# Relax! The Semilenient Core of Choreographic Programming (Extended Version)

DAN PLYUKHIN, University of Southern Denmark, Denmark
XUEYING QIN, University of Southern Denmark, Denmark
FABRIZIO MONTESI, University of Southern Denmark, Denmark

The past few years have seen a surge of interest in choreographic programming, a programming paradigm for concurrent and distributed systems. The paradigm allows programmers to implement a distributed interaction protocol with a single high-level program, called a *choreography*, and then mechanically *project* it into correct implementations of its participating processes. A choreography can be expressed as a $\lambda$-term parameterized by constructors for creating data "at" a process and for communicating data between processes. Through this lens, recent work has shown how one can add choreographies to mainstream languages like Java, or even embed choreographies as a DSL in languages like Haskell and Rust. These new choreographic languages allow programmers to write in applicative style (like in functional programming) and write higher-order choreographies for better modularity. But the semantics of functional choreographic languages is not well-understood. Whereas typical $\lambda$-calculi can have their operational semantics defined with just a few rules, existing models for *choreographic* $\lambda$-calculi have *dozens* of complex rules and *no clear or agreed-upon evaluation strategy*.

We show that functional choreographic programming is simple. Beginning with the Chor$\lambda$ model from previous work, we strip away inessential features to produce a "core" model called $\lambda^\chi$. We discover that underneath Chor$\lambda$'s apparently ad-hoc semantics lies a close connection to non-strict $\lambda$-calculi; we call the resulting evaluation strategy *semilenient*. Then, inspired by previous non-strict calculi, we develop a notion of *choreographic evaluation contexts* and a special *commute* rule to simplify and explain the unusual semantics of functional choreographic languages. The extra structure leads us to a presentation of $\lambda^\chi$ with just ten rules, and a discovery of three missing rules in previous presentations of Chor$\lambda$. We also show how the extra structure comes with nice properties, which we use to simplify the correspondence proof between choreographies and their projections. Our model serves as both a principled foundation for functional choreographic languages and a good entry point for newcomers.

CCS Concepts: • **Theory of computation → Lambda calculus**; **Distributed computing models**; • **Computing methodologies → Distributed programming languages**.

Additional Key Words and Phrases: Choreographies, Concurrency, $\lambda$-calculus

**ACM Reference Format:**
Dan Plyukhin, Xueying Qin, and Fabrizio Montesi. 2025. Relax! The Semilenient Core of Choreographic Programming (Extended Version). *Proc. ACM Program. Lang.* 9, ICFP, Article 269 (August 2025), 45 pages. https://doi.org/10.1145/3747538

## 1 Introduction

Choreographic programming [Montesi 2013] is a paradigm for writing concurrent code. Programmers can write a single program, called a *choreography*, and *project* it (i.e., compile it) to

Authors' Contact Information: Dan Plyukhin, University of Southern Denmark, Denmark, dplyukhin@imada.sdu.dk; Xueying Qin, University of Southern Denmark, Denmark, xyqin@imada.sdu.dk; Fabrizio Montesi, University of Southern Denmark, Denmark, fmontesi@imada.sdu.dk.


ACM 2475-1421/2025/8-ART269
https://doi.org/10.1145/3747538

generate correct implementations of each process in the application. An oft-repeated slogan is "deadlock-freedom by design", which means that processes projected from a choreography do not deadlock [Carbone and Montesi 2013]. This property is no accident: in fact, proof normalisation in linear logic corresponds to a first-order choreographic programming language [Carbone et al. 2018]. Choreographies also have many other practical benefits: in just the past four years, choreographies have been used to reduce proof burden for Hoare-style verification of concurrent systems [Cruz-Filipe et al. 2023a; van den Bos and Jongmans 2023], as an intermediate representation for distributed cryptography [Acay et al. 2021], and as a tool for implementing distributed applications [Lugovic and Montesi 2024]. For a comprehensive overview, we refer the reader to Montesi's textbook [Montesi 2023].

A recent flurry of activity can be traced to 2020, when the first choreographic programming language for realistic software development appeared: *Choral* [Giallorenzo et al. 2020, 2024] showed how a mainstream language like Java can be made "choreographic" by adding datatypes for located values $T@\mathsf{p}$, and by adding functions of type $T@\mathsf{p} \rightarrow T@\mathsf{q}$ to denote communication from process p to process q. Extending this idea, it was found that (at the cost of some expressive power) choreographic programming can be implemented as a library in sufficiently expressive languages [Shen et al. 2023]. These discoveries led to an explosion of choreographic programming in languages like Haskell [Bates et al. 2025; Shen et al. 2023], Rust [Bates et al. 2025; Laddad et al. 2024], Clojure [klo 2025], and Elixir [cho 2025]. Choral in particular has been used to develop practical applications like IRC [Lugovic and Montesi 2024] and model-serving pipelines [Plyukhin et al. 2024] with similar performance to hand-written processes and deadlock-freedom by design.

Many new choreographic languages are *higher-order*, meaning programmers can write choreographies that take other choreographies as parameters. Higher-order choreographies are useful for writing modular systems: for example, one can implement a replicated key-value store as a choreography and parameterize it by another choreography that implements the replication protocol [Shen et al. 2023]. Higher-order choreographic programming was first presented in Choral, and subsequently came attempts to formalize its semantics [Cruz-Filipe et al. 2022; Cruz-Filipe et al. 2023; Graversen et al. 2024; Hirsch and Garg 2022]. These formalizations all build on the $\lambda$-calculus, but unfortunately they lack its elegant simplicity. For instance, a semantics for Plotkin's call-by-value $\lambda$-calculus can be given with just one axiom [Plotkin 1975], and the call-by-need $\lambda$-calculus only needs three [Ariola et al. 1995]. In contrast, choreographic $\lambda$-calculi currently use more than 20, sometimes almost 40, rules [Cruz-Filipe et al. 2022; Cruz-Filipe et al. 2023; Hirsch and Garg 2022].

Part of the complexity in past models stems from ambition. For instance, Chor$\lambda$ [Cruz-Filipe et al. 2022] includes constructors for algebraic datatypes, so it is not truly a "core" model. But more fundamentally, *researchers have not settled on an evaluation strategy!* Consider the two current foundational models:

- Pirouette [Hirsch and Garg 2022] is call-by-value, but this requires global synchronization—even for non-involved participants—on every choreography call. This is inconsistent with Pirouette's intraprocedural semantics, which has more concurrency than call-by-value. Thus Pirouette lacks a reasonable version of the ($\beta$) axiom, which states that any expression can be factored out into a separate definition (or, dually, inlined) without changing the program's semantics [Barendregt 1984].
- Chor$\lambda$ [Cruz-Filipe et al. 2022] is a more faithful model of higher-order choreographic languages, but its evaluation strategy is mysterious—in fact we shall see it is neither strict nor lazy. The model also crucially depends on ad-hoc "restructuring" rules, and its correctness

proof is quite intimidating. More recent models avoid these restructuring rules by ignoring recursion [Bates et al. 2025] or relying on extra synchronization [Graversen et al. 2024].

Our goal is to show that Chor$\lambda$'s approach is the right way to go, and that its unusual semantics is not a wart—it is a beauty mark. We make our case by developing a new presentation of the core of Chor$\lambda$ and showing that it has a clear evaluation strategy combining features from strict (call-by-value) and non-strict (lenient, or call-by-future [Arvind et al. 1986]) calculi. This connection to non-strict calculi is particularly surprising because our semantics "emerges from" the semantics of the network, which is call-by-value. Because of its close connection to lenient calculi, we call our evaluation strategy *semilenient*.

The remainder of the paper presents $\lambda^\chi$, a model that reveals the elegant functional core at the heart of choreographic programming. Our key contributions are:

- *A streamlined model.* The operational semantics of $\lambda^\chi$ has just ten rules. We accomplish this by cutting away inessential features (like Chor$\lambda$'s data structures and Pirouette's multiple abstractions) and by introducing choreographic evaluation contexts to capture the semilenient evaluation strategy. These evaluation contexts are governed by simple laws, which researchers can use as a recipe to find the right semantics in their own models. We use the extra structure to define a simple *commute* rule, which summarizes what would otherwise be eight seemingly ad-hoc rules. Using the new rule, we discover *three missing rules* in Chor$\lambda$'s published semantics that are necessary for choreographies to match the behavior of their projections.
- *A simplified correspondence proof.* The hallmark result of choreographic programming languages is a *Projection Theorem*, which explains how choreographies and networks correspond. With prior approaches, this result could only be proved by a large and difficult argument by structural induction. Doing so does not give much intuition about how choreographies relate to their projection, and it is easy to make mistakes—leading to incorrect definitions. Here we find that evaluation contexts can shed some light: it turns out that evaluation contexts in the choreography are projected into evaluation contexts at the network. We use this result, along with some other informative lemmas, to give a nice visual proof of the Projection Theorem by commuting diagrams.
- *A stronger correspondence result.* Chor$\lambda$ has conspicuous features at the network level that are not present in ordinary process languages. When compiling to a conventional call-by-value language that lacks these features, it is unclear if important results like deadlock-freedom will actually hold. We fill in the missing piece with a novel *prophecy relation* that lets networks predict the future, and a *Prophecy Theorem* that shows networks with prophecy are no more powerful than regular networks. By working modulo prophecy, our Projection Theorem establishes a weak bisimulation between choreographies and plain old call-by-value networks.

Our model shows higher-order choreographic programming has a simple and elegant foundation in the $\lambda$-calculus. We believe these developments will serve as a good introduction for researchers to the beauty of choreographic programming, and as a practical jumping-off point for future work. We will proceed in three easy pieces. Section 2 introduces the network language—our compilation target—and explains the need for choreographies. Section 3 presents $\lambda^\chi$ and its connections to other $\lambda$-calculi. Section 4 defines the projection from $\lambda^\chi$ to the network level and a new technique for proving its correctness. We wrap up with related work in Section 5 and conclusions in Section 6. In some cases, proofs have been omitted for space; they can be found in Appendix A.

## 2 Networks

Figure 1 presents our network language. We assume an unbounded set of *variables* $x, y, z, \ldots$, of *procedure names* $f, g, h, \ldots$, of *labels* $l_1, l_2, \ldots$, and *process names* $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$. A value $L$ may be a

Terms:

$$\begin{aligned}
P, Q, R &::= L \mid P\ P \mid \text{if } P \text{ then } P \text{ else } P \mid \oplus_{\mathsf{p}}\ l\ P \mid \&_{\mathsf{p}}\{l_1 : P_1, \ldots, l_n : P_n\} \mid f(\overline{\mathsf{p}}) \\
L &::= x \mid \lambda x : S.\ P \mid \mathsf{send}_{\mathsf{p}} \mid \mathsf{recv}_{\mathsf{q}} \mid c \mid \bot \\
S &::= S \to S \mid \alpha \mid \bot \\
\mathbb{P} &::= \{f_i(\overline{\mathsf{p}}) : S_i = P_i\}_{i \in I} \\
\mathcal{N} &::= \mathsf{p}[P] \ \mid\ (\mathcal{N} \mid \mathcal{N})
\end{aligned}$$

Frames:

$$\mathcal{F} ::= \bullet\ P \mid L\ \bullet \mid \text{if } \bullet \text{ then } P \text{ else } P$$

Evaluation contexts:

$$\mathcal{E} ::= \bullet \mid \mathcal{F}[\mathcal{E}]$$

Notions of reduction:

$$\begin{array}{rcll}
\mathsf{send}_{\mathsf{p}}\ c & \xmapsto{\mathsf{send}_{\mathsf{p}}\ c} & \bot & \textit{(send)} \\
\mathsf{recv}_{\mathsf{p}}\ \bot & \xmapsto{\mathsf{recv}_{\mathsf{p}}\ c} & c & \textit{(receive)} \\
\oplus_{\mathsf{p}}\ l\ P & \xmapsto{\oplus_{\mathsf{p}}\ l} & P & \textit{(choice)} \\
\&_{\mathsf{p}}\{l_1 : P_1, \ldots, l_n : P_n\} & \xmapsto{\&_{\mathsf{p}}\ l_i} & P_i & \textit{(offer)} \\
(\lambda x : T.\ P)\ L & \longmapsto & P[x := L] & \textit{(p-app)} \\
\text{if } c \text{ then } P_{\text{true}} \text{ else } P_{\text{false}} & \longmapsto & P_c & \textit{(p-if)} \\
f(\overline{\mathsf{p}}) & \longmapsto & P[\overline{\mathsf{q}} := \overline{\mathsf{p}}] & \textit{(p-def)} \\
 & & \text{where } (f(\overline{\mathsf{q}}) : S = P) \in \mathbb{P} & \\
\bot\ \bot & \longmapsto & \bot & \textit{(bottom)}
\end{array}$$

Evaluation strategy:

$$\frac{P \mapsto P'}{\mathsf{p}[\mathcal{E}[P]] \mid \mathcal{N} \xrightarrow{\tau} \mathsf{p}[\mathcal{E}[P']] \mid \mathcal{N}}\ [\textit{p-tau}]$$

$$\frac{P_1 \xmapsto{\mathsf{send}_{\mathsf{q}}\ c} P_1' \qquad P_2 \xmapsto{\mathsf{recv}_{\mathsf{p}}\ c} P_2'}{\mathsf{p}[\mathcal{E}_1[P_1]] \mid \mathsf{q}[\mathcal{E}_2[P_2]] \mid \mathcal{N} \xrightarrow{\mathsf{com}_{\mathsf{p},\mathsf{q}}} \mathsf{p}[\mathcal{E}_1[P_1']] \mid \mathsf{q}[\mathcal{E}_2[P_2']] \mid \mathcal{N}}\ [\textit{p-com}]$$

$$\frac{P_1 \xmapsto{\oplus_{\mathsf{p}}\ l} P_1' \qquad P_2 \xmapsto{\&_{\mathsf{q}}\ l_i} P_2'}{\mathsf{p}[\mathcal{E}_1[P_1]] \mid \mathsf{q}[\mathcal{E}_2[P_2]] \mid \mathcal{N} \xrightarrow{\mathsf{select}_{\mathsf{p},\mathsf{q}}\ l} \mathsf{p}[\mathcal{E}_1[P_1']] \mid \mathsf{q}[\mathcal{E}_2[P_2']] \mid \mathcal{N}}\ [\textit{p-select}]$$

Fig. 1. Syntax and semantics for networks

variable $x$, an abstraction $\lambda x : S.\ P$, a communication primitive $\mathsf{send}_{\mathsf{p}}$ or $\mathsf{recv}_{\mathsf{p}}$, or a constant $c$. We assume the language includes constants $\mathsf{true}$ and $\mathsf{false}$ with the usual meaning, and a distinguished constant $\bot$ that indicates "nothing left to do". Note that process names and procedure names are not values; our choreography model will require a strict separation between processes and ordinary

$$\frac{\mathsf{p} \in \Theta \quad \Theta; \Gamma \vdash P : S}{\Theta; \Gamma \vdash \oplus_{\mathsf{p}}\ l\ P : S}\ [\text{NTCho}] \qquad \frac{\mathsf{p} \in \Theta \quad \Theta; \Gamma \vdash P_i : S \text{ for } 1 \le i \le n}{\Theta; \Gamma \vdash \&_{\mathsf{p}}\{l_1 : P_1, \dots, l_n : P_n\} : S}\ [\text{NTOff}]$$

$$\frac{\mathsf{p} \in \Theta}{\Theta; \Gamma \vdash \mathsf{send}_{\mathsf{p}} : S \to \bot}\ [\text{NTSend}] \qquad \frac{\mathsf{p} \in \Theta}{\Theta; \Gamma \vdash \mathsf{recv}_{\mathsf{p}} : \bot \to S}\ [\text{NTRecv}]$$

$$\frac{\Gamma, x : S \vdash P : S'}{\Theta; \Gamma \vdash \lambda x : S.\ P : S \to S'}\ [\text{NTAbs}] \qquad \frac{x : S \in \Gamma}{\Theta; \Gamma \vdash x : S}\ [\text{NTVar}]$$

$$\frac{\Theta; \Gamma \vdash P_1 : S \to S' \quad \Theta; \Gamma \vdash P_2 : S}{\Theta; \Gamma \vdash P_1\ P_2 : S'}\ [\text{NTApp}]$$

$$\frac{\Theta; \Gamma \vdash P : \mathsf{Bool} \quad \Theta; \Gamma \vdash P_1 : S \quad \Theta; \Gamma \vdash P_2 : S}{\Theta; \Gamma \vdash \mathsf{if}\ P\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 : S}\ [\text{NTIf}]$$

$$\frac{\mathsf{type}(c) = \alpha}{\Theta; \Gamma \vdash c : \alpha}\ [\text{NTConst}] \qquad \frac{(f(\overline{\mathsf{q}}) : S = P) \in \mathbb{P} \quad |\overline{\mathsf{p}}| = |\overline{\mathsf{q}}| \quad \mathsf{distinct}(\overline{\mathsf{p}}) \quad \overline{\mathsf{p}} \subseteq \Theta}{\Theta; \Gamma \vdash f(\overline{\mathsf{p}}) : S}\ [\text{NTDef}]$$

Fig. 2. Typing rules for processes.

data, and combining the two requires process polymorphism [Graversen et al. 2024] which is beyond the scope of our work.

A *process* $P$ is composed of familiar constructs like values $L$, applications $P_1\ P_2$, and if-expressions $\mathsf{if}\ P'\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2$. We also include primitives for distributed choice, drawn from concurrency theory: $\&_{\mathsf{p}}\{l_1 : P_1, \dots, l_n : P_n\}$ and $\oplus_{\mathsf{p}}\ l\ P$, explained below. A *network* $\mathcal{N}$ is a fixed set of processes $\mathsf{p}_1[P_1] \mid \dots \mid \mathsf{p}_n[P_n]$, where the process names $\mathsf{p}_1, \dots, \mathsf{p}_n$ are distinct and the processes $P_1, \dots, P_n$ are *closed*, i.e., they have no free variables.

To write non-terminating programs, the semantics is parameterized by a set of recursive procedure definitions $\mathbb{P} = \{f_i(\overline{\mathsf{p}}) : S_i = P_i\}_{i \in \mathcal{I}}$. Each procedure $f_i$ in $\mathbb{P}$ is parameterized by a list of process names $\overline{\mathsf{p}}$. When $(f(\overline{\mathsf{p}}) : S = P) \in \mathbb{P}$ and some process makes a procedure call $f(\overline{\mathsf{q}})$, the process names $\overline{\mathsf{q}}$ are substituted for $\overline{\mathsf{p}}$ in the procedure body $P$. Thus procedures can be applied to process names and abstractions can be applied to values, but not vice versa.

For the static semantics of processes, we assume a set of base types $\alpha \in \{\mathsf{Bool}, \bot, \dots\}$ and a function 'type(−)' mapping constants to their types. Typing judgments for processes have the form $\Theta; \Gamma \vdash_{\mathbb{P}} P : S$, where $\Theta$ is a set of process names in scope, $\Gamma$ is a typing context mapping from variables to types, $\mathbb{P}$ is a set of procedure definitions, $P$ is a process, and $S$ is its type; for readability, we often leave $\mathbb{P}$ implicit and instead write $\Theta; \Gamma \vdash P : S$. A set of procedure definitions $\mathbb{P}$ is *well-typed* if, for each definition $f(\overline{\mathsf{p}}) : S = P$ in $\mathbb{P}$, we have $\overline{\mathsf{p}}; \Gamma \vdash P : S$. The full set of typing rules for processes appears in Figure 2.

A process can send q a value $L$ by applying $\mathsf{send}_{\mathsf{q}}$ to $L$; likewise, it can wait for a message from p with the application $\mathsf{recv}_{\mathsf{p}}\ \bot$. A process can signal a control flow decision to q with the expression $\oplus_{\mathsf{q}}\ l\ P$, which means "send label $l$ to q and continue as $P$"; conversely, it can wait for a decision from p with the expression $\&_{\mathsf{p}}\{l_1 : Q_1, \dots, l_n : Q_n\}$, which means "upon receiving label $l_i$ from p, continue as $Q_i$". Strictly speaking, distributed choice can be implemented with ordinary $\mathsf{send}_{\mathsf{q}}$ and $\mathsf{recv}_{\mathsf{p}}$—but it is common practice to make the two primitives distinct in process calculi based on linear logic and session types. We include choice primitives to retain coherence with that work.

