs the request.

Here we assume the following data:

- A task (of type $\mathsf{Int} \mathbin{@} \mathsf{C} \to_{\varnothing} \mathsf{Int} \mathbin{@} \mathsf{C}$) located at C
- For each of E and S, a local function Compute which executes a task
- An authentication choreography Authenticate between S and a another process Z. This choreography takes a key AuthKey and checks if the holder of that key is authorized to run a task on S.
- A key AuthKey for C
- A logging choreography LogRequest involving two roles, provocatively called E and S. This choreography takes a client, a task, and the result of executing the task (at C) as input. It then creates a log entry at S.

- A local function HandleHere, which E uses to determine whether it can handle a task locally. If HandleHere returns false, then the task must be shipped to S. Unlike other data, this is represented as input to the choreography.

The choreography begins with C sending the task to E; we call the resulting task x. (Note that x is the name of the task *on* E, not the name of the task on C.) E then checks whether it can handle x using HandleHere. If so, E informs S and C that it is computing the task. After performing the task, E sends the result to C. It furthermore informs S so that it knows that it needs to log the task.

If E cannot handle the task, then it again informs C and S. S then makes a decision on whether C has authorisation to request a task from S. To make this decision, S sends an authentication protocol to E. Because communications swap the sender and the receiver in the communicated value, we write this authentication protocol with S playing the role of the client. The protocol is therefore parameterized on the authenticator. Once E receives the authentication protocol, it can instantiate the authenticator as S. E finally sends the (now complete) protocol to C; running this protocol will have C send its key to S, possibly among other actions required for authentication. If the authentication procedure succeeds, then S informs E and C of this. E can then send the task to S, who computes it and returns the result to C. If authentication fails, then S informs E and C of this and the task fails, resulting in a 0 on C. Either way, we finish the choreography by logging the task and its result using the function LogRequest.

For S to send an authentication protocol which it is itself involved with requires a bit of trickery. Usually, we would expect every part of the sent value located at S to be moved to the receiver (first E and then after another communication C) but obviously that would mean S cannot be involved. We therefore send an authentication protocol that is parametric on the authenticator, Z, and only instantiate Z as S after the first communication from S to E has taken place.

Projecting this protocol to C leads to the following code:

$$
\begin{array}{l}
\lambda\, \mathsf{HandleHere} : \bot. \\
\left(\begin{array}{l}
\lambda\, x : \bot. \\
\quad \mathsf{LogRequest}\; E\; S\; C\; \bot \\
\quad \&_E \left\{\begin{array}{l}
\mathsf{here} : \mathsf{recv}_E\; \bot, \\
\mathsf{atS} : \left(\begin{array}{l}
\lambda\, y : \bot.\, \&_S\; \{\mathsf{go} : \mathsf{recv}_S\; \bot, \mathsf{no} : 0\} \\
\left(\left(\begin{array}{l}
\mathsf{recv}_E\; \mathsf{sub}[S \mapsto E] \left(\begin{array}{l}
\Lambda\, Z :: \mathsf{Proc}. \\
\quad \mathsf{AmI}\; Z \\
\quad\; ?\, \lambda\, z : \bot.\, \mathsf{Authenticate} \\
\qquad\qquad \mathsf{recv}_S\; \bot \\
\quad\; \&\; \mathsf{sub}[S \mapsto Z]\; \bot
\end{array}\right) \\
\qquad S
\end{array}\right) \\
\mathsf{AuthKey}
\end{array}\right)\right)
\end{array}\right)
\end{array}\right\}
\end{array}\right) \\
\mathsf{send}_E\; \mathsf{task}
\end{array}
\tag{6.1}
$$

We see that the second case gets projected as an application of a new abstraction on the new variable y, with C's part of the condition as the right side of the application. Since the condition contains a delegation, we get some process substitutions representing values with unknown locations being communicated between other processes. Because of the AmI in the second branch, none of the substituted processes are ever reached. Therefore, these substitutions do nothing. Since C is not involved in E's decision to delegate to S (or not), we do not see any of the code involved in the decision here. Instead, we get a straightforward offer as the result of merging the projection of each branch of the involved conditional.

We now show the projection for E:

$$
\begin{array}{l}
\lambda\ \mathsf{HandleHere} : \mathsf{Int} \to \mathsf{Int} \to \mathsf{Bool}. \\
\left(\begin{array}{l}
\lambda\ x : \mathsf{Int} \to \mathsf{Int}. \\
\quad \mathsf{LogRequest}\ E\ S\ C\ x \\
\quad\ \mathsf{case}\ \mathsf{HandleHere}\ x\ \mathsf{of} \\
\qquad \mathsf{inl}\ \ y \Rightarrow \left(\begin{array}{l} \oplus_C\ \mathsf{here} \\ \oplus_S\ \mathsf{here} \\ \mathsf{send}_C\ \ \mathsf{Compute}\ x \end{array}\right); \\
\qquad \mathsf{inr}\ \ y \Rightarrow \oplus_C\ \mathsf{atS} \\
\qquad\qquad\qquad \oplus_S\ \mathsf{you} \\
\qquad\qquad\qquad \left(\begin{array}{l}
\lambda\ y : \bot.\ \&_S\ \{\mathsf{go} : \mathsf{send}\ \ x, \mathsf{no} : \bot\} \\
\left(\left(\begin{array}{l}
\mathsf{send}_C \left(\begin{array}{l}
\mathsf{recv}_S \left(\begin{array}{l}
\Lambda\, \mathsf{Z} :: \mathsf{Proc}.\ \mathsf{AmI}\ \mathsf{Z} \\
\qquad ?\,\lambda\ z : \bot.\ \mathsf{Authenticate} \\
\qquad\qquad \mathsf{recv}_{\mathsf{Z}}\ \bot \\
\qquad \&\ \ \mathsf{sub}[S \mapsto \mathsf{Z}]\ \bot
\end{array}\right) \\
\quad S
\end{array}\right) \\
\bot
\end{array}\right)\right)
\end{array}\right)
\end{array}\right) \\
\quad \mathsf{recv}_C\ \bot
\end{array}
\tag{6.2}
$$

Note that we treat the second case almost the same as in C, except that E is involved in both communications of the delegation. Since the condition of the first case is located at E, it gets projected as a case. Keep in mind that since we model communication as an exchange, what will actually be executed at S after the delegation takes place is the right branch of the AmI in the projection of E.

Finally, we show the projection for S:

$$
\lambda\,\mathsf{HandleHere} : \bot.\;
\left(
\begin{array}{l}
\lambda x : \bot. \\
\quad \mathsf{LogRequest}\; E\; S\; C\; \bot \\
\qquad \&_E \left\{ \begin{array}{l}
\mathsf{here} : \bot, \\
\mathsf{you} : \left(\mathsf{case} \left(\left(\mathsf{sub}[E \mapsto C] \left(\begin{array}{l}
\mathsf{send}_E\; \Lambda Z :: \mathsf{Proc}. \\
\qquad \mathsf{AmI}\; Z \\
\qquad ?\,\lambda z : \mathsf{Int}.\; \mathsf{Authenticate} \\
\qquad\qquad \lambda y : \mathsf{Int}.\; y\; z \\
\qquad \&\; \lambda z : \mathsf{Int}.\; \mathsf{Authenticate} \\
\qquad\qquad \mathsf{send}_Z\; z \\
S
\end{array} \right) \right) \; \bot \right) \; \mathsf{of} \left(\begin{array}{l}
\mathsf{inl}\; y \Rightarrow \left(\begin{array}{l} \oplus_E\; \mathsf{go} \\ \oplus_C\; \mathsf{go} \\ \mathsf{send}_C\; (\mathsf{Compute}\; (\mathsf{recv}_E\; \bot)) \end{array} \right); \\
\mathsf{inr}\; y \Rightarrow \oplus_E\; \mathsf{no} \\
\qquad\qquad \oplus_C\; \mathsf{no} \\
\qquad\qquad \bot
\end{array} \right) \right)
\end{array} \right. \\
\bot
\end{array}
\right)
\tag{6.3}
$$

Here we finally see the projection of what S actually wants C to do in order to authenticate. We also see that in the case where Z gets instantiated as the same process it is communicating with, which would be S if the protocol did not get communicated before Z is instantiated, the communication gets replaced by an identity function $\lambda\, y : \mathsf{Int}.\, y$.

7 Related Work

7.1 Choreographies

Choreographies are inspired by the "Alice and Bob" notation for security protocols by Needham and Schroeder (1978), which included a term for expressing a communication from a participant to another. The same idea inspired later several graphical notations for modelling interactions among processes, including sequence diagrams and message sequence charts (Object Management Group, 2017; International Telecommunication Union, 1996).

A systematic introduction to theory of choreographic languages and their historical development can be found in (Montesi, 2023). We recap and discuss relevant recent developments. The first sophisticated languages for expressing choreographies were invented to describe interactions between web services. The Web Services Choreography Description Language (WS-CDL) by The World Wide Web Consortium (W3C) (2004) is a choreographic language which describes the expected observable interactions between web services from a global point of view (Zongyan et al., 2007). Carbone et al. (2012) later formalized endpoint projection for a theory of choreographies based on WS-CDL, and in particular introduced the merging operator (which we adjusted and extended to our setting). This inspired more work on choreographies and projection and eventually the birth of choreographic programming—where choreographies are programs compiled to executable code—and the first choreographic programming language, Chor (Montesi, 2013). As choreographic programming languages became more complex, Cruz-Filipe and Montesi (2020) developed a *core calculus of choreographies* (CC). Montesi (2023) revisited and generalised CC in his text on foundations of choreographic languages. Cruz-Filipe et al. (2021) then formalized this new version and its properties in the Coq theorem prover (The Coq development team, 2004). Later, Pohjola et al. (2022) developed a certified end-to-end compiler from another variation of CC to CakeML by using the HOL theorem prover.

One of the primary design goals of all of choreographic programming languages is *deadlock-freedom by design* (Carbone and Montesi, 2013)—the operational correspondence between the choreography and the distributed network ensures deadlock-freedom for the network. PolyChorλ achieves this goal. Montesi (2023) discusses restrictions for a procedural imperative choreographic language in order to obtain a stronger liveness property (starvation-freedom). The idea is to prove that processes eventually get involved in transitions at the choreographic level, and then leverage the correctness of endpoint projection to obtain the same result about choreography projections. This idea might work for PolyChorλ as well, but whether and how the technical devices for starvation-freedom in (Montesi, 2023) can be adapted to PolyChorλ is not clear due to the different nature of our language (functional instead of imperative). Alternatively, one could adapt static analyses for lock-freedom—like that in (Kobayashi, 2006)—to choreographies. We leave explorations of liveness properties other than deadlock-freedom in PolyChorλ to future work.

The first choreographic language with limited process polymorphism was Procedural Choreographies (PC) (Cruz-Filipe and Montesi, 2017). In PC, a choreography comes with an environment of predefined *procedures* parametric on process names which may be called by the choreography. These procedures have a number of limitations compared to the process polymorphism of PolyChorλ: they cannot contain any free processes, they cannot be partially instantiated, and they are lower-order—that is, they must be defined in the environment rather than as part of a larger choreography. These limitations allow the projection function to know how the procedure will be instantiated, whereas in PolyChorλ we may need to compute the processes involved first. This has major implications for projection: in PC, it is easy to tell when projecting a procedure call which processes are involved and therefore need a local copy of the call. However, PolyChorλ's full process polymorphism allows the function and process names to each be enclosed in a context. While this allows greater flexibility for programmers, it forces us to project a process-polymorphic functions to every process and let each process determine at runtime whether it is involved.

Recently, there has been a fair amount of interest in higher-order and functional programming for choreographies (Giallorenzo et al., 2020; Hirsch and Garg, 2022; Cruz-Filipe et al., 2022; Shen et al., 2023). The first higher-order choreographic programming language, Choral (Giallorenzo et al., 2020), is an object-oriented choreographic language compiled to Java code. Thus, Choral choreographies can depend on other choreographies, allowing programmers to reuse code. Choral was also the first choreographic language to treat $\mathsf{com}^{\tau}_{\mathsf{P},\mathsf{Q}}$ as a first-class function.

While Choral gave a practical implementation of higher-order choreographies, it did not establish their theoretical properties. Two different—but independently developed—works filled this gap, including Chorλ, the basis of PolyChorλ. Chorλ is a functional choreographic calculus based on the λ-calculus. In this work, we extended Chorλ with type and process polymorphism and the ability to send non-local values such as choreographies. Chorλ, and hence PolyChorλ, provides a core language for higher-order choreographies, thus allowing us to establish their properties. Since the original Chorλ has parametric procedures like PC and Choral, it lacks PolyChorλ's property that a choreography diverging in one process must diverge in every process. This forces Chorλ to have both a complex notion of out-of-order execution and a more lax correspondence between actions in the network and the choreography.

The other work establishing the theoretical properties of higher-order choreographic programming is Pirouette (Hirsch and Garg, 2022), which is also a functional choreographic programming language based on simply-typed λ calculus. Unlike Chorλ (and thus PolyChorλ), Pirouette does not allow processes to send messages written in Pirouette. Instead, it takes inspiration from lower-order choreographic programming languages in which (the computations to produce) messages are written in their own separate language. Like other choreographic languages (Montesi, 2023; Cruz-Filipe et al., 2021), Pirouette's design is parametrized by the language for writing messages. Thus, Pirouette can describe communication patterns between processes that draw from a large swath of languages for their local computations. Nevertheless, this design means that Pirouette fundamentally cannot allow programs to send choreographic functions, unlike PolyChorλ.

Moreover, unlike Chorλ and PolyChorλ, Pirouette forces every process to synchronize when applying a function. This allows Pirouette to establish a bisimulation relation with its network programming language, a result formalized in Coq. This result allows a traditional—and verified—proof of deadlock-freedom by construction. However, this constant synchronization would be a bottleneck in real-world systems; PolyChorλ's choice to obtain a weaker—but strong-enough—connection between the languages allows it to avoid this high cost.

7.2 Concurrent Functional Programming

Functional concurrent programming has a long history, starting with attempts to parallelize executions of functional programs (Burton, 1987). The first language for functional programming with communications on channels was Facile (Giacalone et al., 1989) which, unlike later choreographic languages, had an abstraction over *process IDs* very similar to process polymorphism. A more recent work, which more-closely resembles choreographic programming, is Links (Cooper et al., 2006), with the RPC calculus (Cooper and Wadler, 2009) as its core language. Links and the RPC calculus, like choreographies, allow a programmer to describe programs where different parts of the computation takes place at different locations and then compile it to separate code for each location. Interestingly, though Links has explicit communication, in the RPC calculus the top level does not, and communications are created only when projecting a function located at a different process. Moreover, the RPC calculus does not feature multiple threads of computation; instead, on communication the single thread of computation moves to a new location while other locations block. The RPC calculus was later extended with location polymorphism, very similar to our and Facile's process polymorphism (Choi et al., 2020). However, as the RPC calculus only deals with systems of 2 processes, a client and a server, they project a process abstraction as a pair, and then the location as picking the correct part of this pair. This solution creates a simpler network language but is not suitable for a framework with an arbitrary number of participants such as PolyChorλ. Moreover, the RPC calculus—like PolyChorλ but unlike traditional choreographic languages—does not have out-of-order execution at the top level.

Session types were applied to a concurrent functional calculus with asynchronous communication by Gay and Vasconcelos (2010). Though initially this language did not guarantee deadlock-freedom, only runtime safety, later versions of GV (Wadler, 2012; Lindley and Morris, 2015) did. Jacobs et al. (2022) extended GV with *global types* (Honda et al., 2016), which generalise session types to protocols with multiple participants. Similarly to choreographic programming, global types offer a global viewpoint on interactions. However, they are intended as specifications and thus cannot express computation. Global types are typically projected onto *local types*, which manually-written programs can later be checked against. In choreographic programming, by contrast, choreographies are projected directly to programs. Some works mix the approaches (e.g., Scalas et al., 2017): given a global type, a compiler produces typestate-oriented libraries (Aldrich et al., 2009) that help the users with following the global type correctly (but not with performing the right computations at the right time).

Session types have also been used to study global higher-order programming outside of functional settings. Mostrous and Yoshida (2007) describe the challenges associated with obtaining subject reduction when sessions can pass other sessions between them. Based on this, Mostrous and Yoshida (2009) define the higher-order π-Calculus with asynchronous sessions, a calculus combining elements of the π-calculus and λ-calculus.

Castellani et al. (2020) propose a notion of *session types with delegation*. They write delegation by enclosing a part of the global type in brackets. During the execution of such a part, one process acts on another's behalf by temporarily taking its name. This means that they do not need to inform other participants in the delegated computation that delegation is happening. However, nested delegations can cause deadlocks.

ML5 (Licata and Harper, 2010; Murphy VII et al., 2007) is a functional concurrent programming language based on the semantics of modal logic. However, instead of the send and recv terms of choreographic languages, they have a primitive get$[w]$ M, which makes another process w evaluate M and return the result. Since M may include other gets, this construct gives ML5 something resembling PolyChorλ's ability to send a full choreography. However, the result of evaluating this "choreography" must be at the receiver and then returned to the sender.

Multitier programming languages, like ScalaLoci (Weisenburger and Salvaneschi, 2020), offer another paradigm for describing global views of distributed systems. Like Choral, ScalaLoci is built on top of an existing object-oriented language: in this case, Scala. In ScalaLoci and other *tierless* languages, an object describes a whole system containing multiple processes and functions. Differently from choreographic programming, multitier programming does not allow for modelling the intended flow of communications. Rather, communication happens implicitly through remote function calls and the concrete protocol to be followed is largely left to be decided by a runtime middleware. For a more detailed comparison of choreographic and multitier languages, see the work of Giallorenzo et al. (2021).

8 Conclusion

In this paper, we presented PolyChorλ, the first higher-order choreographic programming language with process polymorphism. PolyChorλ has a type and kind system based on System Fω, but extended such that process names are types of kind Proc. Moreover, we showed how to obtain a concurrent interpretation of PolyChorλ programs in a process language by using a new construct corresponding by the ability of a process to know its identity. We found that this construct was necessary if process variables are able to be instantiated as the process they are located at, but using a choreographic language abstracts from this necessity. Our explorations of process polymorphism also allowed PolyChorλ to describe a communication of a non-local value from P to Q as sending the part of the message owned by P to Q. These non-local values include full choreographies, creating a simple and flexible way to describe delegation by communicating a distributed function describing the delegated task. This innovation required a new notion of communication as an exchange in which the delegator rather than being left with an empty value after sending a choreography is left with a function which will allow it to potentially take part in the delegated task, e.g., by receiving a result at the end.

Process polymorphism fills much of the gap between previous works on the theory of higher-order choreographies and practical languages. However, there is still more work to do. For instance, currently PolyChorλ does not support recursive types. Our current results rely on types normalizing to a type value, which recursive types may not do. System Fω does not have our restriction of type abstractions only being instantiated with type values. However, PolyChorλ needs to ensure that communications are only undertaken between processes, rather than complicated type expressions resulting in processes. Thus, we need to treat our type system as call-by-name, leading to the restriction above. In order to support recursive types, we would need to either make endpoint projection capable of projecting to a possibly-nonterminating description of a process, or limit recursive types ability to make type computations fail to terminate.

Furthermore, one can imagine allowing processes to send types and process names as well as values. This would, for example, allow us to program a server to wait to receive the name of a client which it will have to interact with. Since this form of delegation is common in practice, understanding how to provide this capability in a choreographic language, while retaining the guarantees of choreographic programming, would enable programmers to apply their usual programming patterns to choreographic code.

We project local type despite lacking a typing system for local processes. Our unusual network communication semantics have made it difficult to define local typing rules for sends and recvs, and we therefore leave local typing (or alternatively type erasure) as future work.

Certain, instant, and synchronous communication is convenient for theoretical study, but such assumptions do not match real-world distributed systems. Cruz-Filipe and Montesi

(2017) model asynchronous communication in choreographies via runtime terms representing messages in transit. We could adapt this method to PolyChorλ by having the communication primitive reduce in two steps: first to a runtime term and then to the delivered value. However, this extension would be nontrivial, since it is not obvious how to represent messages in transit when those messages are non-local values such as choreographies. In addition, the way we represent a communication at the local level (swapping values rather than only moving a value from sender to receiver) might require additional machinery (e.g., new runtime terms) to capture its asynchronous execution.

We also leave practical implementation of PolyChorλ's new features to future work. This could be achieved by extending Choral (Giallorenzo et al., 2020), the original inspiration for Chorλ. Communication primitives in Choral are user-defined—not fixed by any middleware or compiler—so it is possible to define new communication abstractions involving multiple roles. However, we need to manipulate roles at runtime in our local semantics, while the Choral compiler erases roles when projection code to Java. We may be able to overcome this issue by reifying roles in projected code or by using reflection.

While these gaps between theory and practice persist, process polymorphism in PolyChorλ brings us much closer to realistic choreographic languages for distributed systems. Choreographic programs promise to provide easier and cleaner concurrent and distributed programming with better guarantees. With higher-order choreographic programming and process polymorphism, the fulfilment of that promise is nearly within reach.

Acknowledgements

Graversen and Montesi were partially supported by Villum Fonden, grant no. 29518.

References

Aldrich, J., Sunshine, J., Saini, D. & Sparks, Z. (2009) Typestate-oriented programming. Companion to the 24th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2009, October 25-29, 2009, Orlando, Florida, USA. ACM. pp. 1015–1022.

Burton, F. (1987) Functional programming for concurrent and distributed programming. *The Computer Journal*. **30**(5), 437–450.