We give the network language a standard call-by-value semantics using evaluation contexts, with a small creative choice in how we present the latter. Usually, evaluation contexts are defined

$$\mathsf{auth}\begin{bmatrix}\text{let } x = \text{recv}_{\mathsf{c}} \text{ in} \\ \text{if } \mathit{valid}\, x \\ \text{then } \oplus_{\mathsf{ws}}\ \textsc{ok} \\ \text{else } \oplus_{\mathsf{ws}}\ \textsc{ko}\end{bmatrix} \;\Big|\; \mathsf{c}\begin{bmatrix}\text{send}_{\mathsf{auth}}\ \mathit{creds}; \\ \text{let } \mathit{result} = \text{recv}_{\mathsf{ws}}\ \bot \text{ in} \\ \mathit{print}\ \mathit{result}\end{bmatrix} \;\Big|\; \mathsf{ws}\begin{bmatrix}\&_{\mathsf{auth}}\{ \\ \quad \textsc{ok} : \text{send}_{\mathsf{c}}\ \mathit{newToken}() \\ \quad \textsc{ko} : \text{send}_{\mathsf{c}}\ \mathit{noToken} \\ \}\end{bmatrix}$$

Fig. 3. A simple distributed authentication protocol based on OpenID [Montesi 2023]. The example uses let-sugar, i.e. $\text{let } x = P_1 \text{ in } P_2 \equiv (\lambda x.\, P_2)\ P_1$, and (;)-sugar, i.e. $P_1;\, P_2 \equiv (\lambda x.\, P_2)\ P_1$ where $x$ is fresh. We will use this convention in the network language throughout the paper.

```
let x = com_{c,auth} creds@c in
let result =
    if valid@auth x then
      select_{auth,ws} OK
      newToken(ws)
    else
      select_{auth,ws} KO
      noToken@ws
 in print@c (com_{ws,c} result)
```

Fig. 4. A choreographic implementation of the distributed authentication protocol. Projecting this choreography generates a network equivalent to the one in Figure 3.

by direct induction like so:

$$\mathcal{E}' \ ::= \mathcal{E}'\ P \mid L\ \mathcal{E}' \mid \text{if } \mathcal{E}' \text{ then } P \text{ else } P \mid \bullet$$

Instead we have defined evaluation contexts as a stack of *frames* $\mathcal{F}$, so any evaluation context $\mathcal{E}$ can be written as $\mathcal{F}_1[\mathcal{F}_2[\dots]]$. This presentation—which we borrowed from System $F_J$ [Maurer et al. 2017], but dates back at least to Huet's zippers [Huet 1997]—will become useful in Section 3 when we introduce choreographic evaluation contexts as stacks of both *choreographic frames* and so-called *answering contexts*. By routine induction, the definitions of $\mathcal{E}$ and $\mathcal{E}'$ are equivalent.

### 2.1 Motivating Choreographies

Figure 3 implements a simple authentication protocol in the network language, with types omitted for readability. Before reading our explanation below, we encourage the reader to stop and think about what will happen when Figure 3 is executed. Can the network reach a deadlock, i.e., a state where some processes are not values and yet the network has no next step?

Many programmers will intuitively reason about networks by sketching a sequence diagram [Object Management Group 2017] or using Alice-and-Bob notation [Needham and Schroeder 1978]. Others might construct a multiparty session type [Honda et al. 2016] and verify the network conforms to the type. But diagrams and session types both involve *supplementing* an existing network with extra information to specify its emergent behavior and *verifying* that the network actually respects the specification.

Choreographic programming is a more direct approach that aims to turn Alice-and-Bob notation into a *bona fide* programming language. This allows us to write concurrent programs at a higher level of abstraction and generate processes that implement the desired protocol. Figure 4 shows an implementation of the authentication protocol as a $\lambda^\chi$ choreography, with types omitted; we will not formalize its syntax and semantics until Section 3, so observe just the broad features for now. Notice the choreography looks like an ordinary functional program, but with three new attributes: (1) a syntax for tagging constants with locations, e.g. *creds*@c represents the *creds* constant at process c; (2) a primitive $\mathsf{com}_{p,q}$ for communicating a value from p to q; and (3) a primitive $\mathsf{select}_{p,q}\ l$ so p can inform q about a control flow decision. Readers who drew a sequence diagram for Figure 3 can check that Figure 4 likely resembles their informal specification.

### 2.2 Authentication Explained

Reading the choreography in Figure 4, the protocol begins with an expression $\mathsf{com}_{c,auth}$ *creds*@c that communicates the value *creds* from c to auth. This single expression corresponds to the terms $\mathsf{send}_{auth}$ *creds* and $\mathsf{recv}_c \perp$ in Figure 3. Next, auth checks if the credentials are valid and sends a signal to ws—either $\mathsf{select}_{auth,ws}$ OK or $\mathsf{select}_{auth,ws}$ KO. These expressions correspond to the choice auth makes—either $\oplus ws$ OK or $\oplus ws$ KO—and the offer $\&_{auth}\{\dots\}$ made by ws in Figure 3. Finally, the result is sent to c and printed. Figure 4 can be projected to produce a network very much like Figure 3, with minor differences that will be evident in Section 4. By construction, the projected network is both type-safe and deadlock-free [Cruz-Filipe et al. 2023].

Readers familiar with choreographic programming will notice Figure 4 is written in applicative style, like a conventional functional language. The if-expression on lines 3–8 returns a value that we can bind to the variable *result*, and line 9 applies *print*@c directly to the expression $\mathsf{com}_{ws,c}$ *result* without giving a name to the intermediate result. Applicative features were not available to programmers until the Choral [Giallorenzo et al. 2024] language was released, and most formal models still only allow a first-order imperative programming style [Montesi 2023; Plyukhin et al. 2024]. In the next section, we explore why functional choreographic programming is so elusive.

## 3 Choreographies

Let us now introduce our choreography model $\lambda^\chi$ properly. A *(choreographic) value* $V$ may be a variable $x$, an abstraction $\lambda x : T.\, M$, a communication primitive $\mathsf{com}_{p,q}$, or a constant $c@p$ located at process p. A *choreography* $M$ may be an application $M_1\ M_2$, an if-expression $\mathsf{if}\ M'\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$, a selection $\mathsf{select}_{p,q}\ l\ M'$, a choreographic procedure call $f(\overline{p})$, or a value $V$. Choreographies also have let-expressions $\mathsf{let}\ x = M\ \mathsf{in}\ M'$, which (unlike in the network language) are not just sugar—we shall see why this is useful below. Aside from let-expressions, the syntax of $\lambda^\chi$ is essentially the same as our network model, except the communication primitives $\mathsf{send}_q$, $\mathsf{recv}_p$ have been unified by $\mathsf{com}_{p,q}$ and the choice primitives $\oplus_q$, $\&_p$ have been unified by $\mathsf{select}_{p,q}$. We also assume if-expressions and choreographic procedure calls are either let-bound or in tail position; for example, $f(\overline{p})\ M$ must be expanded as $\mathsf{let}\ x = f(\overline{p})\ \mathsf{in}\ M$. The syntax is summarized in Figure 5.

Choreographic types $T$ are also similar to network-level types. Base types $\alpha@p$ are now tagged with their location, and function types $T \to_{\overline{p}} T'$ are tagged with a list of processes $\overline{p}$ called *proxy processes*; our type system will ensure that if a process p is involved in computing the function's result, then p will occur in $T$, $T'$, or $\overline{p}$. If the function type has no proxy processes, we will write it as $T \to T'$. The set of processes that *occur* in a term is formalized by the function 'pn(−)', defined at the bottom of Figure 5.

Typing judgments for choreographies, written $\Theta; \Gamma \vdash_{\mathbb{D}} M : T$, are analogous to typing judgments for processes: $\Theta$ is a set of process names in scope, $\Gamma$ is a mapping from variables to choreographic types, $\mathbb{D}$ is a set of choreographic procedure definitions, $M$ is a choreography, and $T$ is a choreographic type; for readability we usually leave $\mathbb{D}$ implicit. The $\lambda^\chi$ typing rules are defined in Figure 6 and are very similar to the network-level typing rules. Notice how the network-level rules [NTSend], [NTRecv] have been unified by the choreographic rule [TCom]—likewise for

Terms:

$$M ::= V \mid M\ M \mid \mathsf{let}\ x : T = M\ \mathsf{in}\ M \mid \mathsf{if}\ M\ \mathsf{then}\ M\ \mathsf{else}\ M \mid \mathsf{select}_{\mathsf{p},\mathsf{p}}\ l\ M \mid f(\overline{\mathsf{p}})$$
$$V ::= x \mid \lambda x : T.M \mid \mathsf{com}_{\mathsf{p},\mathsf{p}} \mid c@\mathsf{p}$$
$$T ::= T \to_{\overline{\mathsf{p}}} T \mid \alpha@\mathsf{p}$$
$$\mathbb{D} ::= \{f_i(\overline{\mathsf{p}}) : T_i = M_i\}_{i \in \mathcal{I}}$$

Frames:

$$\mathcal{F} ::= \bullet\ M \mid V\ \bullet \mid \mathsf{if}\ \bullet\ \mathsf{then}\ M\ \mathsf{else}\ M \mid \mathsf{let}\ x : T = \bullet\ \mathsf{in}\ M$$

Answering contexts:

$$\mathcal{A} ::= \mathsf{let}\ x : T = M\ \mathsf{in}\ \bullet \mid \mathsf{select}_{\mathsf{q},\mathsf{r}}\ l\ \bullet$$

Evaluation contexts:

$$\mathcal{E}_{\overline{\mathsf{p}}} ::= \bullet \mid \mathcal{F}[\mathcal{E}_{\overline{\mathsf{p}}}] \mid \mathcal{A}[\mathcal{E}_{\overline{\mathsf{p}}}] \qquad \text{where } \overline{\mathsf{p}} \# \mathrm{pn}(\mathcal{A})$$

Notions of reduction:

$$\mathsf{com}_{\mathsf{p},\mathsf{q}}\ c@\mathsf{p} \xmapsto{\mathsf{com}_{\mathsf{p},\mathsf{q}}} c@\mathsf{q} \qquad (com)$$
$$\mathsf{select}_{\mathsf{p},\mathsf{q}}\ l\ M \xmapsto{\mathsf{select}_{\mathsf{p},\mathsf{q}}\ l} M \qquad (select)$$
$$(\lambda x : T.M)\ M' \xmapsto{\overline{\mathsf{p}}} \mathsf{let}\ x : T = M'\ \mathsf{in}\ M \qquad (app)$$
$$\mathsf{let}\ x : T = V\ \mathsf{in}\ M \xmapsto{\overline{\mathsf{p}}} M[x := V] \quad \text{where } \overline{\mathsf{p}} = \mathrm{pn}(V) \qquad (let)$$
$$\mathsf{if}\ c@\mathsf{p}\ \mathsf{then}\ M_{\mathsf{true}}\ \mathsf{else}\ M_{\mathsf{false}} \xmapsto{\mathsf{p}} M_c \qquad (if)$$
$$f(\overline{\mathsf{p}}) \xmapsto{\overline{\mathsf{p}}} M[\overline{\mathsf{q}} := \overline{\mathsf{p}}] \quad \text{where } (f(\overline{\mathsf{q}}) : T = M) \in \mathbb{D} \qquad (def)$$
$$\mathcal{F}[\mathcal{A}[M]] \xmapsto{\overline{\mathsf{p}}} \mathcal{A}[\mathcal{F}[M]] \qquad (commute)$$

Evaluation strategy:

$$\frac{M \overset{\overline{\mathsf{p}}}{\mapsto} M' \quad \mathsf{p} \in \overline{\mathsf{p}}}{\mathcal{E}_{\mathsf{p}}[M] \xrightarrow{\tau} \mathcal{E}_{\mathsf{p}}[M']}\ [\textit{c-tau}] \qquad \frac{M \xmapsto{\mathsf{com}_{\mathsf{p},\mathsf{q}}} M'}{\mathcal{E}_{\mathsf{p},\mathsf{q}}[M] \xrightarrow{\mathsf{com}_{\mathsf{p},\mathsf{q}}} \mathcal{E}_{\mathsf{p},\mathsf{q}}[M']}\ [\textit{c-com}] \qquad \frac{M \xmapsto{\mathsf{select}_{\mathsf{p},\mathsf{q}}\ l} M'}{\mathcal{E}_{\mathsf{p},\mathsf{q}}[M] \xrightarrow{\mathsf{select}_{\mathsf{p},\mathsf{q}}\ l} \mathcal{E}_{\mathsf{p},\mathsf{q}}[M']}\ [\textit{c-select}]$$

Mentioned processes:

$$\mathrm{pn}(M_1\ M_2) = \mathrm{pn}(M_1) \cup \mathrm{pn}(M_2)$$
$$\mathrm{pn}(\mathsf{if}\ M\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2) = \mathrm{pn}(M) \cup \mathrm{pn}(M_1) \cup \mathrm{pn}(M_2)$$
$$\mathrm{pn}(\lambda x : T.M) = \mathrm{pn}(T) \cup \mathrm{pn}(M)$$
$$\mathrm{pn}(\mathsf{com}_{\mathsf{p},\mathsf{q}}) = \{\mathsf{p}, \mathsf{q}\}$$
$$\mathrm{pn}(\bullet) = \emptyset$$
$$\mathrm{pn}(\mathsf{select}_{\mathsf{p},\mathsf{q}}\ l\ M) = \{\mathsf{p}, \mathsf{q}\} \cup \mathrm{pn}(M)$$
$$\mathrm{pn}(\mathsf{let}\ x : T = M\ \mathsf{in}\ M') = \mathrm{pn}(T) \cup \mathrm{pn}(M) \cup \mathrm{pn}(M')$$
$$\mathrm{pn}(x) = \mathrm{pn}(\mathrm{type}(x))$$
$$\mathrm{pn}(c@\mathsf{p}) = \mathrm{pn}(T@\mathsf{p}) = \mathsf{p}$$
$$\mathrm{pn}(T_1 \to_{\overline{\mathsf{p}}} T_2) = \mathrm{pn}(T_1) \cup \mathrm{pn}(T_2) \cup \overline{\mathsf{p}}$$

Fig. 5. Syntax and semantics for $\lambda^{\chi}$

$$\frac{\Theta'; \Gamma, x : T \vdash M : T' \quad \mathsf{pn}(T \rightarrow_{\overline{\mathsf{p}}} T') = \Theta' \subseteq \Theta}{\Theta; \Gamma \vdash \lambda x : T.M : T \rightarrow_{\overline{\mathsf{p}}} T'} \text{[TABS]}$$

$$\frac{x : T \in \Gamma \quad \mathsf{pn}(T) \subseteq \Theta}{\Theta; \Gamma \vdash x : T} \text{[TVAR]} \qquad \frac{\Theta; \Gamma \vdash M_1 : T \rightarrow_{\overline{\mathsf{p}}} T' \quad \Theta; \Gamma \vdash M_2 : T}{\Theta; \Gamma \vdash M_1\ M_2 : T'} \text{[TAPP]}$$

$$\frac{\Gamma \vdash M : \mathsf{Bool@p} \quad \Theta; \Gamma \vdash M_1 : T \quad \Theta; \Gamma \vdash M_2 : T \quad \mathsf{p} \in \Theta}{\Theta; \Gamma \vdash \mathsf{if}\ M\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 : T} \text{[TIF]}$$

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

$$\frac{\mathsf{p} \in \Theta \quad \mathsf{type}(c) = \alpha}{\Theta; \Gamma \vdash c@\mathsf{p} : \alpha@\mathsf{p}} \text{[TCONST]} \qquad \frac{\mathsf{p}, \mathsf{q} \in \Theta}{\Theta; \Gamma \vdash \mathsf{com}_{\mathsf{p},\mathsf{q}} : \alpha@\mathsf{p} \rightarrow_{\emptyset} \alpha@\mathsf{q}} \text{[TCOM]}$$

$$\frac{(f(\overline{\mathsf{q}}) : T = M) \in \mathbb{D} \quad |\overline{\mathsf{p}}| = |\overline{\mathsf{q}}| \quad \mathsf{distinct}(\overline{\mathsf{p}})}{\Theta; \Gamma \vdash f(\overline{\mathsf{p}}) : T[\overline{\mathsf{q}} := \overline{\mathsf{p}}]} \text{[TDEF]}$$

$$\frac{\Theta; \Gamma \vdash M_1 : T_1 \quad \Theta; \Gamma, x : T_1 \vdash M_2 : T_2}{\Theta; \Gamma \vdash \mathsf{let}\ x : T_1 = M_1\ \mathsf{in}\ M_2 : T_2} \text{[TLET]}$$

Fig. 6. Full set of typing rules for $\lambda^\chi$.

[NTCho], [NTOff] and [TSel]. A set of choreographic procedure definitions $\mathbb{D}$ is *well-typed* if, for each definition $f(\overline{\mathsf{p}}) : T = M$ in $\mathbb{D}$, we have $\overline{\mathsf{p}}; \Gamma \vdash_{\mathbb{D}} M : T$.

The key novelty of $\lambda^\chi$ is how we present its dynamic semantics. But before diving into the details, let us discuss evaluation strategies.

### 3.1 Evaluation Strategies

The dynamic semantics of a $\lambda$-calculus can be presented in two parts. The first part is a *notion of reduction* $M \overset{\mu}{\longmapsto} M'$, which defines atomic computational steps such as $\beta$-reduction; we say $M$ is a redex, $M'$ is its contractum, and $\mu$ is an optional transition label. The second part is an *evaluation strategy* $M \xrightarrow{\mu} M'$, which defines the order redexes should be reduced [Barendregt 1984]. Two well-known evaluation strategies are call-by-value ("strict", like our network language) and call-by-need ("lazy", elegantly presented in the call-by-need calculus [Ariola et al. 1995]). These two evaluation strategies are deterministic: for any process $P$ in our network language there is at most one $P'$ such that $P \overset{\mu}{\longmapsto} P'$. (However, a network $\mathcal{N}$ may have more than one $\mathcal{N}'$ such that $\mathcal{N} \xrightarrow{\mu} \mathcal{N}'$, because processes execute concurrently.)

Choreographic programming languages have nondeterministic evaluation strategies. To motivate this, consider the first-order choreography in Figure 7. It defines a choreographic procedure $loop(\mathsf{p}, \mathsf{q})$ which passes the integer 1 between p and q *ad infinitum*. Processes $\mathsf{r}_1$ and $\mathsf{r}_2$ enter this procedure on line 4, and $\mathsf{r}_3$ sends the integer 2 to $\mathsf{r}_4$ on line 5. A typical projection for this choreography is shown on the right; notice that the *loop* procedure was projected into two implementations, one for each of its role parameters.

We expect choreographic programs to have the same behavior as their projection. Inspecting the projected network, we see that the execution of $\mathsf{r}_1, \mathsf{r}_2$ and $\mathsf{r}_3, \mathsf{r}_4$ can be arbitrarily interleaved. This implies that a usual sequential semantics is inadequate for the choreography in Figure 7: it must be possible for the procedure call on line 4 to execute *out of order* with the communication on line 5, because the instructions involve different processes. This semantics, where the semicolon

$$\begin{array}{lll}
\mathit{loop}(\mathsf{p},\mathsf{q}) =_{def} & \mathit{loop}_{\mathsf{p}}(\mathsf{q}) =_{def} & \mathit{loop}_{\mathsf{q}}(\mathsf{p}) =_{def} \\
\quad \mathsf{com}_{\mathsf{p},\mathsf{q}}\ 1@\mathsf{p}; & \quad \mathsf{send}_{\mathsf{q}}\ 1; & \quad \mathsf{recv}_{\mathsf{p}}\ \bot; \\
\quad \mathit{loop}(\mathsf{q},\mathsf{p}) & \quad \mathit{loop}_{\mathsf{q}}(\mathsf{q}) & \quad \mathit{loop}_{\mathsf{p}}(\mathsf{p}) \\
\mathit{loop}(\mathsf{r}_1,\mathsf{r}_2); & & \\
\mathsf{com}_{\mathsf{r}_3,\mathsf{r}_4}\ 2 & \multicolumn{2}{l}{\mathsf{r}_1\big[\mathit{loop}_{\mathsf{p}}(\mathsf{r}_2)\big] \mid \mathsf{r}_2\big[\mathit{loop}_{\mathsf{q}}(\mathsf{r}_1)\big] \mid \mathsf{r}_3\big[\mathsf{send}_{\mathsf{r}_4}\ 1\big] \mid \mathsf{r}_4\big[\mathsf{recv}_{\mathsf{r}_3}\ \bot\big]}
\end{array}$$

Fig. 7. A choreography with a loop (left) and a typical projection (right).

operator is not quite sequential and not quite parallel, is the *de facto* standard in modern choreography languages and textbooks [Montesi 2023] and is crucial for the linear logic interpretation of choreographies [Carbone et al. 2018].

### 3.2 Lenient Semantics

When adapting a first-order language to the higher-order setting, we expect certain equivalences to be obeyed. Functional languages often use let-sugar and (;)-sugar, as seen in Figure 3. These desugarings give us a hint about what the meaning of an application $(\lambda x : T.\ M)\ M'$ should be. Specifically, we expect these three choreographies to have identical semantics:

$$(\lambda x.\ \mathsf{com}_{\mathsf{r}_3,\mathsf{r}_4}\ 2)\ \mathit{loop}(\mathsf{r}_1,\mathsf{r}_2) \tag{1}$$

$$\mathsf{let}\ x = \mathit{loop}(\mathsf{r}_1,\mathsf{r}_2)\ \mathsf{in}\ \mathsf{com}_{\mathsf{r}_3,\mathsf{r}_4}\ 2. \tag{2}$$

$$\mathit{loop}(\mathsf{r}_1,\mathsf{r}_2);\ \mathsf{com}_{\mathsf{r}_3,\mathsf{r}_4}\ 2. \tag{3}$$

Seen in this light, one might suggest a *lenient* semantics. Lenient evaluation is a nondeterministic evaluation strategy that is neither strict nor lazy [Tremblay 2000] but has much in common with call-by-need [Ariola et al. 1995; Arvind et al. 1996]. Its famous exponents include Id [Arvind et al. 1986], parallel Haskell [Arvind et al. 1996], and the Verse calculus [Augustsson et al. 2023].