Caires, L. & Pfenning, F. (2010) Session types as intuitionistic linear propositions. Concurrency Theory (CONCUR).

Carbone, M., Honda, K. & Yoshida, N. (2012) Structured communication-centered programming for web services. *Transactions on Programming Languages and Systems (TOPLAS)*. **34**(2).

Carbone, M. & Montesi, F. (2013) Deadlock-freedom-by-design: Multiparty asynchronous global programming. Principles of Programming Languages (POPL).

Castagna, G., Dezani-Ciancaglini, M. & Padovani, L. (2012) On global types and multi-party session. *Log. Methods Comput. Sci.* **8**(1).

Castellani, I., Dezani-Ciancaglini, M., Giannini, P. & Horne, R. (2020) Global types with internal delegation. *TCS*. **807**, 128–153.

Choi, K., Cheney, J., Fowler, S. & Lindley, S. (2020) A polymorphic RPC calculus. *SCP*. **197**, 102499.

Cooper, E., Lindley, S., Wadler, P. & Yallop, J. (2006) Links: Web programming without tiers. Formal Methods for Components and Objects (FMCO).

Cooper, E. E. & Wadler, P. (2009) The RPC calculus. Principles and Practice of Declarative Programming (PPDP).
Cruz-Filipe, L., Graversen, E., Lugovic, L., Montesi, F. & Peressotti, M. (2022) Functional choreographic programming. ICTAC.
Cruz-Filipe, L., Graversen, E., Lugovic, L., Montesi, F. & Peressotti, M. (2023) Modular compilation for higher-order functional choreographies. ECOOP.
Cruz-Filipe, L. & Montesi, F. (2017) On asynchrony and choreographies. Interaction and Concurrency Experience (ICE).
Cruz-Filipe, L. & Montesi, F. (2017) Procedural choreographic programming. Formal Techniques for Distributed Objects, Components, and Systems (FORTE).
Cruz-Filipe, L. & Montesi, F. (2020) A core model for choreographic programming. *Theor. Comput. Sci.* **802**, 38–66.
Cruz-Filipe, L., Montesi, F. & Peressotti, M. (2021) Formalizing a turing-complete choreographic language in coq. Interactive Theorem Proving (ITP).
Dalla Preda, M., Gabbrielli, M., Giallorenzo, S., Lanese, I. & Mauro, J. (2017) Dynamic choreographies: Theory and implementation. *Logical Methods in Computer Science (LMCS)*. **13**(2).
Dardha, O., Giachino, E. & Sangiorgi, D. (2012) Session types revisited. Principles and Practice of Declarative Programming (PPDP).
DeYoung, H., Caires, L., Pfenning, F. & Toninho, B. (2012) Cut reduction in linear logic as asynchronous session-typed communication. Computer Science Logic (CSL).
Gay, S. J. & Vasconcelos, V. T. (2010) Linear type theory for asynchronous session types. *Journal of Functional Programming (JFP)*. **20**(1).
Giacalone, A., Mishra, P. & Prasad, S. (1989) Facile: A symmetric integration of concurrent and functional programming. *International Journal of Parallel Programming*. **18**, 121–160.
Giallorenzo, S., Montesi, F. & Peressotti, M. (2020) Choreographies as objects.
Giallorenzo, S., Montesi, F., Peressotti, M., Richter, D., Salvaneschi, G. & Weisenburger, P. (2021) Multiparty languages: The choreographic and multitier cases (pearl). European Conference on Object-Oriented Programming (ECOOP).
Girard, J.-Y. (1972) *Interprétation fonctionnelle et élimination des coupures de l'artihmétique d'ordre supérieur*. Ph.D. thesis. Université Paris 7.
Hirsch, A. K. & Garg, D. (2022) Pirouette: Higher-order typed functional choreographies. Principles of Programming Languages (POPL).
Honda, K. (1993) Types for dyadic interaction. Concurrency Theory (CONCUR).
Honda, K., Vasconcelos, V. T. & Kubo, M. (1998) Language primitives and type discipline for structured communication-based programming. European Symposium on Programming (ESOP).
Honda, K., Yoshida, N. & Carbone, M. (2016) Multiparty asyncrhonous session types. *Journal of the ACM*. **63**(1), 1–67.
International Telecommunication Union. (1996) Recommendation Z.120: Message sequence chart.
Jacobs, J., Balzer, S. & Krebbers, R. (2022) Multiparty GV: functional multiparty session types with certified deadlock freedom. *Proc. ACM Program. Lang.* **6**(ICFP), 466–495.
Jongmans, S. & van den Bos, P. (2022) A predicate transformer for choreographies - computing preconditions in choreographic programming. European Symposium on Programming (ESOP).
Kobayashi, N. (2006) A new type system for deadlock-free processes. CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings. Springer. pp. 233–247.
Licata, D. R. & Harper, R. (2010) A monadic formalization of ML5. Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP).
Lindley, S. & Morris, J. (2017) Lightweight functional session types. In *Behavioural Types: from Theory to Tools*, Gay, S. & Ravara, A. (eds). River Publishers. chapter 12.
Lindley, S. & Morris, J. G. (2015) A semantics for propositions as sessions. Programming Languages and Systems.
López, H. A., Nielson, F. & Nielson, H. R. (2016) Enforcing availability in failure-aware communicating systems. Formal Techniques for Distributed Objects, Components, and Systems

(FORTE).

The Coq development team. (2004) *The Coq proof assistant reference manual*. LogiCal Project. Version 8.0.

Montesi, F. (2013) *Choreographic Programming*. Ph.D. thesis. IT University of Copenhagen.

Montesi, F. (2023) *Introduction to Choreographies*. Cambridge University Press.

Mostrous, D. & Yoshida, N. (2007) Two session typing systems for higher-order mobile processes. TLCA. Springer. pp. 321–335.

Mostrous, D. & Yoshida, N. (2009) Session-based communication optimisation for higher-order mobile processes. TLCA. Springer. pp. 203–218.

Murphy VII, T., Crary, K. & Harper, R. (2007) Type-safe distributed programming with ML5. Trustworthy Global Computer (TGC).

Needham, R. & Schroeder, M. (1978) Using encryption for authentication in large networks of computers. *Commun. ACM*. **21**(12), 993–999.

Needham, R. M. & Schroeder, M. D. (1978) Using encryption for authentication in large networks of computers. *Communications of the ACM (CACM)*. **21**(12).

Object Management Group. (2017) Unified modelling language, version 2.5.1.

Pohjola, J. Å., Gómez-Londoño, A., Shaker, J. & Norrish, M. (2022) Kalas: A verified, end-to-end compiler for a choreographic language. 13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. pp. 27:1–27:18.

Scalas, A., Dardha, O., Hu, R. & Yoshida, N. (2017) A linear decomposition of multiparty sessions for safe distributed programming. 31st European Conference on Object-Oriented Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain. Schloss Dagstuhl - Leibniz-Zentrum für Informatik. pp. 24:1–24:31.

Scalas, A. & Yoshida, N. (2019) Less is more: Multiparty session types revisited. Principles of Programming Languages (POPL).

Shen, G., Kashiwa, S. & Kuper, L. (2023) Haschor: Functional choreographic programming for all (functional pearl). *CoRR*. **abs/2303.00924**.

The World Wide Web Consortium (W3C). (2004) Ws choreography model overview. Accessed January 29,2021.

Wadler, P. (2012) Propositions as sessions. International Conference on Functional Programming (ICFP).

Weisenburger, P. & Salvaneschi, G. (2020) Implementing a language for distributed systems: Choices and experiences with type level and macro programming in scala. *ASEP*. **4**(3), 17.

Zongyan, Q., Xiangpeng, Z., Chao, C. & Hongli, Y. (2007) Towards the theoretical foundation of choreography. The Web Conference (WWW).

1 Full PolyChorλ Typing Rules

$$[\textsc{TUnit}]\ \frac{\Theta;\Gamma \vdash v :: \mathsf{Proc}}{\Theta;\Gamma \vdash () \mathbin{@} v : () \mathbin{@} v} \qquad [\textsc{TInt}]\ \frac{\Theta;\Gamma \vdash v :: \mathsf{Proc}}{\Theta;\Gamma \vdash n \mathbin{@} v : \mathsf{Int} \mathbin{@} v}$$

$$[\textsc{TApp}]\ \frac{\Theta;\Gamma \vdash N : \tau_1 \to_\rho \tau_2 \qquad \Theta;\Gamma \vdash M : \tau_1}{\Theta;\Gamma \vdash N\ M : \tau_2}$$

$$[\textsc{TAbs}]\ \frac{\begin{array}{c}\Theta;\Gamma \vdash \tau_1 :: * \qquad \Theta;\Gamma' \vdash v :: \mathsf{Proc} \text{ for all } v \in \rho \\ \Theta \cap (\rho \cup \mathsf{ip}(\tau_1) \cup \mathsf{ip}(\tau_2) \cup \mathrm{ftv}(\tau_1) \cup \mathrm{ftv}(\tau_2)); \Gamma, x : \tau_1 \vdash M : \tau_2\end{array}}{\Theta;\Gamma \vdash \lambda\, x : \tau_1 . M : \tau_1 \to_\rho \tau_2}$$

$$[\textsc{TSel}]\ \frac{\Theta;\Gamma\vdash v_1 :: \mathsf{Proc} \qquad \Theta;\Gamma\vdash v_2 :: \mathsf{Proc} \qquad \Theta;\Gamma\vdash M:\tau}{\Theta;\Gamma\vdash \mathsf{select}_{v_1,v_2}\ \ell\ M:\tau}$$

$$[\textsc{TCom}]\ \frac{\begin{array}{c}\Theta;\Gamma\vdash \tau :: \mathsf{Proc}\Rightarrow *\\ \Theta;\Gamma\vdash v_1 :: \mathsf{Proc}\setminus(\mathsf{ip}(\tau)\cup \mathrm{ftv}(\tau)) \qquad \Theta;\Gamma\vdash v_2 :: \mathsf{Proc}\setminus(\mathsf{ip}(\tau)\cup \mathrm{ftv}(\tau))\end{array}}{\Theta;\Gamma\vdash \mathsf{com}^{\tau}_{v_1,v_2} : (\tau\ v_1 \to_{\varnothing} \tau\ v_2)}$$

$$[\textsc{TAppT}]\ \frac{\Theta;\Gamma\vdash M:\forall X::\mathsf{K}.\ \tau_1 \qquad \Theta;\Gamma\vdash \tau_2 :: \mathsf{K}}{\Theta;\Gamma\vdash M\ \tau_2 : \tau[X\mapsto \tau_2]}$$