Lenient semantics allows expressions to be evaluated concurrently *up to data dependency*. Hence it would indeed allow $\mathsf{com}_{\mathsf{r}_3,\mathsf{r}_4}\ 2$ and $\mathit{loop}(\mathsf{r}_1,\mathsf{r}_2)$ to execute concurrently because neither expression depends on the other. But unfortunately, a lenient semantics would be *too concurrent!* Specifically, it allows us to unfold the $\mathit{loop}(\mathsf{r}_1,\mathsf{r}_2)$ call once:

$$\mathsf{com}_{\mathsf{r}_1,\mathsf{r}_2}\ 1@\mathsf{r}_1;\ \mathit{loop}(\mathsf{r}_2,\mathsf{r}_1);\ \mathsf{com}_{\mathsf{r}_3,\mathsf{r}_4}\ 2 \tag{4}$$

And then to unfold the $\mathit{loop}(\mathsf{r}_2,\mathsf{r}_1)$ call:

$$\mathsf{com}_{\mathsf{r}_1,\mathsf{r}_2}\ 1@\mathsf{r}_1;\ \mathsf{com}_{\mathsf{r}_2,\mathsf{r}_1}\ 1@\mathsf{r}_2;\ \mathit{loop}(\mathsf{r}_1,\mathsf{r}_2);\ \mathsf{com}_{\mathsf{r}_3,\mathsf{r}_4}\ 2 \tag{5}$$

And then to reduce $\mathsf{com}_{\mathsf{r}_2,\mathsf{r}_1}\ 1@\mathsf{r}_2$. In other words, a lenient semantics would tell us $\mathsf{r}_2$ can send a message to $\mathsf{r}_1$ in the initial state of the network. But by inspecting the projected code in Figure 7, we can see that $\mathsf{r}_2$ will never send a message until it first receives a message from $\mathsf{r}_1$.

Thus lenient evaluation captures the *concurrency up to data dependency* of choreographies, but fails to capture *process dependencies* that enforce sequentiality. Armed with this insight, we will use prior art in non-strict $\lambda$-calculi [Ariola et al. 1995; Arvind et al. 1996; Maurer et al. 2017] to develop a language that is lenient up to process ordering—or *semilenient* for short.

### 3.3 Semilenient Semantics

We now explain $\lambda^{\chi}$'s semilenient semantics formally. We do this by defining notions of reduction—rules *(com)*, *(select)*, *(app)*, *(let)*, *(if)*, *(def)*, and *(commute)* in Figure 5—and an appropriate notion of evaluation context. Unlike ordinary evaluation contexts, a choreographic evaluation context is indexed by a set of process names $\mathcal{E}_{\overline{p}}$. Decomposing a choreography $M$ with an evaluation context $M = \mathcal{E}_{\overline{p}}[M']$ means that, if $p \in \overline{p}$, then p will evaluate $M'$ in its next step. Since choreographies are concurrent, a term $M$ can have distinct decompositions $M = \mathcal{E}_p[M'] = \mathcal{E}_q[M'']$ for different processes p, q.

Our evaluation contexts may appear complex, but in fact there are principles we can use to derive them. Whereas evaluation contexts in the network language are stacks of frames (Section 2), choreographic evaluation contexts are stacks of (choreographic) frames together with so-called *answering contexts*, explained below.

*3.3.1 Choreographic Frames and Answering Contexts.* A *(choreographic) frame* $\mathcal{F}$ is the choreographic analogue of a network-level frame. Notice that each kind of network-level frame in Figure 1 corresponds directly to a choreographic frame in Figure 5; the only extra case we add is $\mathcal{F} ::= \mathsf{let}\ x : T = \bullet\ \mathsf{in}\ M$ because let-bindings are no longer just syntactic sugar in $\lambda^\chi$.

An *answering context* $\mathcal{A}$ is a context where, if $p \notin \mathrm{pn}(\mathcal{A})$, then p has no work left to do. By filling an answering context with a value $\mathcal{A}[V]$, we make a choreographic term where p is ready to pass $V$ to its enclosing context. Answering contexts are analogous to the notion of *answers* in the call-by-need lambda calculus [Ariola et al. 1995] and the notion of *tail contexts* in System $F_J$ [Maurer et al. 2017].

How did we arrive at these definitions for $\mathcal{F}$ and $\mathcal{A}$? In fact, they emerge as properties of the projection function, which we define in Section 4. Choreographic frames are defined so that $\mathcal{F}$ projects to a network-level frame $\mathcal{F}$. Meanwhile, answering contexts are defined so that the projection of $\mathcal{A}[M]$ for p is equal to the projection of $M$ whenever $p \notin \mathrm{pn}(\mathcal{A})$. In other words, answering contexts are precisely the contexts that disappear during projection.

*3.3.2 Notions of Reduction and the Commute Rule.* Most of the notions of reduction in $\lambda^\chi$ should be unsurprising. The *(com)* rule moves a constant from p to q. The *(select)* rule simply steps into its continuation; it is a synchronization between p and q. The *(if)* and *(def)* rules are the same as in the network level, but with extra annotations to involve processes. However, the network's *(p-app)* rule is split into two steps: *(app)* reduces an application $(\lambda x : T.M)\ M'$ into an explicit let-binding $\mathsf{let}\ x : T = M'\ \mathsf{in}\ M$, allowing our evaluation contexts to begin evaluating $M$ without waiting for $M'$ to be a value. Once $M'$ is a value, the *(let)* rule substitutes that value into the body of the let-binding. (Another example of this can be seen in the call-by-need calculus, where the context let $x = M'$ in $M$ denotes a "thunk" $x$ bound to $M'$ in the expression $M$ [Ariola et al. 1995].) The transition labels for all these rules are generally determined by the redex, except in the case of *(app)* and *(commute)* where the label $\overline{p}$ can be any list of process names.

The *(commute)* rule is unusual: it pushes a frame $\mathcal{F}$ into an answering context $\mathcal{A}$. This rule, which we borrowed from System $F_J$ [Maurer et al. 2017], precisely captures the mysterious and ad-hoc "restructuring" rules from Chor$\lambda$. Consider the term:

$$\begin{pmatrix}\mathsf{let}\ x = f(\mathsf{r})\ \mathsf{in} \\ \mathsf{com}_{p,q}\end{pmatrix}\ 0@p,$$

where $f(\mathsf{r})$ diverges. Its projection at p is simply $\mathsf{send}_q\ 0$ because the call $f(\mathsf{r})$ is irrelevant to p. (Our type system guarantees, in general, that $f(\overline{r})$ only involves $\overline{r}$.) Likewise, the projection at q is $\mathsf{recv}_p\ \bot$. In fact we can construct many examples like this:

$$\mathsf{com}_{p,q}\begin{pmatrix}\mathsf{let}\ x = f(\mathsf{r})\ \mathsf{in} \\ 0@p\end{pmatrix} \qquad \begin{array}{l}\mathsf{let}\ x = \begin{pmatrix}\mathsf{let}\ y = f(\mathsf{r})\ \mathsf{in} \\ 0@p\end{pmatrix}\ \mathsf{in} \\ \mathsf{com}_{p,q}\ x\end{array} \qquad \mathsf{com}_{p,q}\begin{pmatrix}\mathsf{if}\ \begin{pmatrix}\mathsf{let}\ x = f(\mathsf{r})\ \mathsf{in} \\ \mathsf{true}@p\end{pmatrix} \\ \mathsf{then}\ 0@p \\ \mathsf{else}\ 1@p\end{pmatrix}$$

In each case above, the choreography semantics should allow p and q to communicate without waiting for r—but the nonterminating computation $f(\mathsf{r})$ prevents us from ever creating the redex $\mathsf{com}_{\mathsf{p,q}}\ 0@\mathsf{p}$. Higher-order choreographic languages therefore need rules for rewriting terms to expose reducible expressions; let us call these extra rules *restructuring rules*.

How does one know what restructuring rules a choreography language will need? In past approaches, there was only one way to find out: try to prove that choreographies and networks correspond, and add more restructuring rules when the proof gets stuck. This solution is unappealing because even a simple language requires many restructuring rules, and it is easy to miss cases when working by hand. Our presentation reveals a hidden structure behind the restructuring rules: they are all instances of the *(commute)* rule.

Figure 8 shows the restructuring rules generated by *(commute)* if we inline the definitions of $\mathcal{F}$ and $\mathcal{A}$. Some of these rules further reinforce the connection between choreographies and non-strict calculi: *(let-let)* and *(app-let)* correspond to *(let-A)* and *(let-C)* from the call-by-need $\lambda$-calculus [Ariola et al. 1995]. These rules, along with *(let-app)*, *(app-sel)*, and *(sel-app)*, are all found (with a slightly different presentation) in Chor$\lambda$ [Cruz-Filipe et al. 2023].

Surprisingly, the remaining three rules *(if-let)*, *(if-sel)*, and *(let-sel)* from Figure 8 are all missing from the original publication of Chor$\lambda$. Upon closer inspection, we discovered that all three rules are indeed necessary for the correspondence result, and they were missed because they correspond to subtle edge cases in the proof. Chor$\lambda$'s semantics tells us the following three programs will never evaluate $M_1$ if $f(\mathsf{r})$ diverges, but any compiler that uses Chor$\lambda$'s projection function *will* in fact evaluate $M_1$:

$$\mathsf{if}\ \begin{pmatrix}\mathsf{let}\ x = f(\mathsf{r})\ \mathsf{in}\\ \mathsf{true}@\mathsf{p}\end{pmatrix}\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 \qquad \begin{array}{l}\mathsf{let}\ x = f(\mathsf{r})\ \mathsf{in}\\ \mathsf{if}\ (\mathsf{select}_{\mathsf{r,s}}\ l\ \mathsf{true}@\mathsf{p})\\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2\end{array} \qquad \begin{array}{l}\mathsf{let}\ x = f(\mathsf{r})\ \mathsf{in}\\ \mathsf{let}\ y = (\mathsf{select}_{\mathsf{r,s}}\ l\ \mathsf{true}@\mathsf{p})\ \mathsf{in}\\ \mathsf{if}\ y\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2\end{array}$$

The fact that the *(commute)* rule guided us to these missing rules in Chor$\lambda$ is a testament to the usefulness of our approach.

### 3.4 Properties of $\lambda^\chi$

Applying *(app)* and *(commute)* reductions is similar to putting a term in Administrative Normal Form (ANF) [Flanagan et al. 1993]. We define normal forms like so:

*Definition 3.1.* A choreography $M$ is in *normal form* if it cannot be expressed in the form $\mathcal{E}_\mathsf{p}[\Delta]$ for any p, where $\Delta$ is a redex for *(app)* or *(commute)*.

Every closed choreography has a normal form. In the following, $M \overset{\tau}{\twoheadrightarrow} M'$ denotes a sequence of zero or more $\tau$ transitions $M \overset{\tau}{\to} \dots \overset{\tau}{\to} M'$:

Lemma 3.2. *For any closed choreography $M$, there exists $\tilde{M}$ in normal form where $M \overset{\tau}{\twoheadrightarrow} \tilde{M}$ using only (commute) and (app) reductions.*

Proof. Omitted for space. Complete proofs can be found in Appendix A. □

As we saw in Section 3.3.2, putting a choreography in normal form can expose redexes that allow processes to make progress. Normal forms are also simplify proofs, as we will see in Section 4.

Next, we establish an important lemma about the "next steps" a choreography can take. We will use the superscript $(-)^*$ to denote a stack of contexts—for example, $\mathcal{A}^*$ is either a hole $\bullet$ or an answering context filled with a stack $\mathcal{A}[\mathcal{A}^*]$.

*Definition 3.3.* A *redex at* p is a choreographic redex $\Delta$ where either (1) $\Delta \overset{\mu}{\longmapsto} \Delta'$ for some $\mu, \Delta'$ where $\mathsf{p} \in \mathsf{pn}(\mu)$, (2) $\Delta = \mathsf{com}_{\mathsf{q,p}}\ M$ for some $\mathsf{q}, M$, or (3) $\Delta = \mathsf{if}\ M\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$ for some $M, M_1, M_2$ where $\mathsf{p} \in (\mathsf{pn}(M_1) \cup \mathsf{pn}(M_2)) \setminus \mathsf{pn}(M)$.

$$
\begin{array}{rcll}
\text{if } (\text{let } x = M_1 \text{ in } M_2) \text{ then } M_3 \text{ else } M_4 & \mapsto & \text{let } x = M_1 \text{ in } (\text{if } M_2 \text{ then } M_3 \text{ else } M_4) & \textit{(if-let)} \\
\text{let } y = (\text{let } x = M_1 \text{ in } M_2) \text{ in } M_3 & \mapsto & \text{let } x = M_1 \text{ in } (\text{let } y = M_2 \text{ in } M_3) & \textit{(let-let)} \\
V\ (\text{let } x = M_1 \text{ in } M_2) & \mapsto & \text{let } x = M_1 \text{ in } V\ M_2 & \textit{(app-let)} \\
(\text{let } x = M_1 \text{ in } M_2)\ M_3 & \mapsto & \text{let } x = M_1 \text{ in } M_2\ M_3 & \textit{(let-app)} \\
V\ (\text{select}_{\mathsf{p,q}}\ l\ M) & \mapsto & \text{select}_{\mathsf{p,q}}\ l\ (V\ M) & \textit{(app-sel)} \\
(\text{select}_{\mathsf{p,q}}\ l\ M_1)\ M_2 & \mapsto & \text{select}_{\mathsf{p,q}}\ l\ (M_1\ M_2) & \textit{(sel-app)} \\
\text{if } (\text{select}_{\mathsf{p,q}}\ l\ M_1) \text{ then } M_2 \text{ else } M_3 & \mapsto & \text{select}_{\mathsf{p,q}}\ l\ (\text{if } M_1 \text{ then } M_2 \text{ else } M_3) & \textit{(if-sel)} \\
\text{let } x = \text{select}_{\mathsf{p,q}}\ l\ M_1 \text{ in } M_2 & \mapsto & \text{select}_{\mathsf{p,q}}\ l\ (\text{let } x = M_1 \text{ in } M_2) & \textit{(let-sel)}
\end{array}
$$

Fig. 8. This figure shows all the rules entailed by *(commute)* if we wrote them out explicitly.

Lemma 3.4. *Let $M$ be a closed choreography with $\mathsf{p} \in \text{pn}(M)$. Either:*

*(1) $M = \mathcal{E}_\mathsf{p}[\Delta]$ for some evaluation context $\mathcal{E}_\mathsf{p}$ where $\Delta$ is a redex at $\mathsf{p}$, or*
*(2) $M = \mathcal{A}^*[V]$ for some stack of answering contexts $\mathcal{A}^*$ where $\mathsf{p} \notin \text{pn}(\mathcal{A}^*)$ and $V$ is a value.*

The above result generalizes the usual property in deterministic languages, where every term is either a value or has an available redex. The next lemma implies that if $M$ is in normal form, the decomposition $M = \mathcal{E}_\mathsf{p}[\Delta]$ is unique for each process p.

Lemma 3.5. *Let $M$ be a closed choreography in normal form and let $\mathsf{p}, \mathsf{q}$ be processes that are not necessarily distinct. Assume $M = \mathcal{E}_\mathsf{p}[\Delta_1] = \mathcal{E}_\mathsf{q}[\Delta_2]$ for some evaluation contexts $\mathcal{E}_\mathsf{p}, \mathcal{E}_\mathsf{q}$ where $\Delta_1$ and $\Delta_2$ are redexes at both $\mathsf{p}$ and $\mathsf{q}$. Then $\mathcal{E}_\mathsf{p} = \mathcal{E}_\mathsf{q}$ and $\Delta_1 = \Delta_2$.*

We conclude by showing the type system for $\lambda^\chi$ is sound with respect to our semantics. This result, coupled with the Projection Theorem, will allow us to prove that projections are deadlock-free in Section 4.

Theorem 3.6 (Progress of evaluation). *Let $M$ be a choreography. If there exist $\Theta, T$ such that $\Theta; \emptyset \vdash M : T$, then either $M$ is a value $V$ or there exists $M'$ such that $M \xrightarrow{\mu} M'$.*

Theorem 3.7 (Preservation of evaluation). *Let $M$ be a choreography. If there exist $\Theta, \Gamma, T$ such that $\Theta; \Gamma \vdash M : T$, then $\Theta; \Gamma \vdash M' : T$ for any $M'$ such that $M \xrightarrow{\mu} M'$.*

## 4 Projection

Now the rubber meets the road: we present the *projection* function, which compiles $\lambda^\chi$ choreographies into networks. We will also show that any execution of the choreography can be matched by an execution of its projection (*completeness*) and any execution of the projection can be matched by the choreography (*soundness*). Together, these two properties are called the Projection Theorem.

Our proof strategy will take a significantly different route from prior work [Cruz-Filipe et al. 2023; Hirsch and Garg 2022; Montesi 2023]. We begin with a series of useful lemmas relating choreographic and network-level evaluation contexts. Then, instead of proving a direct correspondence between choreographies and their projections, we do it in two steps: first establishing a correspondence between choreographies and "superpowered" networks that can predict the future, and then showing that superpowered networks are equivalent to ordinary networks. The payoff for this extra work will be an elegant proof of the Projection Theorem and a clean correspondence (namely, a weak bisimulation) between $\lambda^\chi$ and the network language.

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

$$[\![\mathsf{let}\ x : T = M_1\ \mathsf{in}\ M_2]\!]_{\mathsf{p}} = \begin{cases} (\lambda x : [\![T]\!]_{\mathsf{p}}.[\![M_2]\!]_{\mathsf{p}})\ [\![M_1]\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(T) \cup \mathrm{pn}(M_1) \\ [\![M_2]\!]_{\mathsf{p}} & \text{otherwise} \end{cases}$$

$$[\![M_1\ M_2]\!]_{\mathsf{p}} = \begin{cases} [\![M_1]\!]_{\mathsf{p}}\ [\![M_2]\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(M_1) \cup \mathrm{pn}(M_2) \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\mathsf{if}\ M\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2]\!]_{\mathsf{p}} = \begin{cases} \mathsf{if}\ [\![M]\!]_{\mathsf{p}}\ \mathsf{then}\ [\![M_1]\!]_{\mathsf{p}}\ \mathsf{else}\ [\![M_2]\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(M)) \\ (\lambda x : \bot.[\![M_1]\!]_{\mathsf{p}} \sqcup [\![M_2]\!]_{\mathsf{p}})\ [\![M]\!]_{\mathsf{p}} & \text{else if } \mathsf{p} \in \mathrm{pn}(M) \\ [\![M_1]\!]_{\mathsf{p}} \sqcup [\![M_2]\!]_{\mathsf{p}} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{com}_{\mathsf{q},\mathsf{r}}]\!]_{\mathsf{p}} = \begin{cases} \mathsf{send}_{\mathsf{r}} & \text{if } \mathsf{p} = \mathsf{q} \\ \mathsf{recv}_{\mathsf{q}} & \text{if } \mathsf{p} = \mathsf{r} \\ \bot & \text{otherwise} \end{cases} \qquad [\![f(\mathsf{p}_1, \dots, \mathsf{p}_n)]\!]_{\mathsf{p}} = \begin{cases} f_i(\mathsf{p}_1, \dots, \mathsf{p}_{i-1}, \mathsf{p}_{i+1}, \dots, \mathsf{p}_n) & \text{if } \mathsf{p} = \mathsf{p}_i \\ \bot & \text{otherwise} \end{cases}$$

$$[\![c@\mathsf{r}]\!]_{\mathsf{p}} = \begin{cases} c & \text{if } \mathsf{p} = \mathsf{r} \\ \bot & \text{otherwise} \end{cases}$$

$$[\![\mathsf{select}_{\mathsf{q},\mathsf{r}}\ l\ M]\!]_{\mathsf{p}} = \begin{cases} \oplus_{\mathsf{r}}\ l\ [\![M]\!]_{\mathsf{q}} & \text{if } \mathsf{p} = \mathsf{q} \\ \&_{\mathsf{q}}\{l : [\![M]\!]_{\mathsf{r}}\} & \text{if } \mathsf{p} = \mathsf{r} \\ [\![M]\!]_{\mathsf{p}} & \text{otherwise} \end{cases} \qquad [\![x]\!]_{\mathsf{p}} = \begin{cases} x & \text{if } \mathsf{p} \in \mathrm{pn}(\mathrm{type}(x)) \\ \bot & \text{otherwise} \end{cases}$$

Operators:

$$\&_{\mathsf{q}}\{l_i : P_i\}_{i \in \mathcal{I}} \sqcup \&_{\mathsf{q}}\{l_j : P_j\}_{j \in \mathcal{J}} = \&_{\mathsf{q}}\{l_k : P_k\}_{k \in \mathcal{I} \cup \mathcal{J}} \qquad \text{if } \mathcal{I} \text{ and } \mathcal{J} \text{ are disjoint}$$

$$\bot \sqcup \bot = \bot$$

Types:

$$\left[\!\!\left[T_1 \to_{\overline{\mathsf{p}}} T_2\right]\!\!\right]_{\mathsf{p}} = \begin{cases} [\![T_1]\!]_{\mathsf{p}} \to [\![T_2]\!]_{\mathsf{p}} & \text{if } \mathsf{p} \in \mathrm{pn}(T_1) \cup \mathrm{pn}(T_2) \cup \overline{\mathsf{p}} \\ \bot & \text{otherwise} \end{cases} \qquad [\![\alpha@\mathsf{r}]\!]_{\mathsf{p}} = \begin{cases} \alpha & \text{if } \mathsf{p} = \mathsf{r} \\ \bot & \text{otherwise} \end{cases}$$

Fig. 9. Endpoint projection for $\lambda^{\chi}$.

### 4.1 The Projection Function

Projection is formally defined in Figure 9 by the relation $[\![M]\!]_{\mathsf{p}} = P$, where $P$ is the projection of $M$ for process p. The relation is defined by induction on the typing derivations of choreographies as in prior work [Cruz-Filipe et al. 2023; Hirsch and Garg 2022], though we omit the derivations in the figure for clarity. When the subscript is omitted, $[\![M]\!]$ is the parallel composition of the projections of the processes in the choreography, *i.e.*, $[\![M]\!] = \mathsf{p}_1\left[[\![M]\!]_{\mathsf{p}_1}\right] \mid \dots \mid \mathsf{p}_n\left[[\![M]\!]_{\mathsf{p}_n}\right]$ where $\mathrm{pn}(M) = \{\mathsf{p}_1, \dots, \mathsf{p}_n\}$.

Many of the cases in Figure 9 should be unsurprising. For example, $\mathsf{com}_{\mathsf{p},\mathsf{q}}$ is projected to $\mathsf{send}_{\mathsf{q}}$ for p and $\mathsf{recv}_{\mathsf{p}}$ for q. Notice also that, whenever a process p is not involved in $M$, the choreography is projected to the bottom value $\bot$. However, to understand the projection of an if-expression, readers unfamiliar with choreographic programming now need to be introduced to the concept of *projectability* [Cruz-Filipe and Montesi 2020; Montesi 2023]. Notice that Figure 9 contains the mapping:

$$[\![\mathsf{if}\ M\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2]\!]_{\mathsf{p}} = [\![M_1]\!]_{\mathsf{p}} \sqcup [\![M_2]\!]_{\mathsf{p}} \quad \text{when } \mathsf{p} \notin \mathrm{pn}(M). \tag{6}$$

This is because, if the expression $M$ is evaluated to a boolean at some q, there is no way for p to directly observe q's value; we say that p needs *knowledge of choice*. To obtain knowledge of choice,

$$\mathcal{B} ::= \oplus_{\mathrm{p}}\ l\ \mathcal{B} \mid \&_{\mathrm{p}}\{l : \mathcal{B}\} \mid (\lambda x : S.\mathcal{B})\ P \mid \bullet$$
$$\mathcal{O} ::= \mathcal{B}[\mathcal{O}] \mid \mathcal{E}[\mathcal{O}] \mid \bullet$$

Fig. 10. Degenerate contexts for networks.

q must explicitly inform p about its decision via selections $\mathsf{select}_{\mathrm{q,p}}\ l_1$ in $M_1$ and $\mathsf{select}_{\mathrm{q,p}}\ l_2\ M_2$, where $l_1$ and $l_2$ are distinct labels; these selections can be seen in Figure 4.

Choreographic languages ensure that knowledge of choice is propagated correctly via the *merge* operator ($\sqcup$) seen in Equation (6) and defined in Figure 9. The merge operator is a partial function, so the projection can be undefined if its criteria are not met. A choreography with no projection is said to be *unprojectable*. In the rest of this section, we will assume all choreographies are projectable unless otherwise stated. (Experts will notice our merge operator is more restrictive than in Chor$\lambda$; see Section 5 for details.)

### 4.2 Properties of Projection

We now establish some properties about projection and evaluation contexts, fulfilling the promises made in Section 3. All choreographies are assumed to be projectable and may contain free variables, unless otherwise stated.

Informally, the projection $[\![M]\!]_{\mathrm{p}}$ is always "the same as $M$, but with all the parts not related to p edited out". For example, values are projected to values and any $M$ where p is not involved will be projected to $\bot$. The latter result is used to establish modularity: editing the behavior of q should not change the projection for p if p $\neq$ q.

Lemma 4.1. *If $V$ is a choreographic value then $[\![V]\!]_{\mathrm{p}}$ is a network-level value.*

Lemma 4.2. *Let $M$ be a choreography where $\mathrm{p} \notin \mathrm{pn}(M)$. Then $[\![M]\!]_{\mathrm{p}} = \bot$.*

Lemma 4.3 (Modularity). *A context $C$ is a choreography with a unique hole $\bullet$ in place of some subexpression. Let $C$ be a context and $M_1, M_2$ choreographies such that $\mathrm{p} \notin \mathrm{pn}(M_1)$ and $\mathrm{p} \notin \mathrm{pn}(M_2)$. Then $[\![C[M_1]]\!]_{\mathrm{p}} = [\![C[M_2]]\!]_{\mathrm{p}}$.*

Below we establish novel properties of choreographic evaluation contexts. Lemma 4.4 is our "fundamental property of answering contexts": if a process r is not involved in $\mathcal{A}$, then the context will disappear during projection. Likewise, Lemma 4.5 is the "fundamental property of choreographic frames": if r is involved in the scrutinee of a choreographic frame $\mathcal{F}$, the projection will be a network-level frame, i.e., $[\![\mathcal{F}[M]]\!]_{\mathrm{r}} = \mathcal{F}[\,[\![M]\!]_{\mathrm{r}}\,]$ for some $\mathcal{F}$. Choreographic redexes $\Delta$ at r also project to network-level redexes $\delta$ at r, so long as $\Delta$ is not a *(commute)* or *(app)* redex, by Lemma 4.6. When the choreography is in normal form, these results imply that evaluation contexts containing redexes are projected to evaluation contexts containing redexes: $[\![\mathcal{E}_{\mathrm{r}}[\Delta]]\!]_{\mathrm{r}} = \mathcal{E}[\delta]$.

But before we can define these properties, there is a snag: we know $[\![\mathcal{E}_{\mathrm{r}}[M]]\!]_{\mathrm{p}}$ should be an evaluation context when r = p, but what about when r $\neq$ p? For this we must introduce *degenerate contexts* $\mathcal{B}, \mathcal{O}$ defined in Figure 10. These contexts will play an important role in Section 4.3, but for now we will simply assert that they emerge naturally in the lemmas below as a complement to evaluation contexts.

Lemma 4.4. *Let $\mathcal{A}$ be an answering context.*
*(1) If* $\mathsf{r} \notin \mathrm{pn}(\mathcal{A})$ *then* $[\![\mathcal{A}[M]]\!]_{\mathsf{r}} = [\![M]\!]_{\mathsf{r}}$ *for any* $M$.
*(2) If* $\mathsf{r} \in \mathrm{pn}(\mathcal{A})$ *then there exists* $\mathcal{B}$ *such that, for any* $M$, $[\![\mathcal{A}[M]]\!]_{\mathsf{r}} = \mathcal{B}[\,[\![M]\!]_{\mathsf{r}}\,]$.

Lemma 4.5. *Let* $M = \mathcal{F}[N]$ *where* $\Theta; \Gamma \vdash N : T$. *Then there exists a network-level frame* $\mathcal{F}$ *such that, for any* $N'$ *where* $\Theta; \Gamma \vdash N' : T$,
*(1) If* $\mathsf{r} \in \mathrm{pn}(N')$ *then* $\mathcal{F}[\,[\![N']\!]_{\mathsf{r}}\,] = [\![\mathcal{F}[N']]\!]_{\mathsf{r}}$.
*(2) If* $\mathsf{r} \notin \mathrm{pn}(N')$ *then* $\mathcal{F}[\,[\![N']\!]_{\mathsf{r}}\,] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![\mathcal{F}[N']]\!]_{\mathsf{r}}$.

Lemma 4.6. *Let* $\Delta$ *be a redex at* $\mathsf{r}$ *that is not an (app) or (commute) redex. Then* $[\![\Delta]\!]_{\mathsf{r}} = \delta$ *for some network-level redex* $\delta$.

Lemma 4.7. *Let* $M = \mathcal{E}_{\mathsf{r}}[N]$ *where* $\Theta; \Gamma \vdash N : T$. *Then there exists a network-level evaluation context* $\mathcal{E}$ *such that, for any* $N'$ *where* $\Theta; \Gamma \vdash N' : T$,
*(1) If* $\mathsf{r} \in \mathrm{pn}(N')$ *then* $\mathcal{E}[\,[\![N']\!]_{\mathsf{r}}\,] = [\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{r}}$.
*(2) If* $\mathsf{r} \notin \mathrm{pn}(N')$ *then* $\mathcal{E}[\,[\![N']\!]_{\mathsf{r}}\,] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{r}}$.

### 4.3 The Projection Theorem

How does a choreography $M$ relate to its projection $[\![M]\!]$? One might hope they directly correspond, so $M \xrightarrow{\mu} M'$ implies $[\![M]\!] \xrightarrow{\mu} [\![M']\!]$ and vice versa. As a pair of diagrams, where the vertical bar denotes endpoint projection:

$$\begin{array}{ccc} M & \xrightarrow{\mu} & M' \\ | & & | \\ [\![M]\!] & \overset{\mu}{\dashrightarrow} & [\![M']\!] \end{array} \qquad\qquad \begin{array}{ccc} M & \overset{\mu}{\dashrightarrow} & M' \\ | & & | \\ [\![M]\!] & \xrightarrow{\mu} & [\![M']\!] \end{array}$$

This would mean the projection function $[\![-]\!]$ is a *(strong) bisimulation.* But unfortunately projection is not a bisimulation, even in first-order models. Let us consider several reasons why.

*4.3.1 The Multistep Problem.* Certain computations can be done in one step choreographically, but require multiple steps at the network level. Take substitution for example, with types omitted for legibility:

$$\begin{array}{ccc} \mathsf{let}\ x = \mathsf{com}_{\mathsf{p},\mathsf{q}}\ \mathsf{in}\ x & \xrightarrow{\tau} & \mathsf{com}_{\mathsf{p},\mathsf{q}} \\ | & & | \\ \mathsf{p}\big[(\lambda x.\, x)\ \mathsf{send}_{\mathsf{q}}\big] \mid \mathsf{q}\big[(\lambda x.\, x)\ \mathsf{recv}_{\mathsf{p}}\big] & \overset{\tau}{\dashrightarrow}\!\!\!\!\rightarrow & \mathsf{p}\big[\mathsf{send}_{\mathsf{q}}\big] \mid \mathsf{q}\big[\mathsf{recv}_{\mathsf{p}}\big] \end{array}$$

Notice the projection in the bottom left corner is a pair of processes, with $\big[\!\big[\mathsf{let}\ x = \mathsf{com}_{\mathsf{p},\mathsf{q}}\ \mathsf{in}\ x\big]\!\big]_{\mathsf{p}} = (\lambda x.\, x)\ \mathsf{send}_{\mathsf{q}}$ and $\big[\!\big[\mathsf{let}\ x = \mathsf{com}_{\mathsf{p},\mathsf{q}}\ \mathsf{in}\ x\big]\!\big]_{\mathsf{q}} = (\lambda x.\, x)\ \mathsf{recv}_{\mathsf{p}}$ by the definition of projection in Figure 9. Whereas the choreography can reduce in one step using *(let)*, the network must take two steps: a *(p-app)* step at p and one at q.

The textbook solution is to add runtime-only terms [Montesi 2023; Plyukhin et al. 2024]: in the example above, we could have the choreography $M$ reduce to the "signposted" term $\mathsf{q}.M$ to indicate that q has taken a step but p has not. Signposting would allow us to prove strong correspondence results between choreographies and networks, but it clutters the syntax and semantics. To make $\lambda^{\chi}$ more accessible to newcomers, we avoid this route. We opt instead to exhibit a *weak bisimulation* between choreographies and their projections, so that for example $M \xrightarrow{\tau} M'$ implies $[\![M]\!] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![M']\!]$.

*4.3.2 The Choice Problem.* Choreographies have global information that is not immediately apparent at the process level. Consider what happens to the projection of the choreography $\mathsf{if}\ \mathsf{true@p}\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$, where $M_1 = \mathsf{select}_{\mathsf{p,q}}\ l\ \mathsf{true@q}$ and $M_2 = \mathsf{select}_{\mathsf{p,q}}\ l'\ \mathsf{false@q}$, after p evaluates the guard of the if-expression:

$$
\begin{array}{ccccc}
\mathsf{if}\ \mathsf{true@p}\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2 & \xrightarrow{\quad\tau\quad} & M_1 & \xrightarrow{\quad\tau\quad} & \mathsf{true@q} \\
\big| & & & & \big| \\
\mathsf{p}\big[\llbracket M \rrbracket_{\mathsf{p}}\big] \mid \mathsf{q}\big[\llbracket M_1 \rrbracket_{\mathsf{p}} \sqcup \llbracket M_2 \rrbracket_{\mathsf{p}}\big] & \overset{\tau}{\dashrightarrow} & \mathsf{p}\big[\llbracket M_1 \rrbracket_{\mathsf{p}}\big] \mid \mathsf{q}\big[\llbracket M_1 \rrbracket_{\mathsf{p}} \sqcup \llbracket M_2 \rrbracket_{\mathsf{p}}\big] & \overset{\tau}{\dashrightarrow} & \mathsf{p}\big[\bot\big] \mid \mathsf{q}\big[\mathsf{true}\big]
\end{array}
$$

There is an intermediate step where the choreography is $M_1 = \mathsf{select}_{\mathsf{p,q}}\ l\ \mathsf{true@q}$, but the network is not equal to $\llbracket M_1 \rrbracket$. Specifically, since the knowledge of p's choice has not yet propagated to q, the body of q is the *merge* $\llbracket M_1 \rrbracket_{\mathsf{q}} \sqcup \llbracket M_2 \rrbracket_{\mathsf{q}}$ instead of $\llbracket M_1 \rrbracket_{\mathsf{q}}$.

In general the choice problem means projection is not even a weak bisimulation, because nonterminating computations can prevent the network from ever propagating all the knowledge of choice available in the choreography. The textbook solution for this problem is to prove a weaker property, namely that $M \xrightarrow{\mu} M'$ and $\llbracket M \rrbracket \xrightarrow{\mu} \mathcal{N}$ implies $\mathcal{N}$ has "at least as many branches" as $\llbracket M' \rrbracket$, formalized by a relation $(\sqsubseteq)$ [Montesi 2023]. We will take a slightly different approach, incorporating the $(\sqsubseteq)$ relation as part of a more powerful relation introduced in Section 4.3.4.

*4.3.3 The Recursion Problem.* Choreographies in $\lambda^\chi$ do not necessarily terminate. Consider the program (with types omitted):

$$
\mathit{diverge}(\mathsf{p}) =_{\mathrm{def}} \mathit{diverge}(\mathsf{p})
$$

$$
M = \begin{pmatrix} \mathsf{let}\ x = \mathit{diverge}(\mathsf{p})\ \mathsf{in} \\ \mathsf{let}\ f = \mathsf{com}_{\mathsf{p,q}}\ \mathsf{in} \\ f \end{pmatrix},
$$

which has the projection:

$$
\begin{aligned}
\mathit{diverge}() &=_{\mathrm{def}} \mathit{diverge}() \\
\llbracket M \rrbracket_{\mathsf{p}} &= (\lambda x.\ (\lambda f.\ f)\ \mathsf{send}_{\mathsf{q}})\ \mathit{diverge}() \\
\llbracket M \rrbracket_{\mathsf{q}} &= (\lambda f.\ f)\ \mathsf{recv}_{\mathsf{p}}.
\end{aligned}
$$

Semilenient semantics allows us to evaluate $M = \mathcal{E}_{\mathsf{q}}[\mathsf{let}\ f = \mathsf{com}_{\mathsf{p,q}}\ \mathsf{in}\ f] \xrightarrow{\tau} \mathcal{E}_{\mathsf{q}}[\mathsf{com}_{\mathsf{p,q}}]$. This produces a term $M' = \mathcal{E}_{\mathsf{q}}[\mathsf{com}_{\mathsf{p,q}}]$ with the projection:

$$
\begin{aligned}
\llbracket M' \rrbracket_{\mathsf{p}} &= (\lambda x.\ \mathsf{send}_{\mathsf{q}})\ \mathit{diverge}() \\
\llbracket M' \rrbracket_{\mathsf{q}} &= \mathsf{recv}_{\mathsf{p}}
\end{aligned}
$$

Notice that indeed $\llbracket M \rrbracket_{\mathsf{q}} \mapsto \llbracket M' \rrbracket_{\mathsf{q}}$ is a legal transition in the network language, but $\llbracket M \rrbracket_{\mathsf{p}} \mapsto \llbracket M' \rrbracket_{\mathsf{p}}$ is not. In fact, p can never reduce the subterm $(\lambda f.\ f)\ \mathsf{send}_{\mathsf{q}}$ because the subterm $\mathit{diverge}()$ never terminates. Hence the choreography can take steps that its projection may never match.

Chor$\lambda$ solves the problem above by adding rewriting rules at the network level, allowing p to reduce the subterm $(\lambda f.\ f)\ \mathsf{send}_{\mathsf{q}}$ *out of order*. These extra rules are safe to add because they have no side-effects and because the $\lambda$-calculus is confluent, but it is unsatisfying to have rules in the semantics of processes that are only used for proofs.

*4.3.4 A Unified Approach.* We present a novel approach that handles all three of the above problems in one fell swoop. The idea is to introduce a *prophecy* relation[1] on networks $\mathcal{N} \rightsquigarrow \mathcal{N}'$, where $\mathcal{N}'$ is obtained by pruning branches in $\mathcal{N}$ and performing $\tau$-transitions—possibly out of order. Formally:

$$\begin{array}{rcll}
\mathsf{p}\big[\mathcal{O}[P]\big] \mid \mathcal{N} & \rightsquigarrow & \mathsf{p}\big[\mathcal{O}[P']\big] \mid \mathcal{N} & \text{if } P \mapsto P' \quad \textit{(compute)} \\
\mathsf{p}\big[\mathcal{O}[\&_{\mathsf{q}}\{l_i : P_i\}_{i\in\mathcal{I}}]\big] \mid \mathcal{N} & \rightsquigarrow & \mathsf{p}\big[\mathcal{O}[\&_{\mathsf{q}}\{l_i : P_i\}_{i\in\mathcal{J}}]\big] \mid \mathcal{N} & \text{if } \mathcal{J} \subseteq \mathcal{I} \quad \textit{(prune)} \\
\mathsf{p}\big[\mathcal{O}[\mathcal{F}[\mathcal{B}[P]]]\big] \mid \mathcal{N} & \rightsquigarrow & \mathsf{p}\big[\mathcal{O}[\mathcal{B}[\mathcal{F}[P]]]\big] \mid \mathcal{N} & \quad \textit{(commute)} \\
\mathsf{p}\big[\mathcal{O}[\bot\ P]\big] \mid \mathcal{N} & \rightsquigarrow & \mathsf{p}\big[\mathcal{O}[(\lambda x : \bot.\ \bot)\ P]\big] \mid \mathcal{N} & \quad \textit{(bottom}_2\textit{)}
\end{array}$$

where $\mathcal{O}$ denotes a degenerate context, as defined in Figure 10. As we saw in Sections 4.3.1 to 4.3.3, some steps at the choreography level require prophecy steps at the network level:

Lemma 4.8. *Let* $(\rightsquigarrow\!\!\!\!\rightarrow)$ *be the reflexive transitive closure of* $(\rightsquigarrow)$. *If* $M \xrightarrow{\tau}\!\!\!\!\rightarrow M'$ *only by (commute) and (app) reductions, then* $[\![M]\!] \rightsquigarrow\!\!\!\!\rightarrow [\![M']\!]$.

Lemma 4.9. *Let* $\mathsf{p}, \mathsf{r}$ *be roles such that* $\mathsf{p} \neq \mathsf{r}$. *Let* $M = \mathcal{E}_{\mathsf{r}}[N]$ *where* $\Theta; \Gamma \vdash N : T$. *Then there exists a degenerate context* $\mathcal{O}$ *such that, for any* $N'$ *where* $\Theta; \Gamma \vdash N' : T$,

*(1) If* $\mathsf{p} \in \mathrm{pn}(N')$ *then* $\mathcal{O}[\,[\![N']\!]_{\mathsf{p}}\,] = [\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{p}}$.
*(2) If* $\mathsf{p} \notin \mathrm{pn}(N')$ *then* $\mathcal{O}[\,[\![N']\!]_{\mathsf{p}}\,] \rightsquigarrow\!\!\!\!\rightarrow [\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{p}}$.