$$[\textsc{TAbst}]\ \frac{\begin{array}{c}\Theta';\Gamma',X::\mathsf{K}\vdash M:\tau\\ \text{if } \exists \mathsf{K}',\rho.\ \mathsf{K}=\mathsf{K}'\setminus\rho \text{ then } \Gamma'=(\Gamma+X)\ \&\ \rho\setminus X \text{ else } \Gamma'=\Gamma+X\\ \text{if } \mathsf{K}=\mathsf{Proc} \text{ or } \exists\rho.\ \mathsf{K}=\mathsf{Proc}\setminus\rho \text{ then } \Theta'=\Theta,X \text{ else } \Theta'=\Theta\end{array}}{\Theta;\Gamma\vdash \Lambda X::\mathsf{K}.\ M:\forall X::\mathsf{K}.\ \tau}$$

$$[\textsc{TEq}]\ \frac{\Theta;\Gamma\vdash M:\tau_1 \qquad \tau_1\equiv\tau_2 \qquad \Theta;\Gamma\vdash \tau_2 :: *}{\Theta;\Gamma\vdash M:\tau_2}$$

$$[\textsc{TDefs}]\ \frac{\forall f\in \mathsf{domain}(D).\ f:\tau\in\Gamma \wedge \emptyset;\Gamma\vdash D(f):\tau}{\Theta;\Gamma\vdash D} \qquad [\textsc{TVar}]\ \frac{x:\tau\in\Gamma}{\Theta;\Sigma;\Gamma\vdash x:\tau}$$

$$[\textsc{TCase}]\ \frac{\Gamma\vdash N:\tau_1+\tau_2 \qquad \Theta;\Gamma,x:T_1\vdash M':\tau \qquad \Theta;\Gamma,x':T_2\vdash M'':\tau}{\Theta;\Gamma\vdash \mathsf{case}\ N\ \mathsf{of}\ \mathsf{inl}\ \ x\Rightarrow M';\ \ \mathsf{inr}\ \ x'\Rightarrow M'':\tau}$$

$$[\textsc{TFun}]\ \frac{f:\tau\in\Gamma}{\Theta;\Gamma\vdash f:\tau} \qquad [\textsc{TPair}]\ \frac{\Theta;\Gamma\vdash M:\tau_1 \qquad \Theta;\Gamma\vdash N:\tau_2}{\Theta;\Gamma\vdash (M,N):\tau_1\times\tau_2}$$

$$[\textsc{TProj1}]\ \frac{\Theta;\Gamma\vdash M:\tau_1\times\tau_2}{\Theta;\Gamma\vdash \mathsf{fst}\ M:\tau_1} \qquad [\textsc{TProj2}]\ \frac{\Theta;\Gamma\vdash M:\tau_1\times\tau_2}{\Theta;\Gamma\vdash \mathsf{snd}\ M:\tau_2}$$

$$[\textsc{TInl}]\ \frac{\Theta;\Sigma;\Gamma\vdash M:\tau_1}{\Theta;\Sigma;\Gamma\vdash \mathsf{inl}_M\ \tau_2:\tau_1+\tau_2} \qquad [\textsc{TInr}]\ \frac{\Theta;\Sigma;\Gamma\vdash M:\tau_2}{\Theta;\Sigma;\Gamma\vdash \mathsf{inr}_M\ \tau_2:\tau_1+\tau_2}$$

2 Full PolyChorλ Operational Semantics

$$[\textsc{AppAbs}]\ \frac{}{(\lambda\, x:\tau.\ M)\ V\to_D M[x\mapsto V]}$$

$$[\textsc{App1}]\ \frac{M_1\to_D M_2}{M_1\ N\to_D M_2\ N} \qquad [\textsc{App2}]\ \frac{N_1\to_D N_2}{V\ N_1\to_D V\ N_2}$$

$$[\textsc{AppTAbs}]\ \frac{\tau \equiv \nu}{(\Lambda X :: \mathsf{K}.\, M)\ \tau \rightarrow_D M[X \mapsto \nu]} \qquad [\textsc{MTApp1}]\ \frac{M_1 \rightarrow_D M_2}{M_1\ \tau \rightarrow_D M_2\ \tau}$$

$$[\textsc{MTApp2}]\ \frac{\tau_1 \rightarrow_D \tau_2}{V\ \tau_1 \rightarrow_D V\ \tau_2}$$

$$[\textsc{Inl}]\ \frac{M_1 \rightarrow_D M_2}{\mathsf{inl}_\tau\ M_1 \rightarrow_D \mathsf{inl}_\tau\ M_2} \qquad [\textsc{Inl}]\ \frac{M_1 \rightarrow_D M_2}{\mathsf{inr}_\tau\ M_1 \rightarrow_D \mathsf{inr}_\tau\ M_2}$$

$$[\textsc{Case}]\ \frac{N_1 \rightarrow_D N_2}{\mathsf{case}\ N_1\ \mathsf{of\ inl}\ \ x \Rightarrow M_1;\ \mathsf{inr}\ \ y \Rightarrow M_2 \rightarrow_D \mathsf{case}\ N_2\ \mathsf{of\ inl}\ \ x \Rightarrow M_1;\ \mathsf{inr}\ \ y \Rightarrow M_2}$$

$$[\textsc{CaseL}]\ \frac{}{\mathsf{case\ inl}_\tau\ V\ \mathsf{of\ inl}\ \ x \Rightarrow M_1;\ \mathsf{inr}\ \ y \Rightarrow M_2 \rightarrow_D M_1[x \mapsto V]}$$

$$[\textsc{CaseR}]\ \frac{}{\mathsf{case\ inr}_\tau\ V\ \mathsf{of\ inl}\ \ x \Rightarrow M_1;\ \mathsf{inr}\ \ y \Rightarrow M_2 \rightarrow_D M_1[x \mapsto V]}$$

$$[\textsc{Pair1}]\ \frac{M_1 \rightarrow_D M_2}{(M_1, N) \rightarrow_D (M_2, N)} \qquad [\textsc{Pair2}]\ \frac{N_1 \rightarrow_D N_2}{(V, N_1) \rightarrow_D (V, N_2)}$$

$$[\textsc{Fst}]\ \frac{M_1 \rightarrow_D M_2}{\mathsf{fst}\ M_1 \rightarrow_D \mathsf{fst}\ M_2} \qquad [\textsc{Snd}]\ \frac{M_1 \rightarrow_D M_2}{\mathsf{snd}\ M_1 \rightarrow_D \mathsf{snd}\ M_2}$$

$$[\textsc{Proj1}]\ \frac{}{\mathsf{fst}\ \ (V_1, V_2) \rightarrow_D V_1} \qquad [\textsc{Proj2}]\ \frac{}{\mathsf{snd}\ \ (V_1, V_2) \rightarrow_D V_2}$$

$$[\textsc{Def}]\ \frac{}{f \rightarrow_D D(f)}$$

$$[\textsc{Sel}]\ \frac{}{\mathsf{select}_{\mathsf{P},\mathsf{Q}}\ \ell\ M \rightarrow_D M} \qquad [\textsc{Com}]\ \frac{}{\mathsf{com}^{\tau}_{\mathsf{P},\mathsf{Q}}\ V \rightarrow_D V[\mathsf{P} \mapsto \mathsf{Q}]}$$

3 Proof of Theorem 5

In the foregoing we use L to denote local expressions and U to denote local values in order to make the proofs more readable, as we will be switching back and forth between layers a lot.

Before we can prove completeness, we need a few lemmas. First, we show that choreographic values always project to local values.

Lemma 3. *For any choreographic value V and process P, if $\Theta;\Gamma \vdash V : \tau$ then $[\![V]\!]_{\mathsf{P}}$ is either a value or undefined.*

Proof Straightforward from the projection rules. ■

We then prove the same for type values.

Lemma 4. *Given a type value v, if $\Theta;\Gamma \vdash v :: K$ then for any process P in $ip(v)$, $[\![v]\!]_P = v$.*

Proof Straightforward from the projection rules. ■

We then show that type values are projected to $\bot$ at uninvolved processes.

Lemma 5. *Given a type value $v \neq P$, for any process $Q \notin ip(v)$, $[\![v]\!]_Q = \bot$.*

Proof Straightforward from induction on v. ■

And similarly, we show that choreographic values project to $\bot$ at processes not involved in their type.

Lemma 6. *Given a value V, if $\Theta;\Gamma \vdash V : \tau$ then for any process $P \notin ip(\tau)$, we have $[\![V]\!]_P = \bot$ or $[\![V]\!]_P$ is undefined.*

Proof Follows from Lemmas 3 and 5 and the projection rules. ■

Finally, we show that equivalent types are projected to equivalent local types.

Lemma 7. *Given a closed type τ, if $\tau \equiv \tau'$ and $\Theta;\Gamma \vdash \tau :: K$, then for any process P, $[\![\tau]\!]_P \equiv_P [\![\tau']\!]_P$.*

Proof We prove this by structural induction on $\tau \equiv \tau'$. All but one case follow by simple induction.

The one interesting case is if $\tau = \lambda X.\, \mathsf{K}\tau_1 \;\; v$ and $\tau' = \tau_1[X := v]$. Then (1) if $\mathsf{K} = \mathsf{K}' \setminus (\{R\} \cup \rho)$ and $[\![\tau_1]\!]_{\mathsf{P}} = \bot$, we have $[\![\tau]\!]_{\mathsf{P}} = [\![\tau']\!]_{\mathsf{P}} = \bot$. (2) If $\mathsf{K} \in \{\mathsf{Proc}, \mathsf{Proc} \setminus \rho\}$ then $[\![\tau]\!]_{\mathsf{P}} = (\forall X.\, \mathsf{AmI}\; X \;?\; [\![\tau_1[X := \mathsf{P}]]\!]_{\mathsf{P}} \;\&\; [\![\tau_1]\!]_{\mathsf{P}}) \;\; [\![v]\!]_{\mathsf{P}}$ and $[\![\tau']\!]_{\mathsf{P}} = [\![\tau_1[X := v]]\!]_{\mathsf{P}}$. Since τ is a closed type and $\Theta;\Gamma \vdash v :: \mathsf{K}$, v must be a process. If $v = \mathsf{P}$ then obviously $(\forall X.\, \mathsf{AmI}\; X \;?\; [\![\tau_1[X := \mathsf{P}]]\!]_{\mathsf{P}} \;\&\; [\![\tau_1]\!]_{\mathsf{P}}) \;\; [\![v]\!]_{\mathsf{P}} \equiv_{\mathsf{P}} [\![\tau_1[X := v]]\!]_{\mathsf{P}}$. If $v \neq \mathsf{P}$ then $(\forall X.\, \mathsf{AmI}\; X \;?\; [\![\tau_1[X := \mathsf{P}]]\!]_{\mathsf{P}} \;\&\; [\![\tau_1]\!]_{\mathsf{P}}) \;\; [\![v]\!]_{\mathsf{P}} \equiv_{\mathsf{P}} [\![\tau_1]\!]_{\mathsf{P}} \; [X := [\![v]\!]_{\mathsf{P}}]$, but since v is a process Q, $[\![v]\!]_{\mathsf{P}} = \mathsf{Q}$ and $[\![X]\!]_{\mathsf{P}} = X$, and therefore we get $[\![\tau]\!]_{\mathsf{P}} \equiv_{\mathsf{P}} [\![\tau']\!]_{\mathsf{P}}$. And (3) otherwise we have $[\![\tau]\!]_{\mathsf{P}} = (\lambda X.\; [\![\tau_1]\!]_{\mathsf{P}}) \;\; [\![v]\!]_{\mathsf{P}}$ and $[\![\tau']\!]_{\mathsf{P}} = [\![\tau_1]\!]_{\mathsf{P}}[X := [\![v]\!]_{\mathsf{P}}]$. We therefore get $[\![\tau]\!]_{\mathsf{P}} \equiv_{\mathsf{P}} [\![\tau']\!]_{\mathsf{P}}$. ■

We also need to prove that performing a substitution before and after projection yield the same result.