We will use the prophecy relation to capture the idea that a choreography "runs faster (and smarter!)" than its projection, but the projection can always take a finite number of prophecy steps to "catch up". Formally: our goal will be to show that the relation $\mathcal{N} \rightsquigarrow\!\!\!\!\rightarrow [\![M]\!]$ is a weak bisimulation between networks and choreographies.

We prove the result in two pieces. First, we prove the *Prophecy Theorem* (Theorem 4.10), which shows that prophecy steps commute with ordinary steps in the network. Second, we prove the *Projection Theorem* (Theorem 4.11), which shows that executions of choreographies and their choreographies correspond, mediated by the prophecy relation. Then we compose the two results to establish a weak bisimulation (Theorem 4.12). This proof technique is more modular than past work, which implicitly combines the Prophecy and Projection Theorems into one monolithic proof by induction [Cruz-Filipe et al. 2023; Montesi 2023].

We begin with the Prophecy Theorem. The completeness direction says that if a network $\mathcal{N}_1$ can "catch up" to a choreography's projection $[\![M]\!]$ using prophecy steps, then any step by $[\![M]\!]$ can be matched by $\mathcal{N}_1$ after first performing some $\tau$-transitions. Conversely, soundness means that any visible transition by $\mathcal{N}_1$ can be matched by the projection, and invisible transitions by $\mathcal{N}_1$ might not require the projection to take any steps at all.

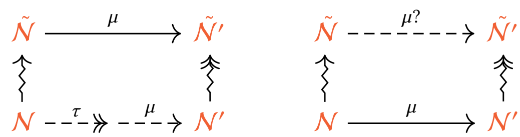
Theorem 4.10 (Prophecy Theorem). *Let* $M$ *be a choreography,* $\mathcal{N}$ *a network, and* $\mathcal{N} \rightsquigarrow\!\!\!\!\rightarrow [\![M]\!]$.

- *(Completeness) If* $[\![M]\!] \xrightarrow{\mu} \tilde{\mathcal{N}}'$ *then there exists* $\mathcal{N}'$ *such that* $\mathcal{N} \xrightarrow{\tau}\!\!\!\!\rightarrow\xrightarrow{\mu} \mathcal{N}' \rightsquigarrow\!\!\!\!\rightarrow \tilde{\mathcal{N}}'$.
- *(Soundness) If* $\mathcal{N} \xrightarrow{\mu} \mathcal{N}'$ *where* $\mu \neq \tau$ *then there exists* $\tilde{\mathcal{N}}'$ *such that* $[\![M]\!] \xrightarrow{\mu} \tilde{\mathcal{N}}'$ *and* $\mathcal{N}' \rightsquigarrow\!\!\!\!\rightarrow \tilde{\mathcal{N}}'$. *If* $\mathcal{N} \xrightarrow{\tau} \mathcal{N}'$ *then either* $\mathcal{N}' \rightsquigarrow\!\!\!\!\rightarrow [\![M]\!]$ *or there exists* $\tilde{\mathcal{N}}'$ *such that* $[\![M]\!] \xrightarrow{\tau} \tilde{\mathcal{N}}'$ *and* $\mathcal{N}' \rightsquigarrow\!\!\!\!\rightarrow \tilde{\mathcal{N}}'$.

Proof (sketch). Let $(\xrightarrow{\mu?})$ denote *exactly one step* if $\mu \neq \tau$ and *at most one step* if $\mu = \tau$. For the Completeness direction, it suffices to prove that if $\mathcal{N} \rightsquigarrow \tilde{\mathcal{N}} \xrightarrow{\mu} \tilde{\mathcal{N}}'$ then there exists $\mathcal{N}'$ such that $\mathcal{N} \xrightarrow{\tau}\!\!\!\!\rightarrow\xrightarrow{\mu} \mathcal{N}' \rightsquigarrow\!\!\!\!\rightarrow \tilde{\mathcal{N}}'$. For Soundness, it suffices to prove that if $\mathcal{N} \rightsquigarrow \tilde{\mathcal{N}} \rightsquigarrow\!\!\!\!\rightarrow [\![M]\!]$ and $\mathcal{N} \xrightarrow{\mu} \mathcal{N}'$ then

[1] The name for this relation is inspired by the related idea of *prophecy variables* [Abadi and Lamport 1991].

there exists $\tilde{\mathcal{N}}'$ such that $\tilde{\mathcal{N}} \xrightarrow{\mu?} \tilde{\mathcal{N}}'$ and $\mathcal{N}' \rightsquigarrow\!\!\!\!\twoheadrightarrow \tilde{\mathcal{N}}'$. As a pair of diagrams:




The proofs proceed by straightforward pattern matching of critical pairs [Barendregt 1984]. □

Next we establish the Projection Theorem. Unlike prior work, we mediate the correspondence between choreographies and their projection via the prophecy relation, and we use the properties of evaluation contexts established in Section 4.2. This extra structure leads to nice diagrammatic proofs, capturing our intuition for how the two models correspond.

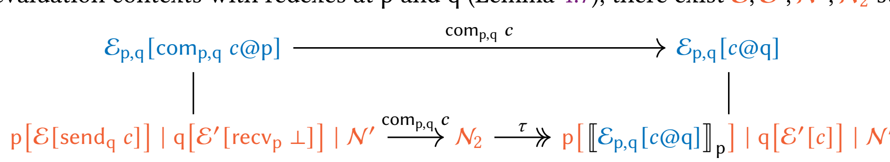
The completeness direction tells us that compiled code exhibits all the behaviors visible in the choreography. The soundness direction (which is typically much harder to prove) tells us that compiled code *only* exhibits behaviors from the choreography.

THEOREM 4.11 (PROJECTION THEOREM). *Let $M_1$ be a closed choreography.*

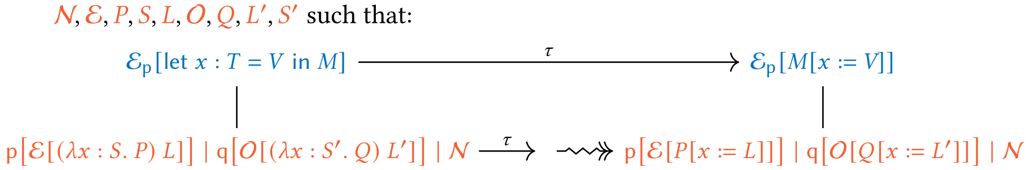
- *(Completeness) If $M_1 \xrightarrow{\mu} M_2$ is not an (app) or (commute) step, then $[\![M_1]\!] \xrightarrow{\mu}\rightsquigarrow\!\!\!\!\twoheadrightarrow [\![M_2]\!]$.*
- *(Soundness) If $[\![M_1]\!] \xrightarrow{\mu} \mathcal{N}_2$ then there exists $M_2$ such that $\mathcal{N}_2 \rightsquigarrow\!\!\!\!\twoheadrightarrow [\![M_2]\!]$ and $M_1 \overset{\tau}{\twoheadrightarrow}\xrightarrow{\mu} M_2$.*

PROOF (COMPLETENESS). Proceed by case analysis on $M_1 \xrightarrow{\mu} M_2$.

- Assume $\mu = \mathsf{com}_{\mathrm{p,q}}\ c$. Then $M_1 = \mathcal{E}_{\mathrm{p,q}}[\mathsf{com}_{\mathrm{p,q}}\ c]$. Since projection at p and q preserves evaluation contexts with redexes at p and q (Lemma 4.7), there exist $\mathcal{E}, \mathcal{E}', \mathcal{N}', \mathcal{N}_2$ such that:




  Specifically, $\mathcal{N}_2 = \mathrm{p}[\mathcal{E}[\bot]] \mid \mathrm{q}[\mathcal{E}'[c]] \mid \mathcal{N}'$. The only complication is that $\mathrm{p} \notin \mathrm{pn}(c@\mathrm{q})$, so $[\![\mathcal{E}_{\mathrm{p,q}}[c@\mathrm{q}]]\!]_{\mathrm{p}} \neq \mathcal{E}[\bot]$ in general; instead, Lemma 4.7 only guarantees process p can *catch up* to its projection via $\mathcal{E}[\bot] \overset{\tau}{\twoheadrightarrow} [\![\mathcal{E}_{\mathrm{p,q}}[c@\mathrm{q}]]\!]_{\mathrm{p}}$. At every other process $\mathrm{r} \notin \{\mathrm{p}, \mathrm{q}\}$, the projection is unchanged due to modularity (Lemma 4.3). Since $(\overset{\tau}{\twoheadrightarrow}) \subseteq (\rightsquigarrow\!\!\!\!\twoheadrightarrow)$, we have $[\![M_1]\!] \xrightarrow{\mu}\rightsquigarrow\!\!\!\!\twoheadrightarrow [\![M_2]\!]$.
- Assume $\mu = \mathsf{select}_{\mathrm{p,q}}\ l$. Similar to above.
- Assume $M_1 = \mathcal{E}_{\mathrm{p}}[\mathsf{let}\ x : T = V\ \mathsf{in}\ M]$ and that $M_2 = \mathcal{E}_{\mathrm{p}}[M[x := V]]$. Crucially, one must show that all processes in $\mathrm{pn}(V) \cup \mathrm{pn}(M)$ can match the choreography's step—not just p. For illustrative purposes, take $\mathrm{q} \in \mathrm{pn}(V) \setminus \{\mathrm{p}\}$. Then by Lemmas 4.1 and 4.7 there exist $\mathcal{N}, \mathcal{E}, P, S, L, \mathcal{O}, Q, L', S'$ such that:




  To complete the commuting diagram, p may need to catch up to its projection like above, and q may need to use the prophecy relation ($\rightsquigarrow$). Below we enumerate the four possible cases, and see that $[\![M_1]\!]_{\mathrm{q}} \rightsquigarrow\!\!\!\!\twoheadrightarrow [\![M_2]\!]_{\mathrm{q}}$ for each q:
  - For p, $[\![M_1]\!]_{\mathrm{p}} = \mathcal{E}[(\lambda x : [\![T]\!]_{\mathrm{p}}.\ [\![M]\!]_{\mathrm{p}})\ [\![V]\!]_{\mathrm{p}}]$. By Lemma A.9, $[\![M]\!]_{\mathrm{q}}[x := [\![V]\!]_{\mathrm{q}}] = [\![M[x := V]]\!]_{\mathrm{q}}$. Hence, by Lemma 4.7, $[\![M_1]\!]_{\mathrm{p}} \overset{\tau}{\twoheadrightarrow} [\![M_2]\!]_{\mathrm{p}}$.

– For each $\mathsf{q} \in \mathrm{pn}(V) \setminus \{\mathsf{p}\}$, there exists $\mathcal{O}$ such that $[\![M_1]\!]_{\mathsf{q}} = \mathcal{O}[(\lambda x : [\![T]\!]_{\mathsf{q}}.\ [\![M]\!]_{\mathsf{q}})\ [\![V]\!]_{\mathsf{q}}]$ by Lemma A.10. Then $[\![M_1]\!]_{\mathsf{q}} \rightsquigarrow \mathcal{O}[[\![M]\!]_{\mathsf{q}}[x := [\![V]\!]_{\mathsf{q}}]]$. Hence $[\![M_1]\!]_{\mathsf{q}} \rightsquigarrow\!\!\!\!\rightarrow [\![M_2]\!]_{\mathsf{q}}$.

– For each $\mathsf{q} \in \mathrm{pn}(M) \setminus \mathrm{pn}(V)$, there exists $\mathcal{O}$ such that $[\![M_1]\!]_{\mathsf{q}} = \mathcal{O}[[\![M]\!]_{\mathsf{q}}]$ by Lemma A.10. Hence $[\![M_1]\!]_{\mathsf{q}} = [\![M_2]\!]_{\mathsf{q}}$.

– For all remaining processes q, $[\![M_1]\!]_{\mathsf{q}} = [\![M_2]\!]_{\mathsf{q}}$ by Lemma 4.3.

- Assume $M_1 = \mathcal{E}_{\mathsf{p}}[M]$ where $M = \mathsf{if}\ c@\mathsf{p}\ \mathsf{then}\ M_{\mathsf{true}}\ \mathsf{else}\ M_{\mathsf{false}}$. Without loss of generality, let $c = \mathsf{true}$. Here we must show that every process $\mathsf{q} \in \mathrm{pn}(M_{\mathsf{true}}) \cup \mathrm{pn}(M_{\mathsf{false}})$ can match the choreography's step. Using the same arguments as above, there exist $\mathcal{N}, \mathcal{E}, \mathcal{O}, P, \mathcal{I}, \mathcal{J}$ where $\mathcal{J} \subseteq \mathcal{I}$ and there exist $l_i, Q_i$ for each $i \in \mathcal{I}$ such that:

$$\begin{array}{ccc}
\mathcal{E}_{\mathsf{p}}[M] & \xrightarrow{\quad\quad\quad\tau\quad\quad\quad} & \mathcal{E}_{\mathsf{p}}[M_{\mathsf{true}}] \\
| & & | \\
\mathsf{p}[\mathcal{E}[P]] \mid \mathsf{q}[\mathcal{O}[\&_{\mathsf{r}}\{l_i : Q_i\}_{i\in\mathcal{I}}]] \mid \mathcal{N} & \xrightarrow{\tau} \quad \rightsquigarrow & \mathsf{p}[\mathcal{E}[P_{\mathsf{true}}]] \mid \mathsf{q}[\mathcal{O}[\&_{\mathsf{r}}\{l_j : Q_j\}_{j\in\mathcal{J}}]] \mid \mathcal{N}
\end{array}$$

Here q can match the choreography's step by pruning its branches with the prophecy relation.

- Assume $M_1 = \mathcal{E}_{\mathsf{p}}[f(\overline{\mathsf{p}})]$ and $M_2 = \mathcal{E}_{\mathsf{p}}[M[\overline{\mathsf{p}} := \overline{\mathsf{q}}]]$, where $(f(\overline{\mathsf{q}}) : S = M) \in \mathbb{D}$. By the same arguments as above, $[\![M_1]\!] \xrightarrow{\tau}\rightsquigarrow\!\!\!\!\rightarrow [\![M_2]\!]$.

□

Proof (Soundness). Proof by case analysis on $[\![M_1]\!] \xrightarrow{\mu} \mathcal{N}_2$. By Lemmas 3.2 and 4.8, we can assume $M_1$ is in normal form without loss of generality.

- Assume $\mu = \mathsf{comp}_{\mathsf{p},\mathsf{q}}$. Then $[\![M_1]\!] = \mathsf{p}[\mathcal{E}[\mathsf{send}_{\mathsf{q}}\ c]] \mid \mathsf{q}[\mathcal{E}'[\mathsf{recv}_{\mathsf{p}}\ \bot]] \mid \mathcal{N}'$. At this point in the proof, we do not know anything about the structure of $M_1$—but we can use results established in Section 4.2 to discover that structure.
By Lemmas 3.4, 4.1 and 4.4, there exist $\mathcal{E}_{\mathsf{p}}, \Delta$ such that $M_1 = \mathcal{E}_{\mathsf{p}}[\Delta]$. Moreover, since $M_1$ is in normal form, $\Delta$ cannot be a *(commute)* or *(app)* redex. By Lemmas 4.6 and 4.7, and the uniqueness of evaluation contexts in the network language, $[\![\mathcal{E}_{\mathsf{p}}[\Delta]]\!]_{\mathsf{p}} = \mathcal{E}[[\![\Delta]\!]_{\mathsf{p}}]$ with $[\![\Delta]\!]_{\mathsf{p}} = \mathsf{send}_{\mathsf{q}}\ c$. Hence $\Delta$ can only be $\mathsf{com}_{\mathsf{p},\mathsf{q}}\ c@\mathsf{p}$. And by Lemma 3.5, $\mathcal{E}_{\mathsf{p}}$ must also be a choreographic evaluation context at q; let us rename $\mathcal{E}_{\mathsf{p}}$ to $\mathcal{E}_{\mathsf{p},\mathsf{q}}$. In summary, we have the following diagram:

$$\begin{array}{ccc}
\mathcal{E}_{\mathsf{p},\mathsf{q}}[\mathsf{com}_{\mathsf{p},\mathsf{q}}\ c@\mathsf{p}] & \xrightarrow{\quad\quad\quad\mathsf{com}_{\mathsf{p},\mathsf{q}}\ c\quad\quad\quad} & \mathcal{E}_{\mathsf{p},\mathsf{q}}[c@\mathsf{q}] \\
| & & | \\
\mathsf{p}[\mathcal{E}[\mathsf{send}_{\mathsf{q}}\ c]] \mid \mathsf{q}[\mathcal{E}'[\mathsf{recv}_{\mathsf{p}}\ \bot]] \mid \mathcal{N}' & \xrightarrow{\mathsf{com}_{\mathsf{p},\mathsf{q}}\ c} \mathcal{N}_2 \xrightarrow{\tau}\!\!\!\!\rightarrow & \mathsf{p}[[\![\mathcal{E}_{\mathsf{p},\mathsf{q}}[c@\mathsf{q}]]\!]_{\mathsf{p}}] \mid \mathsf{q}[\mathcal{E}'[c]] \mid \mathcal{N}'
\end{array}$$

In fact, this diagram is identical to the one we drew for the Completeness proof. Having established the correspondence between $\mathcal{E}_{\mathsf{p},\mathsf{q}}$ and $\mathcal{E}, \mathcal{E}'$, the rest of the proof proceeds the same as it did then. The case for selections is similar.

- Assume $\mu = \tau$ and the step proceeds by reducing a *(p-app)* redex at p. Then we must have $[\![M_1]\!]_{\mathsf{p}} = \mathcal{E}[(\lambda x : S.\ P)\ L]$. By Lemmas 3.4, 4.6 and 4.7, there exist $\mathcal{E}_{\mathsf{p}}, T, M, V$ such that $M_1 = \mathcal{E}_{\mathsf{p}}[\mathsf{let}\ x : T = V\ \mathsf{in}\ M_1']$. Letting $M_2 = \mathcal{E}_{\mathsf{p}}[M_1'[x := V]]$, the rest of the proof proceeds the same as the Completeness proof.

- Assume $\mu = \tau$ and the step proceeds by reducing a *(p-if)* redex at p. Then we must have $[\![M_1]\!]_{\mathsf{p}} = \mathcal{E}[\mathsf{if}\ c\ \mathsf{then}\ P_{\mathsf{true}}\ \mathsf{else}\ P_{\mathsf{false}}]$. Without loss of generality, let $c = \mathsf{true}$. By the arguments as above, there exist $\mathcal{E}_{\mathsf{p}}, M_{\mathsf{true}}, M_{\mathsf{false}}$ such that $M_1 = \mathcal{E}_{\mathsf{p}}[\mathsf{if\ true\ then}\ M_{\mathsf{true}}\ \mathsf{else}\ M_{\mathsf{false}}]$. This case now reduces to the same argument from the Completeness proof.
- Assume $\mu = \tau$ and the step proceeds by reducing a *(p-def)* redex at p. This case proceeds by the same arguments as above.

□

Theorem 4.12. *The relation $\mathcal{N} \rightsquigarrow\!\!\!\twoheadrightarrow [\![M]\!]$ is a weak bisimulation.*

Proof. *(Completeness)* Assume $\mathcal{N}_1 \rightsquigarrow\!\!\!\twoheadrightarrow [\![M_1]\!]$ and $M_1 \xrightarrow{\mu} M_2$. If $M_1 \xrightarrow{\tau} M_2$ by *(app)* or *(commute)*, then $[\![M_1]\!] \rightsquigarrow\!\!\!\twoheadrightarrow [\![M_2]\!]$ by Lemma 4.8; letting $\mathcal{N}_1 = \mathcal{N}_2$, we trivially have $\mathcal{N}_1 \overset{\tau}{\twoheadrightarrow} \mathcal{N}_2$ and $\mathcal{N}_2 \rightsquigarrow\!\!\!\twoheadrightarrow [\![M_2]\!]$. Otherwise, by the Projection Theorem there exists $\mathcal{N}_2'$ such that $[\![M_1]\!] \xrightarrow{\mu} \mathcal{N}_2' \rightsquigarrow\!\!\!\twoheadrightarrow [\![M_2]\!]$. By the Prophecy Theorem, there exists $\mathcal{N}_2$ such that $\mathcal{N}_1 \overset{\tau}{\twoheadrightarrow}\xrightarrow{\mu} \mathcal{N}_2$ and $\mathcal{N}_2 \rightsquigarrow\!\!\!\twoheadrightarrow \mathcal{N}_2'$. Hence $\mathcal{N}_1 \overset{\tau}{\twoheadrightarrow}\xrightarrow{\mu}\overset{\tau}{\twoheadrightarrow} \mathcal{N}_2$ and $\mathcal{N}_2 \rightsquigarrow\!\!\!\twoheadrightarrow [\![M_2]\!]$.

*(Soundness)* Assume $\mathcal{N}_1 \rightsquigarrow\!\!\!\twoheadrightarrow [\![M_1]\!]$ and $\mathcal{N}_1 \xrightarrow{\mu} \mathcal{N}_2$. By the Prophecy Theorem, there are two cases. In the first case, $\mu = \tau$ and $\mathcal{N}_2 \rightsquigarrow\!\!\!\twoheadrightarrow [\![M_1]\!]$; letting $M_1 = M_2$, we trivially have $M_1 \overset{\tau}{\twoheadrightarrow} M_2$ and $\mathcal{N}_2 \rightsquigarrow\!\!\!\twoheadrightarrow [\![M_2]\!]$. In the second case, there exists $\mathcal{N}_2'$ such that $\mathcal{N}_2 \rightsquigarrow\!\!\!\twoheadrightarrow \mathcal{N}_2'$ and $[\![M_1]\!] \xrightarrow{\mu} \mathcal{N}_2'$. By the Projection Theorem, there exists $M_2$ such that $M_1 \overset{\tau}{\twoheadrightarrow}\xrightarrow{\mu} M_2$ and $\mathcal{N}_2' \rightsquigarrow\!\!\!\twoheadrightarrow [\![M_2]\!]$. Hence $M_1 \overset{\tau}{\twoheadrightarrow}\xrightarrow{\mu}\overset{\tau}{\twoheadrightarrow} M_2$ and $\mathcal{N}_2 \rightsquigarrow\!\!\!\twoheadrightarrow [\![M_2]\!]$. □

With the prophecy relation, we eliminated the need for restructuring rules at the network level and made it convenient to prove the Projection Theorem by case analysis on evaluation contexts. Experts in choreographic programming will know that proving the Projection Theorem typically requires a large and tedious induction proof with many similar cases; our proof technique reduces the burden to just a few diagrams with no need for induction. We conclude with a victory lap: proving deadlock-freedom by design.

Theorem 4.13 (Deadlock-Freedom). *Let $M_0$ be a choreography with $\mathrm{pn}(M_0) = \{\mathsf{p}_1, \dots, \mathsf{p}_n\}$. If $[\![M_0]\!]$ evaluates to a network $\mathcal{N}$ that cannot be evaluated any further, then $\mathcal{N} = \mathsf{p}_1[L_1] \mid \dots \mid \mathsf{p}_n[L_n]$ where $L_1, \dots, L_n$ are all values.*

Proof. By Theorem 4.12, $M_0$ evaluates to some $M$ such that $\mathcal{N} \rightsquigarrow\!\!\!\twoheadrightarrow [\![M]\!]$. Let $\tilde{M}$ be the normal form of $M$, so that $M \overset{\tau}{\twoheadrightarrow} \tilde{M}$; by Lemma 4.8, $[\![M]\!] \rightsquigarrow\!\!\!\twoheadrightarrow [\![\tilde{M}]\!]$ and therefore $\mathcal{N} \rightsquigarrow\!\!\!\twoheadrightarrow [\![\tilde{M}]\!]$. Notice $\tilde{M}$ cannot be evaluated any further: by the Projection Theorem, another step $\tilde{M} \xrightarrow{\mu} M'$ would imply $[\![\tilde{M}]\!] \xrightarrow{\mu}\rightsquigarrow\!\!\!\twoheadrightarrow [\![M']\!]$, and so the Prophecy Theorem would imply $\mathcal{N} \overset{\tau}{\twoheadrightarrow}\xrightarrow{\mu} \tilde{\mathcal{N}}'$ for some $\tilde{\mathcal{N}}'$. Since $\tilde{M}$ cannot be evaluated further, Theorem 3.6 implies $\tilde{M}$ is a choreographic value $V$. Hence, by Lemma 4.1, $[\![\tilde{M}]\!]_{\mathsf{p}_i}$ is a value $L_i$ for each $\mathsf{p}_i \in \mathrm{pn}(M)$. Finally, by inspecting the definition of $(\rightsquigarrow)$, $\mathcal{N} \rightsquigarrow\!\!\!\twoheadrightarrow [\![\tilde{M}]\!]$ is only possible if $\mathcal{N} = [\![\tilde{M}]\!]$. □

## 5 Related Work

*Higher-Order Choreographies.* Higher-order choreographic programming was introduced by the Choral programming language [Giallorenzo et al. 2020, 2024]. Choral is also the language that introduced the idea of modeling choreographic data structures and communication by extending mainstream data types with locations and then having functions that input and output data at different locations. The theoretical foundations of this idea have been investigated in Chor$\lambda$ [Cruz-Filipe et al. 2022; Cruz-Filipe et al. 2023] and Pirouette [Hirsch and Garg 2022]. However, as we explained in Section 1, these models sacrifice either adequacy or elegance; $\lambda^{\chi}$ addresses both. Note that, although $\lambda^{\chi}$ is based on Chor$\lambda$, in principle we could develop the same results with Pirouette by changing its evaluation strategy.

Multiparty session types support "nested protocols", which can be seen as a form of higher-order composition for simple choreographies without computation [Demangeon and Honda 2012]. Differently from most higher-order choreographic programming languages and our approach, where code is fully concurrent via out-of-order execution, nested multiparty session types require fixing a role that acts as an orchestrator to direct when a sub-choreography is entered.

*Other Models.* Choreographies in our model have two slightly unusual features: explicit let-bindings and a special *(commute)* rule. Both features have a long history in past work.

Many readers will be familiar with Moggi's computational $\lambda$-calculus, where let-bindings express the sequencing of effects [Moggi 1991]. But explicit let-bindings also arise when embedding $\lambda$-calculi into proof nets, where $(\lambda x.M)\ N \mapsto \mathsf{let}\ x = N\ \mathsf{in}\ M$ and $\mathsf{let}\ x = V\ \mathsf{in}\ M \mapsto M[x := V]$ correspond to multiplicative and exponential cut-elimination, respectively [Accattoli 2015]. Ariola et al. [1995] [1989] also used let-bindings to model sharing in non-strict calculi, similarly to our model.

Most of the models above also require restructuring rules, like those in Figure 8. For instance, our *(let-let)* rule corresponds to monad associativity [Moggi 1991]; our *(let-app)* rule is needed for sharing in call-by-need [Ariola et al. 1995]; our *(let-app)* and *(app-let)* rules are needed to internally characterize solvability and achieve other good properties in call-by-value [Carraro and Guerrieri 2014; Herbelin and Zimmermann 2009]; and similar rules arise when embedding call-by-name terms into proof nets [Régnier 1994]. Maurer et al. [2017] observed that all the rules above can be neatly summarized by a single axiom, which pushes frames inside answering contexts. In choreographic models like $\lambda^{\chi}$ and Chor$\lambda$, the *(commute)* rule ensures soundness, i.e., choreographies can exhibit all the same behaviors as their projections.

*Distributed Data Types.* The full version of Chor$\lambda$ includes constructors for distributed products and sums—we omitted these constructors in pursuit of a simple model. As usual, one can partially recover these constructors with lambda encodings. Consider the standard encodings for products and sums, augmented with extra process annotations:

$$
\begin{aligned}
\mathsf{Pair} &\equiv \lambda x_1 : T_1.\ \lambda x_2 : T_2.\ \lambda p : T_1 \longrightarrow T_2 \longrightarrow_{\overline{\mathsf{p}}} T.\ p\ x_1\ x_2 \\
\mathsf{fst} &\equiv \lambda x_1 : T_1.\ \lambda x_2 : T_2.\ x_1 \\
\mathsf{snd} &\equiv \lambda x_1 : T_1.\ \lambda x_2 : T_2.\ x_2 \\
\mathsf{Inl} &\equiv \lambda x : T_1.\ \lambda l : T_1 \longrightarrow_{\overline{\mathsf{p}}} T.\ \lambda r : T_2 \longrightarrow_{\overline{\mathsf{q}}} T.\ l\ x \\
\mathsf{Inr} &\equiv \lambda x : T_2.\ \lambda l : T_1 \longrightarrow_{\overline{\mathsf{p}}} T.\ \lambda r : T_2 \longrightarrow_{\overline{\mathsf{q}}} T.\ r\ x \\
\mathsf{case} &\equiv \lambda s : (T_1 \longrightarrow_{\overline{\mathsf{p}}} T) \longrightarrow (T_2 \longrightarrow_{\overline{\mathsf{q}}} T) \longrightarrow_{\overline{\mathsf{p}} \cup \overline{\mathsf{q}}} T.\ \lambda l : T_1 \longrightarrow_{\overline{\mathsf{p}}} T.\ \lambda r : T_2 \longrightarrow_{\overline{\mathsf{q}}} T.\ s\ l\ r
\end{aligned}
$$

The principal limitation of this encoding is that the term constructing the datatype must dictate in advance what the type $T$ of the handler will be. In $\lambda^{\chi}$, this also means fixing a *location* for the type $T$ and a set of *proxy processes* $\overline{\mathsf{p}}$ involved in computing $T$. We can loosen these restrictions by adding explicit support for ADTs, as in Chor$\lambda$, or by adding process polymorphism, as in PolyChor$\lambda$ [Graversen et al. 2024].

*Selections.* HasChor [Shen et al. 2023] is a library for choreographic programming in Haskell. In HasChor, if-expressions and selections are merged into a single construct. When one writes $\mathsf{if}\ f(\mathsf{p})\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$, process p implicitly sends a selection to every other process in the choreography—including processes not involved in $M_1$ or $M_2$. Choreographic conclaves [Bates et al. 2025] improve on this mechanism, ensuring only processes involved in the branches will receive a selection. These approaches are useful in library-level choreographic programming, where it is difficult to statically check knowledge of choice.

However, explicit selections are more common in choreographic programming because they offer more control. For instance, in $\lambda^{\chi}$ we may write:

$$\begin{array}{l}\text{if } f(\text{p})\\ \text{then select}_{\text{p,q}}\ L_1\ M_1\\ \text{else} \begin{pmatrix}\text{if } g(\text{p})\\ \text{then select}_{\text{p,q}}\ L_2\ M_2\\ \text{else select}_{\text{p,q}}\ L_3\ M_3\end{pmatrix}\end{array}$$

which projects to the following process at q:

$$\&_{\text{q}}\{L_1 : [\![M_1]\!]_{\text{q}}, L_2 : [\![M_2]\!]_{\text{q}}, L_3 : [\![M_3]\!]_{\text{q}}\}$$

This term only requires one selection from p to q, whereas in HasChor it would require two selections—one for each if-expression. There has been significant work on making explicit selections even more powerful: [Lugovic and Montesi 2024] combines selections and communications into a single "type-driven communication"; [Cruz-Filipe et al. 2023b] coalesces all the selections of a loop into one message; and [Cruz-Filipe and Montesi 2023] develops an algorithm for inferring selections in first-order choreographies.

*Evaluation Under Conditionals.* One rule of Chor$\lambda$ we decided not to capture is *InCase*, which permits execution underneath an if-expression. This rule is also present in Montesi's textbook on choreographic programming, under the name *DelayCond* [Montesi 2023]. The inclusion of *InCase* is typically justified by a choreography like the following:

$$\text{if } c@\text{p then com}_{\text{p,q}}\ 0@\text{p else com}_{\text{p,q}}\ 1@\text{p},$$

This choreography is not projectable in $\lambda^{\chi}$, but it is projectable in Chor$\lambda$ and can be implemented by the network $\text{p}[\text{if } c \text{ then send}_{\text{q}}\ 0 \text{ else send}_{\text{q}}\ 1] \mid \text{q}[\text{recv}_{\text{p}}\ \perp]$. However, in simple examples like these we can make the choreography projectable in $\lambda^{\chi}$ by "pulling out" the communication:

$$\text{com}_{\text{p,q}}\ (\text{if } c@\text{p then } 0@\text{p else } 1@\text{p})$$

The future of the *InCase* rule is unclear. Library-level choreographic programming languages omit the rule because it would be difficult to encode in the host language [Bates et al. 2025; Shen et al. 2023]; recent work in fully out-of-order choreographies[2] is incompatible with the rule as it is currently understood [Plyukhin et al. 2024]; and recent extensions like multiply-located values [Bates et al. 2025] could further increase the expressiveness of $\lambda^{\chi}$ without requiring *InCase*.

## 6 Conclusion

We have presented $\lambda^{\chi}$, an elegant model for higher-order choreographic programming based on the $\lambda$-calculus. In particular, the model shows deep connections between choreographies and non-strict $\lambda$-calculi, culminating in a new evaluation strategy we call *semilenient* evaluation. We hope this work has made higher-order choreographies more accessible, particularly for experts from other fields. We conclude by discussing applications of $\lambda^{\chi}$ and opportunities for future work.

[2]In our terminology, "fully" out-of-order execution means giving choreographies a *lenient* semantics instead of semilenient.

*Implementing EPP and source-level reasoning.* To implement a higher-order choreographic language, compiler authors need to choose a projection function. If they use projection *à la* Pirouette, the language will have an ordinary call-by-value semantics—but only if processes synchronize globally when they enter choreographic procedures. The compiler author could choose to omit these synchronizations, but then the semantics is unknown: users will have no way to predict their programs' behavior except by inspecting the compiled endpoint code. To generate efficient code and retain a connection to some formal model, the compiler author could instead use projection *à la* Chor$\lambda$. But this is not much better, because users need to understand all Chor$\lambda$'s unusual rules and edge cases before they can understand their programs.

Our model simplifies Chor$\lambda$, revealing that its evaluation strategy is straightforward after all: it is just *semilenient*, instead of the usual call-by-value. This means compiler authors can omit needless global synchronizations guilt-free, and compiler users can reason about their programs using our simplified model. Compiler authors could also use our model to develop choreography-level optimizations, like eliminating unnecessary communications, without changing the behavior of the user's program.

*Compiler testing.* How do we know the code generated by a choreographic compiler is correct? Our Projection Theorem tells us what programmers should be able to expect: if $M$ is a choreography that reduces to value $V$ and the compiler is correct, then the code generated by the compiler should evaluate to $[\![V]\!]_{\mathsf{p}}$ at each role p. Moreover, the order of communications we observe in the compiled code should correspond to some execution of $M$ in the choreography semantics. This is a standard result that we can do with any choreography model.

But the principled design of our model also suggests *syntax-directed* ways to test a choreographic compiler. For example, we can test that Lemma 4.4 holds: answering contexts at p should disappear in the projection at p. We can also test Lemma 4.5: frames are projected into frames. If these properties hold, it suggests the compiler does not introduce any unintentional synchronization that would hurt performance.

*A principled foundation for new languages.* Choreographic programming is a very active field of study. Researchers and hobbyists alike are developing choreographic languages with novel features; these features are implemented by starting from an existing model, adding syntax, updating the type system, and implementing projection. But how can we gain confidence that the resulting language retains important properties like deadlock-freedom and type-safety? With prior work, we could only gain this confidence by first formalizing a new semantics *ex nihilo* and proving the projection theorem. Doing this is tedious and error-prone for researchers, and out of the question for working programmers.

Our model gives compiler authors *design principles* for new choreographic languages. We have argued already why we expect the semantics to be semilenient. We have also shown that semilenient semantics emerges naturally from the definition of answering contexts and choreographic frames. These two constructs have formal properties that are easy to check, c.f. Lemmas 4.4 and 4.5. Thus, language designers can already start to have confidence in new features by identifying appropriate generalizations of answering contexts and choreographic frames, and then checking that their projection algorithm satisfies those properties. Eventually, given the similarity of existing choreographic languages to one another, researchers may develop tools that automate the tedious proofs of properties like deadlock-freedom and the projection theorem.

*Future work.* We argued indirectly why $\lambda^{\chi}$ is a good foundation, but the proof is in the pudding. How well does our approach play with orthogonal extensions, like process polymorphism [Graversen et al. 2024], census polymorphism and multiply-located values [Bates et al. 2025], and fully out-of-order execution [Plyukhin et al. 2024]? Is the machinery we introduced sufficient for more complex models, and can it guide researchers toward the "right" abstractions?

Reducing proof burden is another important topic for future work. Although our approach makes proofs more "modular", one can still easily make mistakes. Perhaps the building blocks we introduced here could be factored into reusable proofs or tactics in proof assistants like Rocq or Lean.

## Acknowledgments

Partially supported by Villum Fonden (grant no. 29518). Co-funded by the European Union (ERC, CHORDS, 101124225). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or the European Research Council. Neither the European Union nor the granting authority can be held responsible for them.

## References

2025. Chorex: Choreographic Programming in Elixir. https://github.com/utahplt/chorex.

2025. Klor: Choreographies in Clojure. https://github.com/lovrosdu/klor.

Martín Abadi and Leslie Lamport. 1991. The Existence of Refinement Mappings. *Theor. Comput. Sci.* 82, 2 (1991), 253–284. doi:10.1016/0304-3975(91)90224-P

Cosku Acay, Rolph Recto, Joshua Gancher, Andrew C. Myers, and Elaine Shi. 2021. Viaduct: an extensible, optimizing compiler for secure distributed programs. In *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, Stephen N. Freund and Eran Yahav (Eds.). ACM, 740–755. doi:10.1145/3453483.3454074

Beniamino Accattoli. 2015. Proof Nets and the Call-by-Value Lambda-Calculus. *Theor. Comput. Sci.* 606 (2015), 2–24. doi:10.1016/J.TCS.2015.08.006

Zena Ariola and Arvind. 1989. P-TAC: A Parallel Intermediate Language. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture - FPCA '89*. ACM Press, Imperial College, London, United Kingdom, 230–242. doi:10.1145/99370.99388

Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. 1995. A Call-by-Need Lambda Calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL '95*. ACM Press, San Francisco, California, United States, 233–246. doi:10.1145/199448.199507

Arvind, Jan-Willem Maessen, Rishiyur S. Nikhil, and Joseph Stoy. 1996. A Lambda Calculus with Letrecs and Barriers. In *Foundations of Software Technology and Theoretical Computer Science*, Gerhard Goos, Juris Hartmanis, Jan Leeuwen, V. Chandru, and V. Vinay (Eds.). Vol. 1180. Springer Berlin Heidelberg, Berlin, Heidelberg, 19–36. doi:10.1007/3-540-62034-6_34

Arvind, Rishiyur S. Nikhil, and Keshav Pingali. 1986. I-Structures: Data Structures for Parallel Computing. In *Graph Reduction, Proceedings of a Workshop, Santa Fé, New Mexico, USA, September 29 - October 1, 1986 (Lecture Notes in Computer Science, Vol. 279)*, Joseph H. Fasel and Robert M. Keller (Eds.). Springer, 336–369. doi:10.1007/3-540-18420-1_65

Lennart Augustsson, Joachim Breitner, Koen Claessen, Ranjit Jhala, Simon Peyton Jones, Olin Shivers, Guy L. Steele Jr., and Tim Sweeney. 2023. The Verse Calculus: A Core Calculus for Deterministic Functional Logic Programming. *Proc. ACM Program. Lang.* 7, ICFP (2023), 417–447. doi:10.1145/3607845

Hendrik Pieter Barendregt. 1984. *The Lambda Calculus: Its Syntax and Semantics* (rev. ed ed.). Number v. 103 in Studies in Logic and the Foundations of Mathematics. North-Holland Sole distributors for the U.S.A. and Canada, Elsevier Science Pub. Co, Amsterdam New York New York, N.Y.