Lemma 8. *Given a choreography M with a free variable x, a value V, and a type τ such that $\Theta;\Gamma \vdash \lambda\, x : \tau.\, M \;\; V : \tau'$ and $[\![\lambda\, x : \tau.\, M\; V]\!]$ is defined, we get $[\![M[x := V]]\!]_P = [\![M]\!]_P \; [x := [\![V]\!]_P]$.*

Proof If $\tau = \bot$ then by definition $[\![x]\!]_{\mathsf{P}} = \bot$ and by Lemma 6, $[\![V]\!]_{\mathsf{P}} = \bot$. If $\tau \neq \bot$ then $[\![x]\!]_{\mathsf{P}} = x$ and since we use α-conversion when substituting, we can guarantee that $\mathsf{typeof}(V) = \tau$ anywhere it gets substituted into M, meaning the projection will always be the same as it does not depend on context, only on syntax and type. We therefore get $[\![M[x := V]]\!]_{\mathsf{P}} = [\![M]\!]_{\mathsf{P}} \; [x := [\![V]\!]_{\mathsf{P}}]$. ■

We are now ready to prove completeness.

Proof [Proof of Theorem 5] We prove this by structural induction on $M \rightarrow_D M'$.

- Assume $M = (\lambda\, x : \tau.\, N)\ \ V$ and $M' = N[x := V]$. Then for any process P such that $[\![N]\!]_{\mathsf{P}} \neq \bot$ or $[\![\tau]\!]_{\mathsf{P}} \neq \bot$, we have $[\![M]\!]_{\mathsf{P}} = (\lambda\, x : [\![\tau]\!]_{\mathsf{P}}\, .\, [\![N]\!]_{\mathsf{P}})\ \ [\![V]\!]_{\mathsf{P}}$ and $[\![M']\!]_{\mathsf{P}} = [\![N]\!]_{\mathsf{P}}[x := [\![V]\!]_{\mathsf{P}}]$, and for any other P', we have $\mathsf{P}' \notin \mathsf{ip}(\mathsf{typeof}(V))$ and therefore $[\![M]\!]_{\mathsf{P}'} = [\![M']\!]_{\mathsf{P}'} = \bot$. We therefore get $\mathsf{P}[[\![M]\!]_{\mathsf{P}}] \xrightarrow{\tau}_{[\![D]\!]} [\![M']\!]_{\mathsf{P}}$ for all $\mathsf{P} \in \mathsf{ip}(\mathsf{typeof}(\lambda\, x : \tau.\, N))$ and define $\mathscr{N} = \prod_{\mathsf{P} \in \mathsf{ip}(\mathsf{typeof}(\lambda\, x:\tau.\, N))} \mathsf{P}[[\![M']\!]_{\mathsf{P}}] \mid \prod_{\mathsf{P}' \notin \mathsf{ip}(\mathsf{typeof}((\lambda\, x:\tau.\, N))} \mathsf{P}'[\bot]$ and the result follows.
- Assume $M = (\Lambda X :: \mathsf{K}.\, N)\ \tau$, $\tau \equiv v$, and $M' = N[X := v]$. Then if $K \in \{\mathsf{P},\ \mathsf{P} \setminus \rho\}$, for any process P, $[\![M]\!]_{\mathsf{P}} = (\Lambda X :: \mathsf{K}.\, \mathsf{AmI}\ X\ ?\ [\![N[t := \mathsf{P}]]\!]_{\mathsf{P}}\ \&\ [\![N]\!]_{\mathsf{P}})\ \ [\![\tau]\!]_{\mathsf{P}}$ and the result follows Lemmas 4 and 7, and Rules [NBabs], [NIamr], and [NIaml]. If $\mathsf{K} \notin \{\mathsf{Proc}, \mathsf{Proc} \setminus \rho\}$ then for any process P such that $[\![N]\!]_{\mathsf{P}} = \bot$ and $\mathsf{K} = \mathsf{K}' \setminus (\{\mathsf{P}\} \cup \rho)$, we have $[\![M]\!]_{\mathsf{P}} = [\![M']\!]_{\mathsf{P}} = \bot$, for any other P' we have $[\![M]\!]_{\mathsf{P}'} = (\Lambda X.\, [\![N]\!]_{\mathsf{P}'})\ [\![\tau]\!]_{\mathsf{P}'}$ and $[\![M']\!]_{\mathsf{P}'} = [\![N]\!]_{\mathsf{P}'}[X := [\![v]\!]_{\mathsf{P}'}]$. We therefore get $\mathsf{P}[[\![M]\!]_{\mathsf{P}}] \rightarrow^*_{[\![D]\!]} [\![M']\!]_{\mathsf{P}}$ for all P and the result follows.
- Assume $M = N\ \ M''$, $M' = N'\ \ M''$, and $N \rightarrow_D N'$. Then for any process P such that $[\![N]\!]_{\mathsf{P}} = [\![M'']\!]_{\mathsf{P}} = \bot$, by induction we have $[\![N']\!]_{\mathsf{P}} = \bot$, and therefore $[\![M]\!]_{\mathsf{P}} = [\![M']\!]_{\mathsf{P}} = \bot$. For any process P' such that $\mathsf{P}' \in \mathsf{ip}(\mathsf{typeof}(N))$ or $[\![N]\!]_{\mathsf{P}'} \neq \bot \neq [\![M'']\!]_{\mathsf{P}'}$, $[\![M]\!]_{\mathsf{P}'} = [\![N]\!]_{\mathsf{P}'}\ \ [\![M'']\!]_{\mathsf{P}'}$ and $[\![M']\!]_{\mathsf{P}'} = [\![N']\!]_{\mathsf{P}'}\ \ [\![M'']\!]_{\mathsf{P}'}$. For any other process P'' such that $[\![N]\!]_{\mathsf{P}''} = \bot$, by induction we get $[\![N']\!]_{\mathsf{P}''} = \bot$ and therefore $[\![M]\!]_{\mathsf{P}''} = [\![M']\!]_{\mathsf{P}''} = [\![M'']\!]_{\mathsf{P}''}$. For any other process P''' such that $[\![M'']\!]_{\mathsf{P}'''} = \bot$, we get $[\![M]\!]_{\mathsf{P}'''} = [\![N]\!]_{\mathsf{P}'''}$ and $[\![M']\!]_{\mathsf{P}'''} = [\![N']\!]_{\mathsf{P}'''}$. And by induction $[\![N]\!] \rightarrow^*_{[\![D]\!]} \mathscr{N}_N$ and $\mathscr{N}_N \sqsupseteq [\![N']\!]$. For any process P we therefore get $[\![N]\!]_{\mathsf{P}} \xrightarrow{\mu_0}_{[\![D]\!]} \xrightarrow{\mu_1}_{[\![D]\!]} \ldots L_{\mathsf{P}}$ for $L_{\mathsf{P}} \sqsupseteq [\![N']\!]_{\mathsf{P}}$ for some sequences of transitions $\xrightarrow{\mu_0}_{[\![D]\!]} \xrightarrow{\mu_1}_{[\![D]\!]} \ldots$, and from the network semantics we get

$$[\![M]\!] \rightarrow^*_{[\![D]\!]} \begin{array}{l} \prod_{[\![N]\!]_{\mathsf{P}} = [\![M'']\!]_{\mathsf{P}} = \bot} \mathsf{P}[\bot] \mid \prod_{\mathsf{P}' \in \mathsf{ip}(\mathsf{typeof}(N)) \text{ or } [\![N]\!]_{\mathsf{P}'} \neq \bot \neq [\![M'']\!]_{\mathsf{P}'}} \mathsf{P}'[L_{\mathsf{P}'}\ [\![M'']\!]_{\mathsf{P}'}] \\ \mid \prod_{[\![M]\!]_{\mathsf{P}''} = [\![M'']\!]_{\mathsf{P}''}} \mathsf{P}''[[\![M'']\!]_{\mathsf{P}''}] \mid \prod_{[\![M]\!]_{\mathsf{P}'''} = [\![N]\!]_{\mathsf{P}'''}} \mathsf{P}''[L_{\mathsf{P}''}] \end{array} = \mathscr{N}$$

and $M \rightarrow_D N'\ \ M''$. And since $[\![N]\!] \rightarrow^*_{[\![D]\!]} \mathscr{N}'$ and $[\![N']\!] \rightarrow^*_{[\![D]\!]} \mathscr{N}'_N$, we know these sequences of transitions can synchronise when necessary, and if $[\![N]\!]_{\mathsf{P}''''} \neq [\![N']\!]_{\mathsf{P}''''} = \bot$ then we can do the extra application to get rid of this unit.
- Assume $M = V\ \ N$, $M' = V\ \ N'$, and $N \rightarrow_D N'$. This is similar to the previous case, using Lemma 3 to ensure every process can use Rule [NApp2].
- Assume $M = \mathsf{case}\ N\ \mathsf{of}\ \ \mathsf{inl}\ \ x \Rightarrow N';\ \ \mathsf{inr}\ \ x' \Rightarrow N''$, $M' = \mathsf{case}\ M''\ \mathsf{of}\ \ \mathsf{inl}\ \ x \Rightarrow N';\ \ \mathsf{inr}\ \ x \Rightarrow N''$, and $N \rightarrow_D M''$. Then for any process P such that $\mathsf{P} \in \mathsf{ip}(\mathsf{typeof}(N))$, we have $[\![M]\!]_{\mathsf{P}} = \mathsf{case}\ [\![N]\!]_{\mathsf{P}}\ \mathsf{of\ inl}\ \ x \Rightarrow [\![N']\!]_{\mathsf{P}};\ \mathsf{inr}\ \ x' \Rightarrow [\![N'']\!]_{\mathsf{P}}$ and $[\![M']\!]_{\mathsf{P}} = \mathsf{case}\ [\![M'']\!]_{\mathsf{P}}\ \mathsf{of\ inl}\ \ x \Rightarrow [\![N']\!]_{\mathsf{P}};\ \ \mathsf{inr}\ \ x' \Rightarrow [\![N'']\!]_{\mathsf{P}}$. For any other process P' such that $[\![N]\!]_{\mathsf{P}'} = [\![N']\!]_{\mathsf{P}'} = [\![N'']\!]_{\mathsf{P}'} = \bot$, by induction we get $[\![M'']\!]_{\mathsf{P}'} = \bot$, and therefore $[\![M]\!]_{\mathsf{P}'} = [\![M']\!]_{\mathsf{P}'} = \bot$. For any other process P'' such that $[\![N]\!]_{\mathsf{P}''} = \bot$, we get $[\![M]\!]_{\mathsf{P}''} = [\![M']\!]_{\mathsf{P}''} = [\![N']\!]_{\mathsf{P}''} \sqcup [\![N'']\!]_{\mathsf{P}''}$. For any other processes P''' such that $[\![N']\!]_{\mathsf{P}'''} = [\![N'']\!]_{\mathsf{P}'''} = \bot$, we have $[\![M]\!]_{\mathsf{P}'''} = [\![N]\!]_{\mathsf{P}'''}$ and $[\![M']\!]_{\mathsf{P}'''} = [\![M'']\!]_{\mathsf{P}'''}$. For any other process P'''', we have $[\![M]\!]_{\mathsf{P}''''} = (\lambda x : \bot.[\![N']\!]_{\mathsf{P}''''} \sqcup [\![N'']\!]_{\mathsf{P}''''})\ \ [\![N]\!]_{\mathsf{P}''''}$ and $[\![M']\!]_{\mathsf{P}''''} =$