Mako Bates, Shun Kashiwa, Syed Jafri, Gan Shen, Lindsey Kuper, and Joseph P. Near. 2025. Efficient, Portable, Census-Polymorphic Choreographic Programming. In *PLDI '25: 46th ACM SIGPLAN International Conference on Programming Language Design and Implementation*. ACM, Seoul, South Korea. doi:10.1145/3729296

Marco Carbone and Fabrizio Montesi. 2013. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, Roberto Giacobazzi and Radhia Cousot (Eds.). ACM, 263–274. doi:10.1145/2429069.2429101

Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. 2018. Choreographies, logically. *Distributed Comput.* 31, 1 (2018), 51–67. doi:10.1007/S00446-017-0295-1

Alberto Carraro and Giulio Guerrieri. 2014. A Semantical and Operational Account of Call-by-Value Solvability. In *Foundations of Software Science and Computation Structures - 17th International Conference, FOSSACS 2014, Held as Part of* *the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings (Lecture Notes in Computer Science, Vol. 8412)*, Anca Muscholl (Ed.). Springer, 103–118. doi:10.1007/978-3-642-54830-7_7

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

Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. 2023. Modular Compilation for Higher-Order Functional Choreographies. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:37. doi:10.4230/LIPICS.ECOOP.2023.7

Luís Cruz-Filipe, Eva Graversen, Fabrizio Montesi, and Marco Peressotti. 2023a. Reasoning About Choreographic Programs. In *Coordination Models and Languages - 25th IFIP WG 6.1 International Conference, COORDINATION 2023, Held as Part of the 18th International Federated Conference on Distributed Computing Techniques, DisCoTec 2023, Lisbon, Portugal, June 19-23, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 13908)*, Sung-Shik Jongmans and Antónia Lopes (Eds.). Springer, 144–162. doi:10.1007/978-3-031-35361-1_8

Luís Cruz-Filipe and Fabrizio Montesi. 2020. A Core Model for Choreographic Programming. *Theor. Comput. Sci.* 802 (2020), 38–66. doi:10.1016/J.TCS.2019.07.005

Luís Cruz-Filipe and Fabrizio Montesi. 2023. Now It Compiles! Certified Automatic Repair of Uncompilable Protocols. In *14th International Conference on Interactive Theorem Proving, ITP 2023, July 31 to August 4, 2023, Białystok, Poland (LIPIcs, Vol. 268)*, Adam Naumowicz and René Thiemann (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 11:1–11:19. doi:10.4230/LIPICS.ITP.2023.11

Luís Cruz-Filipe, Fabrizio Montesi, and Robert R. Rasmussen. 2023b. Keep me out of the loop: a more flexible choreographic projection. In *LPAR 2023: Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Manizales, Colombia, 4-9th June 2023 (EPiC Series in Computing, Vol. 94)*, Ruzica Piskac and Andrei Voronkov (Eds.). EasyChair, 144–163. doi:10.29007/WBW3

Romain Demangeon and Kohei Honda. 2012. Nested Protocols in Session Types. In *CONCUR 2012 - Concurrency Theory - 23rd International Conference, CONCUR 2012, Newcastle upon Tyne, UK, September 4-7, 2012. Proceedings (Lecture Notes in Computer Science, Vol. 7454)*, Maciej Koutny and Irek Ulidowski (Eds.). Springer, 272–286. doi:10.1007/978-3-642-32940-1_20

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. *ACM SIGPLAN Notices* 28, 6 (June 1993), 237–247. doi:10.1145/173262.155113

Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2020. Choreographies as Objects. *CoRR* abs/2005.09520 (2020). arXiv:2005.09520 https://arxiv.org/abs/2005.09520

Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2024. Choral: Object-oriented Choreographic Programming. *ACM Trans. Program. Lang. Syst.* 46, 1 (2024), 1:1–1:59. doi:10.1145/3632398

Eva Graversen, Andrew K. Hirsch, and Fabrizio Montesi. 2024. Alice or Bob?: Process Polymorphism in Choreographies. *Journal of Functional Programming* 34 (Jan. 2024). doi:10.1017/S0956796823000114

Hugo Herbelin and Stéphane Zimmermann. 2009. An Operational Account of Call-by-Value Minimal and Classical Lambda-Calculus in "Natural Deduction" Form. In *Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009, Brasilia, Brazil, July 1-3, 2009. Proceedings (Lecture Notes in Computer Science, Vol. 5608)*, Pierre-Louis Curien (Ed.). Springer, 142–156. doi:10.1007/978-3-642-02273-9_12

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

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. *J. ACM* 63, 1 (2016), 9:1–9:67. doi:10.1145/2827695

Gérard Huet. 1997. The Zipper. *Journal of Functional Programming* 7, 5 (1997), 549–554. doi:10.1017/S0956796897002864

Shadaj Laddad, Alvin Cheung, and Joseph M. Hellerstein. 2024. Suki: Choreographed Distributed Dataflow in Rust. arXiv:2406.14733 [cs]

Lovro Lugovic and Fabrizio Montesi. 2024. Real-World Choreographic Programming: Full-Duplex Asynchrony and Interoperability. *Art Sci. Eng. Program.* 8, 2 (2024). doi:10.22152/PROGRAMMING-JOURNAL.ORG/2024/8/8

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon L. Peyton Jones. 2017. Compiling without Continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Albert Cohen and Martin T. Vechev (Eds.). ACM, 482–494. doi:10.1145/3062341.3062380

Eugenio Moggi. 1991. Notions of Computation and Monads. *Information and Computation* 93, 1 (July 1991), 55–92. doi:10.1016/0890-5401(91)90052-4

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

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

Roger M. Needham and Michael D. Schroeder. 1978. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM* 21, 12 (1978), 993–999. doi:10.1145/359657.359659

Object Management Group. 2017. Unified Modeling Language, Version 2.5.1. https://www.omg.org/spec/UML/2.5.1/PDF.

Gordon D. Plotkin. 1975. Call-by-Name, Call-by-Value and the lambda-Calculus. *Theor. Comput. Sci.* 1, 2 (1975), 125–159. doi:10.1016/0304-3975(75)90017-1

Dan Plyukhin, Marco Peressotti, and Fabrizio Montesi. 2024. Ozone: Fully Out-of-Order Choreographies. In *38th European Conference on Object-Oriented Programming, ECOOP 2024, September 16-20, 2024, Vienna, Austria (LIPIcs, Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 31:1–31:28. doi:10.4230/LIPICS.ECOOP.2024.31

Laurent Régnier. 1994. Une Équivalence Sur Les Lambda-Termes. *Theoretical Computer Science* 126, 2 (April 1994), 281–292. doi:10.1016/0304-3975(94)90012-4

Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). *Proc. ACM Program. Lang.* 7, ICFP (2023), 541–565. doi:10.1145/3607849

Guy Tremblay. 2000. Lenient Evaluation Is Neither Strict nor Lazy. *Comput. Lang.* 26, 1 (2000), 43–66. doi:10.1016/S0096-0551(01)00006-6

Petra van den Bos and Sung-Shik Jongmans. 2023. VeyMont: Parallelising Verified Programs Instead of Verifying Parallel Programs. In *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings (Lecture Notes in Computer Science, Vol. 14000)*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.). Springer, 321–339. doi:10.1007/978-3-031-27481-7_19

# A Appendix

## A.1 Properties of $\lambda^\chi$

We begin by showing that any choreography can be put in normal form (c.f. Definition 3.1) using only *(commute)* and *(app)* reductions. The key idea is to perform all possible *(commute)* reductions first, then contract all available *(app)* redexes, and repeat the process until the choreography is in normal form.

To show we can perform all possible *(commute)* reductions in finitely many steps, we need the notion of an evaluation context that cannot be extended:

*Definition A.1.* Let $M = \mathcal{E}_p[M']$ for some p and some $M, \mathcal{E}_p, M'$. We say $\mathcal{E}_p$ is *maximal* if:

(1) there is no $\mathcal{F}, M''$ such that $M' = \mathcal{F}[M'']$; and
(2) there is no $\mathcal{A}, M''$ such that $M' = \mathcal{A}[M'']$ and $p \notin pn(\mathcal{A})$.

Notice that for any $M$ and p there exists a unique maximal evaluation context $\mathcal{E}_p$.

*Definition A.2.* A choreography $M$ is in *commute-normal form* for p if the maximal evaluation context $\mathcal{E}_p$ has the form $\mathcal{A}^*[\mathcal{F}^*]$ for some $\mathcal{A}^*, \mathcal{F}^*$. In other words, $M$ cannot be decomposed in the form $\mathcal{E}'_p[\Delta]$ where $\Delta$ is a *(commute)*-redex. We also say $M$ is in commute-normal form for $\overline{p}$ if $M$ is in commute-normal form for each $p \in \overline{p}$.

Next, the following lemma tells us that a choreography can be placed in commute-normal form for any p using only *(commute)* reductions. If the choreography was already in commute-normal form for some $\overline{p}$, the reductions preserve that fact. Hence, we can perform all possible *(commute)* reductions in $M$ by picking a process p, putting $M$ in commute-normal form for p, and then doing the same for every other process in $M$.

Lemma A.3. *Let $\overline{p}$ be a (possibly empty) list of processes and let q be a process where $q \notin \overline{p}$. Let $M$ be a closed choreography in commute-normal form for $\overline{p}$. Then there exists $\tilde{M}$ such that $M \overset{\tau}{\twoheadrightarrow} \tilde{M}$ only using (commute) reductions, where $\tilde{M}$ is in commute-normal form for $\overline{p}$ and for q.*

Proof. Let $\mathcal{E}_q$ be an evaluation context. We begin by proving that for any $N$ where $\mathcal{E}_q[N]$ is in commute-normal form for $\overline{p}$, there exists $\tilde{\mathcal{E}}_q$ such that: (1) $\mathcal{E}_q[N] \overset{\tau}{\twoheadrightarrow} \tilde{\mathcal{E}}_q[N]$ using only *(commute)* reductions; (2) $\tilde{\mathcal{E}}_q[N]$ is in commute-normal form for $\overline{p}$; and (3) $\tilde{\mathcal{E}}_q = \mathcal{A}^*[\mathcal{F}^*]$ for some $\mathcal{A}^*, \mathcal{F}^*$ where $q \notin \mathrm{pn}(\mathcal{A}^*)$. We prove the result by induction on the structure of $\mathcal{E}_q$.

(1) If $\mathcal{E}_q = \bullet$ then $\tilde{\mathcal{E}}_q = \mathcal{E}_q$.

(2) Let $\mathcal{E}_q = \mathcal{E}'_q[\mathcal{F}]$. By the induction hypothesis, $\mathcal{E}'_q[\mathcal{F}[N]] \xrightarrow{\tau}\!\!\!\twoheadrightarrow \tilde{\mathcal{E}}'_q[\mathcal{F}[N]]$ using only *(commute)* reductions, where $\tilde{\mathcal{E}}'_q = \mathcal{A}^*[\mathcal{F}^*]$ and $q \notin \mathrm{pn}(\mathcal{A}^*)$. Then $\tilde{\mathcal{E}}_q = \tilde{\mathcal{E}}'_q[\mathcal{F}]$.

(3) Let $\mathcal{E}_q = \mathcal{E}'_q[\mathcal{A}]$, where $q \notin \mathrm{pn}(\mathcal{A})$. By the induction hypothesis, $\mathcal{E}'_q[\mathcal{A}[N]] \xrightarrow{\tau}\!\!\!\twoheadrightarrow \tilde{\mathcal{E}}'_q[\mathcal{A}[N]]$ using only *(commute)* reductions, where $\tilde{\mathcal{E}}'_q = \mathcal{A}^*[\mathcal{F}^*]$ and $q \notin \mathrm{pn}(\mathcal{A}^*)$.
Observe $\tilde{\mathcal{E}}'_q[\mathcal{A}[N]] \xrightarrow{\tau}\!\!\!\twoheadrightarrow \mathcal{A}^*[\mathcal{A}[\mathcal{F}^*[N]]]$ using only *(commute)* steps. Let $\tilde{\mathcal{E}}_q = \mathcal{A}^*[\mathcal{A}[\mathcal{F}^*]]$. It remains only to show $\tilde{\mathcal{E}}_q[N]$ is in commute-normal form for each $p \in \overline{p}$.
   1. Assume $p \notin \mathrm{pn}(\mathcal{A}^*)$. Then $\mathcal{A}^*[\mathcal{F}^*]$ is an evaluation context at p. Since we assumed $\tilde{\mathcal{E}}'_q[\mathcal{A}[N]]$ was in commute-normal form for p, there are two possibilities for $\mathcal{A}$:
      a. $\mathcal{A} = \mathsf{select}_{r,s}\ l\ \bullet$ where $p \in \{r, s\}$. Then $\tilde{\mathcal{E}}_q[N] = \mathcal{A}^*[\mathsf{select}_{r,s}\ l\ \mathcal{F}^*[N]]$, which is in commute-normal form for p.
      b. $\mathcal{A} = \mathsf{let}\ x : T = M'\ \mathsf{in}\ \bullet$ where $p \in \mathrm{pn}(M')$. Then $\tilde{\mathcal{E}}_q[N] = \mathcal{A}^*[\mathsf{let}\ x : T = M'\ \mathsf{in}\ \mathcal{F}^*[N]]$, which is in commute-normal form for p.
   2. Otherwise, $p \in \mathrm{pn}(\mathcal{A}^*)$. There are two cases:
      a. $\mathcal{A}^* = \mathcal{A}^*_1[\mathsf{select}_{r,s}\ l\ \mathcal{A}^*_2]$ where $p \in \{r, s\}$. Then $\tilde{\mathcal{E}}_q[N] = \mathcal{A}^*_1[\mathsf{select}_{r,s}\ l\ \mathcal{A}^*_2[\mathcal{A}[\mathcal{F}^*[N]]]]$ is still in commute-normal form for p.
      b. $\mathcal{A}^* = \mathcal{A}^*_1[\mathsf{let}\ x : T = M'\ \mathsf{in}\ \mathcal{A}^*_2]$ where $p \in \mathrm{pn}(M')$. Then $\tilde{\mathcal{E}}_q[N]$ is still in commute-normal form for p.

Now, let $\mathcal{E}_q$ be the maximal evaluation context such that $M = \mathcal{E}_q[N]$ for some $N$. As we showed above, there exists $\tilde{\mathcal{E}}_q$ such that (1) $\mathcal{E}_q[N] \xrightarrow{\tau}\!\!\!\twoheadrightarrow \tilde{\mathcal{E}}_q[N]$ using only *(commute)* reductions, (2) $\tilde{\mathcal{E}}_q[N]$ is in commute-normal form for $\overline{p}$, and (3) $\tilde{\mathcal{E}}_q = \mathcal{A}^*[\mathcal{F}^*]$ for some $\mathcal{A}^*, \mathcal{F}^*$ where $q \notin \mathrm{pn}(\mathcal{A}^*)$. Hence $\tilde{\mathcal{E}}_q[N]$ is in commute-normal form for $\overline{p}$ and q. ◻

We can now prove that every choreography has a normal form:

Lemma 3.2. *For any closed choreography $M$, there exists $\tilde{M}$ in normal form where $M \xrightarrow{\tau}\!\!\!\twoheadrightarrow \tilde{M}$ using only (commute) and (app) reductions.*

Proof. Let $\overline{p} = \mathrm{pn}(M)$. Notice that contracting an *(app)* redex can cause new *(commute)* redexes to appear, and vice versa. However, the number of *(app)* redexes is bounded above by the number of $\lambda$-abstractions that occur in $M$, and *(commute)* reductions do not change the number of $\lambda$-abstractions in the choreography. Hence we can put $M$ in normal form by first putting it in commute-normal form for p, then contracting all available *(app)* redexes, and repeating the process until no more *(app)* redexes can be contracted. ◻

Lemma 3.4. *Let $M$ be a closed choreography with $p \in \mathrm{pn}(M)$. Either:*

*(1) $M = \mathcal{E}_p[\Delta]$ for some evaluation context $\mathcal{E}_p$ where $\Delta$ is a redex at p, or*

*(2) $M = \mathcal{A}^*[V]$ for some stack of answering contexts $\mathcal{A}^*$ where $p \notin \mathrm{pn}(\mathcal{A}^*)$ and $V$ is a value.*

Proof. By structural induction on $M$.

(1) Let $M = V$ for some value $V$. Then $M = \mathcal{A}[V]$ where $\mathcal{A} = \bullet$.

(2) Let $M = f(\overline{p})$. Notice $p \in \overline{p}$ because we assumed $p \in \mathrm{pn}(M)$. Hence $M = \mathcal{E}_p[\Delta]$ where $\mathcal{E}_p = \bullet$ and $\Delta = M$.

(3) Let $M = \mathsf{select}_{q,r}\ l\ M'$.
   1. If $r \in \{q, r\}$ then $M = \mathcal{E}_p[\Delta]$ where $\mathcal{E}_p = \bullet$ and $\Delta = M$.

2. Otherwise, $\mathsf{r} \notin \{\mathsf{q}, \mathsf{r}\}$. By the induction hypothesis for $M'$, there are two cases:
   a. $M' = \mathcal{E}'_{\mathsf{p}}[\Delta]$ for some $\mathcal{E}'_{\mathsf{p}}, \Delta$. Then $M = \mathcal{E}_{\mathsf{p}}[\Delta]$ where $\mathcal{E}_{\mathsf{p}} = \mathcal{A}[\mathcal{E}'_{\mathsf{p}}]$ and $\mathcal{A} = \mathsf{select}_{\mathsf{q},\mathsf{r}}\ l\ \bullet$.
   b. $M' = \mathcal{A}'^*[V]$ for some $\mathcal{A}^*, V$. Then $M = \mathcal{A}^*[V]$ where $\mathcal{A}^* = \mathsf{select}_{\mathsf{q},\mathsf{r}}\ l\ \mathcal{A}'^*$.

(4) Let $M = \mathsf{let}\ x : T = M_1\ \mathsf{in}\ M_2$.

1. Assume $\mathsf{p} \notin \mathrm{pn}(M_1)$. Let $\mathcal{A} = \mathsf{let}\ x : T = M_1\ \mathsf{in}\ \bullet$. By the induction hypothesis for $M_2$, there are two cases outlined below.
   a. Assume $M_2 = \mathcal{E}'_{\mathsf{p}}[\Delta]$ for some $\mathcal{E}'_{\mathsf{p}}, \Delta$. Then $M = \mathcal{E}_{\mathsf{p}}[\Delta]$ where $\mathcal{E}_{\mathsf{p}} = \mathcal{A}[\mathcal{E}'_{\mathsf{p}}]$.
   b. Otherwise, $M_2 = \mathcal{A}^*[V]$ for some $\mathcal{A}^*, V$. Then $M = \mathcal{A}[\mathcal{A}^*[V]]$.
2. Otherwise, by the induction hypothesis for $M_1$, there are three cases outlined below.
   a. Assume $M_1 = \mathcal{E}'_{\mathsf{p}}[\Delta]$ for some $\mathcal{E}'_{\mathsf{p}}, \Delta$. Then $M = \mathcal{E}_{\mathsf{p}}[\Delta]$ where $\mathcal{E}_{\mathsf{p}} = \mathcal{F}[\mathcal{E}'_{\mathsf{p}}]$ and $\mathcal{F} = \mathsf{let}\ x : T = \bullet\ \mathsf{in}\ M_2$.
   b. Assume $M_1 = V$ for some $V$. Then $M = \mathcal{E}_{\mathsf{p}}[\Delta]$ where $\mathcal{E}_{\mathsf{p}} = \bullet$ and $\Delta = M$.
   c. Otherwise, $M_1 = \mathcal{A}^*[V]$ for some $\mathcal{A}^*, V$ where $\mathcal{A}^* \neq \bullet$. Then for some $\mathcal{A}, \mathcal{A}'^*$ we have $\mathcal{A}^* = \mathcal{A}[\mathcal{A}'^*]$. Hence $M = \mathcal{F}[\mathcal{A}[\mathcal{A}'^*[V]]]$, where $\mathcal{F} = \mathsf{let}\ x : T = \bullet\ \mathsf{in}\ M_2$; i.e., $M$ is a redex for the *(commute)* rule. Hence $M = \mathcal{E}_{\mathsf{p}}[\Delta]$ where $\mathcal{E}_{\mathsf{p}} = \bullet$ and $\Delta = M$.

(5) Let $M = M_1\ M_2$. We show $M = \mathcal{E}_{\mathsf{p}}[\Delta]$ for some $\mathcal{E}_{\mathsf{p}}, \Delta$ by induction on the typing derivation of $M_1$.

1. Case TAbs. Then $M$ is a redex for *(app)*.
2. Case TVar. Impossible, because $M_1$ is closed.
3. Case TIf, TDef. Impossible, because we assumed if-expressions and procedure calls are let-bound or in tail position.
4. Case TSel, TLet. Then $M = \mathcal{F}[\mathcal{A}[M']]$ is a redex for *(commute)*, where $M_1 = \mathcal{A}[M']$ and $\mathcal{F} = \bullet\ M_2$.
5. Case TConst. Impossible, because $M_1$ has function type.
6. Case TApp. By the induction hypothesis, $M_1 = \mathcal{E}'_{\mathsf{p}}[\Delta]$ for some $\mathcal{E}'_{\mathsf{p}}, \Delta$. Then $M = \mathcal{F}[\mathcal{E}'_{\mathsf{p}}[\Delta]]$ where $\mathcal{F} = \bullet\ M_2$.
7. Case TCom. If $M_1 = \mathsf{com}_{\mathsf{q},\mathsf{p}}$ for some q, then $M$ is a redex at p. Otherwise, $M_1 = \mathsf{com}_{\mathsf{q},\mathsf{r}}$ for some q, r and $\mathsf{p} \in \mathrm{pn}(M_2)$. By the induction hypothesis for $M_2$, there are three cases outlined below.
   a. Assume $M_2 = \mathcal{E}'_{\mathsf{p}}[\Delta]$ for some $\mathcal{E}'_{\mathsf{p}}, \Delta$. Then $M = \mathcal{E}_{\mathsf{p}}[\Delta]$ where $\mathcal{E}_{\mathsf{p}} = \mathcal{F}[\mathcal{E}'_{\mathsf{p}}]$ and $\mathcal{F} = M_1\ \bullet$.
   b. Assume $M_2$ is a value. Then $M$ is a redex for *(com)*.
   c. Otherwise, $M_2 = \mathcal{A}^*[V]$ for some $\mathcal{A}^*, V$ where $\mathcal{A}^* \neq \bullet$. Then $M$ is a redex for *(commute)*.

(6) Let $M = \mathsf{if}\ M'\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$.