$(\lambda x.[\![N']\!]_{\mathsf{P}''''} \sqcup [\![N'']\!]_{\mathsf{P}''''})\ [\![M'']\!]_{\mathsf{P}''''}$ for $x \notin \mathrm{fv}(N') \cup \mathrm{fv}(N'')$. The rest follows by simple induction similar to the second case.

- Assume $M = \mathsf{case\ inl}_\tau\ V\ \mathsf{of\ inl}\ \ x \Rightarrow N;\ \ \mathsf{inr}\ \ x' \Rightarrow N'$ and $M' = N[x := V]$. Then for any process $\mathsf{P} \in \mathsf{ip}(\mathsf{typeof}(\mathsf{inl}_\tau\ V))$, we have $[\![M]\!]_{\mathsf{P}} = \mathsf{case\ inl}_{[\![\tau]\!]_{\mathsf{P}}}\ [\![V]\!]_{\mathsf{P}}\ \mathsf{of\ inl}\ \ x \Rightarrow [\![N]\!]_{\mathsf{P}};\ \ \mathsf{inr}\ \ x' \Rightarrow [\![N']\!]_{\mathsf{P}}$ and $[\![M']\!]_{\mathsf{P}} = [\![N[x := [\![V]\!]_{\mathsf{P}}]]\!]_{\mathsf{P}}$. By Lemma 6, $[\![N[x := [\![V]\!]_{\mathsf{P}}]]\!]_{\mathsf{P}} = [\![N]\!]_{\mathsf{P}}[x := [\![V]\!]_{\mathsf{P}}]$. For any other process $\mathsf{P}' \notin \mathsf{ip}(\mathsf{typeof}(\mathsf{inl}_\tau\ V))$, $[\![\mathsf{inl}_V\ \tau\mathsf{P}']\!]_= \bot$, and therefore $[\![M]\!]_{\mathsf{P}'} = [\![N]\!]_{\mathsf{P}'} \sqcup [\![N']\!]_{\mathsf{P}'} \sqsupseteq [\![N]\!]_{\mathsf{P}'} = [\![M']\!]_{\mathsf{P}'}$. The result follows.
- Assume $M = \mathsf{case\ inr}_\tau\ V\ \mathsf{of\ inl}\ \ x \Rightarrow N;\ \ \mathsf{inr}\ \ x' \Rightarrow N'$ and $M' = N'[x' := V]$. This case is similar to the previous.
- Assume $M = \mathsf{com}^\tau_{\mathsf{P},\mathsf{Q}}\ V$ and $M' = V[\mathsf{Q} := \mathsf{P}]$. Then if $\mathsf{Q} \neq \mathsf{P}$, $[\![M]\!]_{\mathsf{P}} = \mathsf{recv}_{\mathsf{Q}}\ \ [\![V]\!]_{\mathsf{P}}$, $[\![M']\!]_{\mathsf{P}} = [\![V[\mathsf{Q} := \mathsf{P}]]\!]_{\mathsf{P}} = [\![V]\!]_{\mathsf{Q}}[\mathsf{Q} := \mathsf{P}]$, $[\![M]\!]_{\mathsf{Q}} = \mathsf{send}_{\mathsf{P}}\ \ [\![V]\!]_{\mathsf{Q}}$, $[\![M']\!]_{\mathsf{Q}} = [\![V[\mathsf{Q} := \mathsf{P}]]\!]_{\mathsf{Q}} = [\![V]\!]_{\mathsf{P}}[\mathsf{Q} := \mathsf{P}]$, and for any P' such that $[\![\tau]\!]_{\mathsf{P}'} \neq \bot$, we have $[\![M]\!]_{\mathsf{P}'} = \mathsf{sub}[\mathsf{Q} \mapsto \mathsf{P}]\ \ [\![V]\!]_{\mathsf{P}'}$ and $[\![M']\!]_{\mathsf{P}'} = [\![V[\mathsf{Q} := \mathsf{P}]]\!]_{\mathsf{P}'} = [\![V]\!]_{\mathsf{P}'}[\mathsf{Q} := \mathsf{P}]$, and for any other P'', $[\![M]\!]_{\mathsf{P}''} = [\![M']\!]_{\mathsf{P}''} = \bot$. We therefore get $[\![M]\!]_{\mathsf{P}} \xrightarrow{\mathsf{recv}_{\mathsf{Q}}[\![V]\!]_{\mathsf{Q}}[\mathsf{Q}:=\mathsf{P}]\ \ [\![V]\!]_{\mathsf{P}}}_{[\![D]\!]} [\![M']\!]_{\mathsf{P}}$, $[\![M]\!]_{\mathsf{Q}} \xrightarrow{\mathsf{send}_{\mathsf{P}}[\![V]\!]_{\mathsf{Q}}\ \ [\![V]\!]_{\mathsf{P}}[\mathsf{Q}:=\mathsf{P}]}_{[\![D]\!]} [\![M']\!]_{\mathsf{Q}}$, and $[\![M]\!]_{\mathsf{P}'} \xrightarrow{\tau}_{[\![D]\!]} [\![M']\!]_{\mathsf{P}'}$. We define $\mathscr{N} = [\![M']\!]$ and the result follows. If $\mathsf{Q} = \mathsf{P}$, then $[\![M]\!]_{\mathsf{P}} = (\lambda x.x)\ \ [\![V]\!]_{\mathsf{P}}$ and $[\![M']\!]_{\mathsf{P}} = [\![V]\!]_{\mathsf{P}}$ and $\mathscr{N} = [\![M']\!]$ and the result follows.
- Assume $M = \mathsf{select}_{\mathsf{Q},\mathsf{P}}\ \ell\ M'$. Then $[\![M]\!]_{\mathsf{Q}} = \oplus_{\mathsf{P}}\ \ell\ [\![M']\!]_{\mathsf{Q}}$, $[\![M]\!]_{\mathsf{P}} = \&_{\mathsf{Q}}\ \ \{\ell : [\![M']\!]_{\mathsf{P}}\}$, and for any $\mathsf{P}' \notin \{\mathsf{Q}, \mathsf{P}\}$, $[\![M]\!]_{\mathsf{P}'} = [\![M']\!]_{\mathsf{P}'}$. We therefore get $[\![M]\!] \xrightarrow{\tau_{\mathsf{P},\mathsf{Q}}}_{[\![D]\!]} [\![M]\!] \setminus \{\mathsf{P}, \mathsf{Q}\} \mid \mathsf{P}[[\![M']\!]_{\mathsf{P}}] \mid \mathsf{Q}[[\![M']\!]_{\mathsf{Q}}]$ and the result follows.
- Assume $M = (N, N')$, $N \to_D N''$, and $M' = (N'', N')$. Then the result follows from simple induction.
- Assume $M = (V, N)$, $N \to_D N'$, and $M' = (V, N')$. Then the result follows from simple induction.
- Assume $M = \mathsf{fst}\ (V, V')$ and $M' = V$. Then for any process P such that $[\![V]\!]_{\mathsf{P}} \neq \bot$ or $[\![V]\!]_{\mathsf{P}} \neq \bot$, $[\![M]\!]_{\mathsf{P}} = \mathsf{fst}\ ([\![M']\!]_{\mathsf{P}}, [\![V']\!]_{\mathsf{P}})$ and for any other process $\mathsf{P}' \notin \mathsf{ip}(\mathsf{typeof}((M', V')))$, we have $[\![M]\!]_{\mathsf{P}'} = \bot$ and $[\![M']\!]_{\mathsf{P}'} = \bot$. We define $\mathscr{N} = [\![M']\!]$ and the result follows.
- Assume $M = \mathsf{snd}\ (V, V')$ and $M' = V'$. Then the case is similar to the previous.
- Assume $M = f$ and $M' = D(f)$. Then the result follows from the definition of $[\![D]\!]$.

■

4 Proof of Theorem 6

As with completeness, we need some ancillary lemmas before we can prove soundness. For this, we need a notion of removing processes from a network.

Definition 3. Given a network $\mathscr{N} = \prod_{\mathsf{P} \in \rho} \mathsf{P}[L_{\mathsf{P}}]$, we have $\mathscr{N} \setminus \rho' = \prod_{\mathsf{P} \in (\rho \setminus \rho')} \mathsf{P}[L_{\mathsf{P}}]$.

First we show that actions in a network do not affect the roles not mentioned in the transition label.

Lemma 9. *For any process* P *and network* $\mathcal{N}$, *if* $\mathcal{N} \xrightarrow{\tau_{\mathscr{P}}}_d \mathcal{N}'$ *and* $\mathsf{P} \notin \mathscr{P}$ *then* $\mathcal{N}(\mathsf{P}) = \mathcal{N}'(\mathsf{P})$.

Proof Straightforward from the network semantics. ■

Then we show that removing processes from a network does not prevent it from performing actions involving different processes.

Lemma 10. *For any set of processes* ρ *and network* $\mathcal{N}$, *if* $\mathcal{N} \xrightarrow{\tau_{\mathscr{P}}} \mathcal{N}'$ *and* $\mathscr{P} \cap \rho = \emptyset$ *then* $\mathcal{N} \setminus \rho \xrightarrow{\tau_{\mathscr{P}}}_d \mathcal{N}' \setminus \rho$.

Proof Straightforward from the network semantics. ■

We finally show that if the projection of a choreographic type is equivalent to a local type value, then the original choreographic type is equivalent to a choreographic type value.

Lemma 11. *Given a closed type* τ_1 *and process* P, *if* $\Theta; \Gamma \vdash \tau_1 :: K$ *and* $[\![\tau_1]\!]_{\mathsf{P}} \equiv_{\mathsf{P}} v$, *then there exist a type* v' *such that:* $\tau_1 \equiv v'$ *and* $[\![v']\!]_{\mathsf{P}} = v$.

Proof We prove this by structural induction on τ_1. All but one case follows from simple induction.

Assume $\tau_1 = \tau_2\ \tau_3$. Then if $[\![\tau_2]\!]_{\mathsf{P}} = [\![\tau_2]\!]_{\mathsf{P}} = \bot$, we have $[\![\tau_1]\!]_{\mathsf{P}} = \bot$ and therefore $v = \bot = v'$. Otherwise, if $[\![\tau_3]\!]_{\mathsf{P}'} = \bot$ and $\mathsf{kindof}(\tau_3) = \mathsf{K} \setminus (\{\mathsf{P}\} \cup \rho)$, we get $[\![\tau_1]\!]_{\mathsf{P}} = [\![\tau_2]\!]_{\mathsf{P}}$ and the result follows from induction. Otherwise if $[\![\tau_2]\!]_{\mathsf{P}} = \bot$, we get $[\![\tau_1]\!]_{\mathsf{P}} = [\![\tau_3]\!]_{\mathsf{P}}$ and the result follows from induction. Otherwise, we get $[\![\tau]\!]_{\mathsf{P}} = [\![\tau_2]\!]_{\mathsf{P}}\ [\![\tau_3]\!]_{\mathsf{P}}$. By induction, $[\![\tau_2]\!]_{\mathsf{P}} \equiv_{\mathsf{P}} v_2$ and there exists v_2' such that $\tau_2 \equiv v_2'$ and $[\![v_2']\!]_{\mathsf{P}} = v_2$ and $[\![\tau_3]\!]_{\mathsf{P}} \equiv_{\mathsf{P}} v_3$ and there exists v_3' such that $\tau_3 \equiv v_3'$ and $[\![v_3']\!]_{\mathsf{P}} = v_3$. Because τ_1 is kindable, we have a kind K' such that $\Theta; \Gamma \vdash \tau_2 :: \mathsf{K}' \Rightarrow \mathsf{K}$ and $\Theta; \Gamma \vdash \tau_3 :: \mathsf{K}'$. This means that $v_2' = \lambda X :: \mathsf{K}'.\ v_4$ and if $\mathsf{K}' \in \{\mathsf{Proc}, \mathsf{Proc} \setminus \rho\}$ then $v_2 = \lambda X.\ \mathsf{AmI}\ X\ ?\ [\![v_4[X := \mathsf{P}]]\!]_{\mathsf{P}}\ \&\ [\![v_4]\!]_{\mathsf{P}}$, otherwise $v_2 = \lambda X.\ [\![v_4]\!]_{\mathsf{P}}$. We then get $v \equiv_{\mathsf{P}} [\![v_4'[X := v_3]]\!]_{\mathsf{P}}$ and $v' \equiv v_4'[X := v_3']$, and since X and v_3' are both base types, so are $[\![v_4'[X := v_3]]\!]_{\mathsf{P}}$ and $v_4'[X := v_3']$. ■

We are then ready to prove soundness.

Proof [Proof of Theorem 6] We prove this by structural induction on M.

- Assume $M = V$. Then for any process P, $[\![M]\!]_{\mathsf{P}} = U$, and therefore $[\![M]\!] \stackrel{\tau_{\mathscr{P}}}{\not\longrightarrow}$.
- Assume $M = N_1\ N_2$. Then for any process P such that $[\![N_1]\!]_{\mathsf{P}} = [\![N_2]\!]_{\mathsf{P}} = \bot$, we have $[\![M]\!]_{\mathsf{P}} = \bot$. For any process P' such that $\mathsf{P}' \in \mathsf{ip}(\mathsf{typeof}(N_1))$ or $[\![N_1]\!]_{\mathsf{P}'} \neq \bot \neq [\![N_2]\!]_{\mathsf{P}'}$, $[\![M]\!]_{\mathsf{P}'} = [\![N_1]\!]_{\mathsf{P}'}\ [\![N_2]\!]_{\mathsf{P}'}$. For any other process P'' such that $[\![N_2]\!]_{\mathsf{P}''} = \bot$, we get $[\![M]\!]_{\mathsf{P}''} = [\![N_1]\!]_{\mathsf{P}''}$. For any other process P''', we get $[\![M]\!]_{\mathsf{P}'''} = [\![N_2]\!]_{\mathsf{P}'''}$. We then have 2 cases.
  - Assume $N_2 = V$. Then $[\![N_2]\!]_{\mathsf{P}} = U$ by Lemma 3, and for any P' such that $\mathsf{P}' \notin \mathsf{ip}(\mathsf{typeof}(N_2)) \subseteq \mathsf{ip}(\mathsf{typeof}(N_1))$, by Lemma 6, $[\![N_2]\!]_{\mathsf{P}'} = \bot$ and therefore $[\![M]\!]_{\mathsf{P}'} = [\![N_1]\!]_{\mathsf{P}'}$, and we have 5 cases.

- $*$ Assume $N_1 = \lambda\, x : \tau.\, N_3$. Then for any process P such that $[\![N_3]\!]_{\mathsf{P}} \neq \bot$ or $[\![\tau]\!]_{\mathsf{P}} \neq \bot$, $[\![N_1]\!]_{\mathsf{P}} = \lambda\, x : [\![\tau]\!]_{\mathsf{P}}\, .\, [\![N_3]\!]_{\mathsf{P}}$. And for any other process, $[\![N_1]\!]_{\mathsf{P}} = \bot$. The only transition available at any process, would then use Rule [NAbsApp]. This means for any transition $M \xrightarrow{\tau_{\mathscr{P}}}$, there exists P'' such that $\mathscr{P} = \mathsf{P}''$. We then get $[\![M]\!] \xrightarrow{\tau_{\mathscr{P}}} [\![M]\!] \setminus \{\mathsf{P}''\} \mid \mathsf{P}''[[\![N_3]\!]_{\mathsf{P}''}[x := [\![N_2]\!]_{\mathsf{P}''}]]$. We say that $M' = N_3[x := N_2]$ and the result follows from using Rule [NAbsApp] in every process P such that $[\![M]\!]_{\mathsf{P}} \neq \bot$ and induction.
- $*$ Assume $N_1 = \mathsf{com}^{\tau}_{\mathsf{Q},\mathsf{P}}$. Then if $\mathsf{Q} \neq \mathsf{P}$, $[\![M]\!]_{\mathsf{Q}} = \mathsf{send}_{\mathsf{P}}\ [\![N_2]\!]_{\mathsf{Q}}$, $[\![M]\!]_{\mathsf{P}} = \mathsf{recv}_{\mathsf{P}}\ \bot$, for any $\mathsf{P}' \in \mathsf{ip}(\tau)$, $[\![M]\!]_{\mathsf{P}'} = \mathsf{sub}[\mathsf{Q} \mapsto \mathsf{P}]\ [\![V]\!]_{\mathsf{P}'}$, and for any other process P'', $[\![N_1]\!]_{\mathsf{P}''} = \bot = [\![M]\!]_{\mathsf{P}''}$. And if $\mathsf{Q} = \mathsf{P}$ then $[\![N_1]\!]_{\mathsf{P}} = \lambda x.x$.
  If $\mathscr{P} = \mathsf{Q}, \mathsf{P}$ then $\mathscr{N} = [\![M]\!] \setminus \{\mathsf{Q}, \mathsf{P}\} \mid \mathsf{Q}[[\![N_2]\!]_{\mathsf{P}}\, [\mathsf{Q} := \mathsf{P}]] \mid \mathsf{P}[[\![N_2]\!]_{\mathsf{Q}}[\mathsf{Q} := \mathsf{P}]]$. Because $[\![N_2]\!]_{\mathsf{P}} = \bot$ and $[\![N_2]\!]_{\mathsf{Q}} = U$, $N_2 = V$. Therefore $M \xrightarrow{\mathscr{P}}_D V[\mathsf{Q} := \mathsf{P}]$ and for any $\mathsf{P}' \in \mathsf{ip}(\tau)$, by Rule [NSub], $\mathscr{N}(\mathsf{P}') \xrightarrow{\tau}_{[\![D]\!]} [\![V[\mathsf{Q} := \mathsf{P}]]\!]_{\mathsf{P}'}$ and the result follows from induction.
  If $\mathscr{P} = \mathsf{P}$ then either $\mathsf{Q} = \mathsf{P}$ or $[\![N_1]\!]_{\mathsf{P}} = \mathsf{sub}[\mathsf{Q} \mapsto \mathsf{P}]$. If $\mathsf{Q} = \mathsf{P}$ then $\mathscr{N} = [\![M]\!] \setminus \{\mathsf{P}\} \mid \mathsf{P}[[\![N_2]\!]_{\mathsf{P}}]$ and the rest is similar to above. If $[\![N_1]\!]_{\mathsf{P}} = \mathsf{sub}[\mathsf{Q} \mapsto \mathsf{P}]$ then the case is similar to one of the other two.
- $*$ Otherwise, $N_1 \neq V$ and either $\mathscr{P} = \mathsf{P}$ or $\mathscr{P} = \mathsf{P}, \mathsf{Q}$.
  If $\mathscr{P} = \mathsf{P}$ then either $[\![N_1]\!]_{\mathsf{P}} \xrightarrow{\tau}_{[\![D]\!]_{\mathsf{P}}} L$ and $\mathsf{P} \in \mathsf{ip}(\mathsf{typeof}(N_1))$, $\mathscr{N} = [\![M]\!] \setminus \{\mathsf{P}\} \mid \mathsf{P}[L\ [\![N_2]\!]_{\mathsf{P}}]$. We therefore have $[\![N_1]\!] \xrightarrow{\tau_{\mathsf{P}}}_{[\![D]\!]_{\mathsf{P}}} [\![N_1]\!] \setminus \{\mathsf{P}\} \mid \mathsf{P}[L]$, and by induction, $N_1 \to^*_D N_1'$ such that $[\![N_1]\!] \setminus \{\mathsf{P}\} \mid \mathsf{P}[L] \to^*_D \mathscr{N}_1 \sqsupseteq [\![N_1']\!]$. Since all these transitions can be propagated past N_2 in the network and $[\![N_2]\!]_{\mathsf{P}'}$ in any process P' involved, we get the result for $M' = N_1'\ N_2$.
  If $\mathscr{P} = \mathsf{P}, \mathsf{Q}$ then the case is similar.

– If $N_2 \neq V$ then we have 2 cases.
  - $*$ If $\mathscr{P} = \mathsf{P}$ then either $[\![N_1]\!]_{\mathsf{P}} \xrightarrow{\tau}_{[\![D]\!]_{\mathsf{P}}} L$ or $[\![N_1]\!]_{\mathsf{P}} = U$ and $[\![N_2]\!]_{\mathsf{P}} \xrightarrow{\tau}_{[\![D]\!]_{\mathsf{P}}} L$ and the case is similar to the previous.
  - $*$ If $\mathscr{P} = \mathsf{Q}, \mathsf{P}$ then there exists U such that either $[\![N_1]\!]_{\mathsf{Q}} \xrightarrow{\mathsf{send}_{\mathsf{P}}\ U}_{[\![D]\!]_{\mathsf{Q}}} L_{\mathsf{Q}}$ or $[\![N_2]\!]_{\mathsf{Q}} \xrightarrow{\mathsf{send}_{\mathsf{P}}\ U}_{[\![D]\!]_{\mathsf{Q}}} L_{\mathsf{Q}}$ and $[\![N_1]\!]_{\mathsf{P}} \xrightarrow{\mathsf{recv}_{\mathsf{Q}}\ U[\mathsf{Q}:=\mathsf{P}]}_{[\![D]\!]_{\mathsf{P}}} L_{\mathsf{P}}$ or $[\![N_2]\!]_{\mathsf{P}} \xrightarrow{\mathsf{recv}_{\mathsf{Q}}\ U[\mathsf{Q}:=\mathsf{P}]}_{[\![D]\!]_{\mathsf{P}}} L_{\mathsf{P}}$.
    If $[\![N_1]\!]_{\mathsf{Q}} \xrightarrow{\mathsf{send}_{\mathsf{P}}\ U}_{[\![D]\!]_{\mathsf{Q}}} L_{\mathsf{Q}}$ then $[\![N_1]\!]_{\mathsf{Q}} \neq U'$ and therefore $[\![N_1]\!]_{\mathsf{P}} \xrightarrow{\mathsf{recv}_{\mathsf{Q}}\ U[\mathsf{Q}:=\mathsf{P}]}_{[\![D]\!]_{\mathsf{P}}} L_{\mathsf{P}}$ and the case is similar to the previous. If $[\![N_2]\!]_{\mathsf{Q}} \xrightarrow{\mathsf{send}_{\mathsf{P}}\ U}_{[\![D]\!]_{\mathsf{Q}}} L_{\mathsf{Q}}$ then $[\![N_1]\!]_{\mathsf{Q}} = U'$, and therefore $[\![N_2]\!]_{\mathsf{P}} \xrightarrow{\mathsf{recv}_{\mathsf{Q}}\ U[\mathsf{Q}:=\mathsf{P}]}_{[\![D]\!]_{\mathsf{P}}} L_{\mathsf{P}}$ and the case is similar to the previous.

- Assume $M = N\ \tau$. Then for any process P such that $[\![N]\!]_{\mathsf{P}} = [\![\tau]\!]_{\mathsf{P}} = \bot$, we have $[\![M]\!]_{\mathsf{P}} = \bot$. For any process P' such that $[\![\tau]\!]_{\mathsf{P}'} = \bot$ and $\mathsf{kindof}(\tau) = \mathsf{K} \setminus (\{\mathsf{P}\} \cup \rho)$, $[\![M]\!]_{\mathsf{P}'} = [\![N]\!]_{\mathsf{P}'}$. For any other process P'' such that $[\![\tau]\!]_{\mathsf{P}''} = \bot$, we get $[\![M]\!]_{\mathsf{P}''} = [\![N]\!]_{\mathsf{P}''}$. For any other process P''', we get $[\![M]\!]_{\mathsf{P}'''} = [\![N]\!]_{\mathsf{P}'''}\ \ [\![\tau]\!]_{\mathsf{P}'''}$. This case is similar to the previous unless $N = \Lambda\, X :: \mathsf{K}.\, N'$.
  If $N = \Lambda\, X :: \mathsf{K}.\, N'$ and $\tau \equiv \nu$ then we have two cases. Either $\mathsf{K} \in \{\mathsf{Proc}, \mathsf{Proc} \setminus \rho\}$ or not. If $\mathsf{K} \in \{\mathsf{Proc}, \mathsf{Proc} \setminus \rho\}$ then for any P', $[\![M]\!]_{\mathsf{P}'} = \Lambda X.\, \mathsf{AmI}\quad X\ ?$

$[\![N'[t := \mathsf{P}']]\!]_{\mathsf{P}} \,\&\, [\![N']\!]_{\mathsf{P}'}\ [\![v]\!]_{\mathsf{P}'}$. As $[\![v]\!]_{\mathsf{P}'} = \mathsf{P}$ for some P, the only available transition is using Rule [NBabs], and we therefore get $\mathscr{P} = \mathsf{P}''$ for some P'' and $\mathscr{N} = [\![M]\!] \setminus \{\mathsf{P}''\} \mid \mathsf{P}''[\mathsf{Aml}\ \ \mathsf{P}\ ?\ [\![N'[X := \mathsf{P}'']]\!]_{\mathsf{P}''} \,\&\, [\![N']\!]_{\mathsf{P}''}]$. We then define $M' = N'[X := v]$ and see that the result follows form using Rules [NIamr] and [NProam] on P'' if $\mathsf{P}'' = \mathsf{P}$ and otherwise using Rules [NIaml] and [NProam], at all other processes using Rule [NBAbs] and then either Rules [NIamr] and [NProam] or Rules [NIaml] and [NProam] and the result follows from induction.

If $\mathsf{K} \notin \{\mathsf{Proc}, \mathsf{Proc} \setminus \rho\}$ then the case is similar to $N_1 = \lambda\, X : \tau.\, N_3$ above.

- Assume $M = \mathsf{fst}\, N$. Then either $N \neq V$ and the result follows from induction, or $N = (V, V')$ and for any process $\mathsf{P} \in \mathsf{ip}(\mathsf{typeof}((V, V')))$, $[\![M]\!]_{\mathsf{P}} = \mathsf{fst}\ ([\![V]\!]_{\mathsf{P}}, [\![V']\!]_{\mathsf{P}})$ and for any other process $\mathsf{P}' \notin \mathsf{ip}(\mathsf{typeof}((V, V')))$, by Theorem 6 we have $[\![M]\!]_{\mathsf{P}'} = [\![N]\!]_{\mathsf{P}'} = \bot$, and therefore $[\![M]\!]_{\mathsf{P}'} \not\to_{[\![D]\!]_{\mathsf{P}'}}$.

  If $\mathscr{P} = \mathsf{P} \in \mathsf{ip}(\mathsf{typeof}((V, V')))$ then $\mathscr{N} = [\![M]\!] \setminus \{\mathsf{P}\} \mid \mathsf{P}[[\![V]\!]_{\mathsf{P}}]$ and $M \xrightarrow{\tau_{\mathscr{P}}}_D V$. The result follows by use of Rule [NProj1] and Theorem 6 and induction.
- Assume $N_1 = \mathsf{snd}\, N_2$. This case is similar to the previous.
- Assume $M = (M_1, M_2)$. Then the result follows from simple induction.
- Assume $M = \mathsf{case}\ N\ \mathsf{of\ inl}\ \ x \Rightarrow N';\ \mathsf{inr}\ \ x' \Rightarrow N''$. Then for any process P such that $\mathsf{P} \in \mathsf{ip}(\mathsf{typeof}(N))$, we have $[\![M]\!]_{\mathsf{P}} = \mathsf{case}\ [\![N]\!]_{\mathsf{P}}\ \mathsf{of\ inl}\ \ x \Rightarrow [\![N']\!]_{\mathsf{P}};\ \mathsf{inr}\ \ x' \Rightarrow [\![N'']\!]_{\mathsf{P}}$. For any other process P' such that $[\![N]\!]_{\mathsf{P}'} = [\![N']\!]_{\mathsf{P}'} = [\![N'']\!]_{\mathsf{P}'} = \bot$, $[\![M]\!]_{\mathsf{P}'} = \bot$. For any other process P'' such that $[\![N]\!]_{\mathsf{P}''} = \bot$, we get $[\![M]\!]_{\mathsf{P}''} = [\![N']\!]_{\mathsf{P}''} \sqcup [\![N'']\!]_{\mathsf{P}''}$. For any other processes P''' such that $[\![N']\!]_{\mathsf{P}'''} = [\![N'']\!]_{\mathsf{P}'''} = \bot$, we have $[\![M]\!]_{\mathsf{P}'''} = [\![N]\!]_{\mathsf{P}'''}$. For any other process P'''', we have $[\![M]\!]_{\mathsf{P}''''} = (\lambda x : \bot. [\![N']\!]_{\mathsf{P}''''} \sqcup [\![N'']\!]_{\mathsf{P}''''})\ [\![N]\!]_{\mathsf{P}''''}$. We have two cases.

  – Assume $\mathscr{P} = \mathsf{P} \in \mathsf{ip}(\mathsf{typeof}(N))$. Then we have three cases.

    ∗ Assume $N = \mathsf{inl}_\tau\, V$. Then $[\![N]\!]_{\mathsf{P}} = \mathsf{inl}_{[\![\tau]\!]_{\mathsf{P}}}\ [\![V]\!]_{\mathsf{P}}$ and $\mathscr{N} = [\![M]\!] \setminus \{\mathsf{P}\} \mid \mathsf{P}[[\![N'[x := [\![V]\!]_{\mathsf{P}}]]\!]_{\mathsf{P}}]$. We define $M' = N'$ and since $[\![N']\!]_{\mathsf{P}'} \sqsupseteq [\![N']\!]_{\mathsf{P}'} \sqcup [\![N'']\!]_{\mathsf{P}'}$ the result follows from using Rules [NAbsApp] and [NCaseI] and induction.

    ∗ Assume $N = \mathsf{inr}_\tau\, V$. Then the case is similar to the previous.

    ∗ Otherwise, we use Rule [NCase] and we have a transition $[\![N]\!]_{\mathsf{P}} \xrightarrow{\tau}_{[\![D]\!]_{\mathsf{P}}} L$ such that

    $$\mathscr{N} = [\![M]\!] \setminus \{\mathsf{P}\} \mid \mathsf{P}[\mathsf{case}\ L\ \mathsf{of\ inl}\ \ x \Rightarrow [\![N']\!]_{\mathsf{P}};\ \mathsf{inr}\ \ x' \Rightarrow [\![N'']\!]_{\mathsf{P}}]$$

    and the result follows from induction similar to the last application case.

  – Assume $\mathscr{P} = \mathsf{Q}, \mathsf{P}$. Then the logic is similar to the third subcases of the previous case.
- Assume $M = \mathsf{select}_{\mathsf{Q},\mathsf{P}}\ \ell\, N$. This is similar to the $N_1 = \mathsf{com}^{\tau}_{\mathsf{Q},\mathsf{P}}$ case above.
- Assume $M = f$. Then for any process P, $[\![M]\!]_{\mathsf{P}} = f$. We therefore have some process P such that $\mathscr{P} = \mathsf{P}$ and $\mathscr{N} = ([\![M]\!] \setminus \mathsf{P}) \mid \mathsf{P}[[\![D]\!](f)]$. We then define the required choreography $M' = D(f)$ and network $\mathscr{N}' = [\![M']\!]$ and the result follows.

■