1. Assume $\mathsf{p} \notin \mathrm{pn}(M')$. Then $\mathsf{p} \in \mathrm{pn}(M)$ implies $\mathsf{p} \in \mathrm{pn}(M_1) \cup \mathrm{pn}(M_2)$. Hence $M$ is a redex for p.
2. Otherwise, $\mathsf{p} \in \mathrm{pn}(M')$. By the induction hypothesis for $M'$, there are three cases:
   a. Assume $M' = \mathcal{E}'_{\mathsf{p}}[\Delta]$ for some $\mathcal{E}'_{\mathsf{p}}, \Delta$. Then $M = \mathcal{E}_{\mathsf{p}}[\Delta]$ where $\mathcal{E}_{\mathsf{p}} = \mathcal{F}[\mathcal{E}'_{\mathsf{p}}]$ and $\mathcal{F} = \mathsf{if}\ \bullet\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$.
   b. Assume $M'$ is a value. By inspecting the typing rules, $M'$ must be $\mathsf{true@p}$ or $\mathsf{false@p}$. Hence $M$ is a redex for *(if)*.
   c. Otherwise, $M' = \mathcal{A}^*[V]$ for some $\mathcal{A}^*, V$ where $\mathcal{A}^* \neq \bullet$. Then $M$ is a redex for *(commute)*.

□

Lemma 3.5. *Let $M$ be a closed choreography in normal form and let* p, q *be processes that are not necessarily distinct. Assume $M = \mathcal{E}_{\mathsf{p}}[\Delta_1] = \mathcal{E}_{\mathsf{q}}[\Delta_2]$ for some evaluation contexts $\mathcal{E}_{\mathsf{p}}, \mathcal{E}_{\mathsf{q}}$ where $\Delta_1$ and $\Delta_2$ are redexes at both* p *and* q. *Then $\mathcal{E}_{\mathsf{p}} = \mathcal{E}_{\mathsf{q}}$ and $\Delta_1 = \Delta_2$.*

Proof. By induction on the structure of $\mathcal{E}_{\mathsf{p}}$. Notice it suffices to show $\mathcal{E}_{\mathsf{p}} = \mathcal{E}_{\mathsf{q}}$.

(1) Assume $\mathcal{E}_p = \bullet$. By case analysis on redexes $\Delta_1$, the assumption that $\Delta_1$ is also a redex at q, and the assumption that $M$ is in normal form, we must have $\mathcal{E}_q = \bullet$.
(2) Assume $\mathcal{E}_p = \mathcal{F}[\mathcal{E}'_p]$.
  1. Assume $\mathcal{F} = \bullet\ M'$. Proceed by case analysis on $\mathcal{E}_q$.
    a. Suppose $\mathcal{E}_q = \bullet$. Then $M = \Delta_2$, where $\Delta_2$ is not an *(app)* or *(commute)* redex because $M$ is in normal form. By case analysis on choreographic redexes, this case is impossible because $\mathcal{E}'_p[\Delta_1]$ cannot be a value.
    b. Suppose $\mathcal{E}_q = V\,\mathcal{E}'_q$ for some $\mathcal{E}'_q$. Again, $\mathcal{E}'_p[\Delta_1]$ cannot be a value.
    c. Hence $\mathcal{E}_q = \mathcal{F}[\mathcal{E}'_q]$ for some $\mathcal{E}'_q$. The result follows by the induction hypothesis.
  2. Assume $\mathcal{F} = V\,\bullet$.
    a. Suppose $\mathcal{E}_q = \bullet$. Then $M = \Delta_2$, which is impossible because $\mathcal{E}'_p[\Delta_1]$ cannot be a value.
    b. Suppose $\mathcal{E}_q = \mathcal{E}'_q\ M'$ for some $\mathcal{E}'_q, M'$. Impossible because $\mathcal{E}'_q[\Delta_2]$ cannot be a value $V$.
    c. Hence $\mathcal{E}_q = \mathcal{F}[\mathcal{E}'_q]$ for some $\mathcal{E}'_q$. The result follows by the induction hypothesis.
  3. Assume $\mathcal{F} = \mathsf{if}\ \bullet\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$.
    a. Suppose $\mathcal{E}_q = \bullet$. Then $M = \Delta_2$, which is impossible because $\mathcal{E}'_p[\Delta_1]$ cannot be a value.
    b. Hence $\mathcal{E}_q = \mathcal{F}[\mathcal{E}'_q]$ for some $\mathcal{E}'_q$. The result follows by the induction hypothesis.
  4. Assume $\mathcal{F} = \mathsf{let}\ x : T = \bullet\ \mathsf{in}\ M'$.
    a. Suppose $\mathcal{E}_q = \bullet$. Then $M = \Delta_2$, which is impossible because $\mathcal{E}'_p[\Delta_1]$ cannot be a value.
    b. Suppose $\mathcal{E}_q = \mathsf{let}\ x : T = M''\ \mathsf{in}\ \mathcal{E}'_q$ for some $M'', \mathcal{E}'_q$ where $q \notin \mathrm{pn}(M'')$. Impossible, because $M'' = \mathcal{E}'_p[\Delta_1]$ and $\Delta_1$ is a redex at q, so $q \in \mathrm{pn}(M'')$.
    c. Hence $\mathcal{E}_q = \mathcal{F}[\mathcal{E}'_q]$ for some $\mathcal{E}'_q$. The result follows by the induction hypothesis.
(3) Assume $\mathcal{E}_p = \mathcal{A}[\mathcal{E}'_p]$ where $p \notin \mathrm{pn}(\mathcal{A})$.
  1. Assume $\mathcal{A} = \mathsf{let}\ x : T = M'\ \mathsf{in}\ \bullet$ where $p \notin \mathrm{pn}(M')$.
    a. Suppose $\mathcal{E}_q = \bullet$. Then $M = \Delta_2$, which is impossible because $\mathcal{E}'_p[\Delta_1]$ cannot be a value.
    b. Suppose $\mathcal{E}_q = \mathsf{let}\ x : T = \bullet\ \mathsf{in}\ M''$ for some $M'', \mathcal{E}'_q$. Impossible, because $M' = \mathcal{E}'_q[\Delta_2]$ and $\Delta_2$ is a redex at p, implying $p \in \mathrm{pn}(M')$ after all.
    c. Hence $\mathcal{E}_q = \mathcal{A}[\mathcal{E}'_q]$ for some $\mathcal{E}'_q$. The result follows by the induction hypothesis.
  2. Assume $\mathcal{A} = \mathsf{select}_{r,s}\ l\ \bullet$ where $p \notin \{r, s\}$.
    a. Suppose $q \in \{r, s\}$. Impossible, because $M = \Delta_2$ and $\Delta_2$ is a redex at p, implying $p \in \{r, s\}$ after all.
    b. Hence $\mathcal{E}_q = \mathcal{A}[\mathcal{E}'_q]$ for some $\mathcal{E}'_q$. The result follows by the induction hypothesis.

□

Theorem 3.6 (Progress of evaluation). *Let $M$ be a choreography. If there exist $\Theta, T$ such that $\Theta; \emptyset \vdash M : T$, then either $M$ is a value $V$ or there exists $M'$ such that $M \xrightarrow{\mu} M'$.*

Proof. Shown by different cases of $M$.

(1) If $M$ is a redex or a value, the result holds trivially.
(2) $M = f(\overline{p})$. Then $M$ is a redex for the *(def)* rule.
(3) $M = \mathsf{let}\ x : T_1 = M_1\ \mathsf{in}\ M_2$. If $M_1$ is a value, then $M$ is a redex for the *(let)* rule. If $M_1$ is not a value, then by the inversion of the rule TLet, we can obtain $\Theta; \emptyset \vdash M_1 : T_1$. By the inductive hypothesis, there exists $M'_1$ such that $M_1 \xrightarrow{\mu} M'_1$. Then also $M = \mathcal{F}[M_1] \xrightarrow{\mu} \mathcal{F}[M'_1]$, where $\mathcal{F} = \mathsf{let}\ x : T_1 = \bullet\ \mathsf{in}\ M_2$.
(4) $M = \mathsf{select}_{p,q}\ l\ M_1$. Then $M$ is a redex for the *(select)* rule.
(5) $M = M_1\ M_2$.
  1. $M_1 = V$ is a value, and therefore $M = \mathcal{F}[M_2]$ where $\mathcal{F} = V\,\bullet$. By the inversion of the rule TApp, we can obtain $\Theta; \emptyset \vdash V : T_1 \rightarrow_{\overline{p}} T$, and $\Theta; \emptyset \vdash M_1 : T_1$. Based on the typing information of $V$, it can only take two different forms:

a. $V = \lambda x : T.M_2$, then $M$ is a redex for the *(app)* rule.
b. $V = \mathsf{com}_{p,q}$, then we have two cases:
(1) $M_2 = V'$ is a value. Since $\Theta; \emptyset \vdash M_2 : T_1$ and $M$ is closed, $M_2$ can only be a constant value. Then $M$ is a redex for the *(com)* rule.
(2) $M_2$ is not a value. By the inductive hypothesis, there exists $M_2'$ such that $M_2 \xrightarrow{\mu} M_2'$. Then also $M = \mathcal{F}[M_2] \xrightarrow{\mu} \mathcal{F}[M_2']$.

2. $M_1$ is not a value, and therefore $M = \mathcal{F}[M_1]$ where $\mathcal{F} = \bullet\ M_2$. By the inversion of the rule TApp, we have $\Theta; \emptyset \vdash M_1 : T_1 \rightarrow_{\overline{p}} T$. Then by the inductive hypothesis, we can prove this case.

(6) $M = \mathsf{if}\ M_1\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3$. By the inversion of the rule TIf, we have $\Theta; \Gamma \vdash M_1 : \mathsf{Bool@p}$, $\Theta; \Gamma \vdash M_2 : T$, and $\Theta; \Gamma \vdash M_3 : T$.
1. $M_1 = V$. Since $\Theta; \Gamma \vdash M_1 : \mathsf{Bool@p}$ and $M$ is closed, $M_1$ can only be a constant value. Then $M$ is a redex for the *(if)* rule.
2. $M_1$ is not a value, and therefore $M = \mathcal{F}[M_1]$ where $\mathcal{F} = \mathsf{if}\ \bullet\ \mathsf{then}\ M_2\ \mathsf{else}\ M_3$. Then by the inductive hypothesis, we can prove this case.

□

Lemma A.4 (Variable substitution). *Let $M$ be a term with a free variable $x$, and suppose $\Theta; \Gamma, x : T_2 \vdash M : T_1$. Also, let $V$ be a value with type $T_2$. Then, we can substitute $V$ for all occurrences of $x$ in $M$ and obtain a new term that still has type $T_1$, i.e., $\Theta; \Gamma \vdash M[x := V] : T_1$.*

Proof. This lemma can be proven by structural induction on $M$. □

Lemma A.5 (Process name substitution). *Let $M$ be a term such that $\Theta; \Gamma \vdash M : T$, then $\Theta; \Gamma \vdash M[\overline{q} := \overline{p}] : T[\overline{q} := \overline{p}]$.*

Proof. Similarly, this lemma can be proven by structural induction on $M$. □

Theorem A.6 (Preservation of reduction). *Let $M$ be a choreography. If there exist $\Theta, \Gamma, T$ such that $\Theta; \Gamma \vdash M : T$, then $\Theta; \Gamma \vdash M' : T$ for any $M'$ such that $M \overset{\mu}{\longmapsto} M'$.*

Proof. By induction on reduction rules.

(1) Case *com*. We can prove it by the inversion of the rule TCom.
(2) Case *select*. We can prove it by the inversion of the rule TSel.
(3) Case *app*. We can prove it by the inversion of the rule TApp, and then applying the rule TLet.
(4) Case *let*. In this case $M' = M_2[x := V]$. We can prove it by firstly applying the inversion of the rule TLet, we have $\Theta; \Gamma \vdash V : T_1$ and $\Theta; \Gamma, x : T_1 \vdash M_2 : T_2$. Then by lemma A.4, we can conclude that $\Theta; \Gamma \vdash M_2[x := V] : T_2$ and prove this case.
(5) Case *if*. We can prove it by firstly applying the inversion of the rule TIf, we have $\Theta; \Gamma \vdash M_1 : \mathsf{Bool@p}$, $\Theta; \Gamma \vdash M_2 : T$, and $\Theta; \Gamma \vdash M_3 : T$. If $M_1$ is evaluated to true, then we can prove this case by having $\Theta; \Gamma \vdash M_2 : T$ and $M' = M_2$. If $M_1$ is evaluated to false, then we can prove this case by having $\Theta; \Gamma \vdash M_3 : T$ and $M' = M_3$.
(6) Case *def*. In this case $M' = M_1[\overline{q} := \overline{p}]$, where $(f(\overline{q}) : T_1 = M_1) \in \mathbb{D}$. We would like to show $\Theta; \Gamma \vdash M_1[\overline{q} := \overline{p}] : T$. We have two typing judgments $\Theta; \Gamma \vdash M_1 : T_1$ and $\Theta; \Gamma \vdash f(\overline{p}) : T$. By the rule TDef, we can obtain $\Theta; \Gamma \vdash f(\overline{p}) : T_1[\overline{q} := \overline{p}]$. Therefore we can conclude $T = T_1[\overline{q} := \overline{p}]$. Since $\Theta; \Gamma \vdash M_1 : T_1$, by lemma A.5, we can conclude that $\Theta; \Gamma \vdash M_1[\overline{q} := \overline{p}] : T_1[\overline{q} := \overline{p}]$ and prove this case.
(7) Case *commute*. We need to prove this rule case by case according to the eight cases shown in Figure 8. For each case, we need to firstly apply the inversion of a typing rule to obtain the

typing information of all subterms, and reconstruct the typing information of rewritten by the commute rule according the typing rules which are relevant to that specific case.

□

## A.2 Basic Properties of Projection

Lemma A.7. *Let $M$ be a choreography.*

*(1) If $M$ is a value, then* $\text{pn}(\text{type}(M)) = \text{pn}(M)$.
*(2) If $M$ is not a value, then* $\text{pn}(\text{type}(M)) \subseteq \text{pn}(M)$.

Proof. Proceed by induction on the typing derivations of $M$.

(1) Let $M = \text{select}_{q,r}\ l\ M'$.
  1. By the induction hypothesis, $\text{pn}(\text{type}(M')) \subseteq \text{pn}(M')$.
  2. By the definition of typing rules, $\text{type}(M) = \text{type}(M')$.
  3. Hence $\text{pn}(\text{type}(M)) \subseteq \text{pn}(M') \cup \{q, r\} = \text{pn}(M)$.

(2) Let $M = M_1\ M_2$.
  1. By the induction hypothesis, $\text{pn}(\text{type}(M_1)) \subseteq \text{pn}(M_1)$ and $\text{pn}(\text{type}(M_2)) \subseteq \text{pn}(M_2)$.
  2. By the definition of typing rules, $\text{pn}(\text{type}(M)) \subseteq \text{pn}(\text{type}(M_1)) \cup \text{pn}(\text{type}(M_2))$.
  3. Hence $\text{pn}(\text{type}(M)) \subseteq \text{pn}(M_1) \cup \text{pn}(M_2) = \text{pn}(M)$.

(3) Let $M = \text{if } M' \text{ then } M_1 \text{ else } M_2$.
  1. By the induction hypothesis, $\text{pn}(\text{type}(M')) \subseteq \text{pn}(M')$ and $\text{pn}(\text{type}(M_1)) \subseteq \text{pn}(M_1)$ and $\text{pn}(\text{type}(M_2)) \subseteq \text{pn}(M_2)$.
  2. By the definition of typing rules, $\text{pn}(\text{type}(M)) \subseteq \text{pn}(\text{type}(M_1)) \cup \text{pn}(\text{type}(M_2))$.
  3. Hence $\text{pn}(\text{type}(M)) \subseteq \text{pn}(M_1) \cup \text{pn}(M_2) \subseteq \text{pn}(M') \cup \text{pn}(M_1) \cup \text{pn}(M_2) = \text{pn}(M)$.

(4) In all other cases, the result follows by definition of $\text{pn}(-)$.

□

Lemma 4.2. *Let $M$ be a choreography where* $p \notin \text{pn}(M)$. *Then* $[\![M]\!]_p = \bot$.

Proof. Proceed by induction on the typing derivations of $M$.

(1) Let $M = x$.
  1. Assume $p \notin \text{pn}(M)$.
     a. By Lemma A.7 $p \notin \text{pn}(M)$ implies $p \notin \text{pn}(\text{type}(x))$.
     b. Therefore $[\![x]\!]_p = \bot$.
  2. Assume $[\![M]\!]_p = \bot$.
     a. By definition of projection, $p \notin \text{pn}(\text{type}(x))$.
     b. By Lemma A.7, $p \notin \text{pn}(M)$.

(2) Let $M = \text{select}_{q,r}\ l\ M'$.
  1. Assume $p \notin \text{pn}(M)$.
     a. Then $p \notin \text{pn}(M)$ implies $p \notin \{q, r\} \cup \text{pn}(M')$.
     b. By the induction hypothesis, $[\![M']\!]_p = \bot$.
     c. Hence $[\![M]\!]_p = [\![M']\!]_p = \bot$.
  2. Assume $[\![M]\!]_p = \bot$.
     a. By definition of projection, $p \notin \{q, r\}$.
     b. By the induction hypothesis, $p \notin \text{pn}(M')$.
     c. Hence $p \notin \text{pn}(M)$.

(3) Let $M = \lambda x : T.M'$.
  1. Assume $p \notin \text{pn}(M)$.
     a. By definition of $\text{pn}(-)$, we have $p \notin \text{pn}(T) \cup \text{pn}(M)$.

b. By definition of projection $[\![M]\!]_{\mathsf{p}} = \bot$.

2. Assume $[\![M']\!]_{\mathsf{p}} = \bot$.

a. By definition of projection, $\mathsf{p} \notin \mathrm{pn}(T) \cup \mathrm{pn}(M')$.

b. By definition of $\mathrm{pn}(-)$, we have $\mathsf{p} \notin \mathrm{pn}(M)$.

(4) Let $M = M_1\ M_2$.

1. Assume $\mathsf{p} \notin \mathrm{pn}(M)$.

a. Then $\mathsf{p} \notin \mathrm{pn}(M)$ implies $\mathsf{p} \notin \mathrm{pn}(M_1)$ and $\mathsf{p} \notin \mathrm{pn}(M_2)$.

b. By definition of projection, $[\![M]\!]_{\mathsf{p}} = \bot$.

2. Assume $[\![M]\!]_{\mathsf{p}} = \bot$.

a. By definition of projection, $\mathsf{p} \notin \mathrm{pn}(M_1) \cup \mathrm{pn}(M_2)$.

b. Hence $\mathsf{p} \notin \mathrm{pn}(M)$.

(5) Let $M = \mathsf{if}\ M'\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$.

1. Assume $\mathsf{p} \notin \mathrm{pn}(M)$.

a. Then $\mathsf{p} \notin \mathrm{pn}(M)$ implies $\mathsf{p} \notin \mathrm{pn}(M') \cup \mathrm{pn}(M_1) \cup \mathrm{pn}(M_2)$.

b. By Lemma A.7, $\mathsf{p} \notin \mathrm{pn}(\mathrm{type}(M'))$.

c. By the induction hypothesis, $[\![M_1]\!]_{\mathsf{p}} = [\![M_2]\!]_{\mathsf{p}} = \bot$.

d. By definition of projection, $[\![M]\!]_{\mathsf{p}} = [\![M_1]\!]_{\mathsf{p}} \sqcup [\![M_2]\!]_{\mathsf{p}} = \bot \sqcup \bot = \bot$.

2. Assume $[\![M]\!]_{\mathsf{p}} = \bot$.

a. By definition of projection and $(\sqcup)$, $\mathsf{p} \notin \mathrm{pn}(M')$ and $[\![M_1]\!]_{\mathsf{p}} = [\![M_2]\!]_{\mathsf{p}} = \bot$.

b. By the induction hypothesis, $\mathsf{p} \notin \mathrm{pn}(M_1) \cup \mathrm{pn}(M_2)$.

c. Hence $\mathsf{p} \notin \mathrm{pn}(M)$.

(6) All other cases follow by definition of projection.

□

Lemma 4.3 (Modularity). *A context $C$ is a choreography with a unique hole $\bullet$ in place of some subexpression. Let $C$ be a context and $M_1, M_2$ choreographies such that $\mathsf{p} \notin \mathrm{pn}(M_1)$ and $\mathsf{p} \notin \mathrm{pn}(M_2)$. Then $[\![C[M_1]]\!]_{\mathsf{p}} = [\![C[M_2]]\!]_{\mathsf{p}}$.*

Proof. By routine induction on the structure of contexts $C$; projection treats $M_1$ and $M_2$ the same way.

(1) $C = \bullet$. Follows from Lemma 4.2.

(2) $C = C'\ M$. If $\mathsf{p} \in \mathrm{pn}(M)$ then by the induction hypothesis $[\![C[M_1]]\!]_{\mathsf{p}} = [\![C'[M_1]]\!]_{\mathsf{p}}\ [\![M]\!]_{\mathsf{p}} = [\![C'[M_2]]\!]_{\mathsf{p}}\ [\![M]\!]_{\mathsf{p}} = [\![C[M_2]]\!]_{\mathsf{p}}$. Otherwise $[\![C[M_1]]\!]_{\mathsf{p}} = [\![C[M_2]]\!]_{\mathsf{p}} = \bot$.

(3) $C = M\ C'$. If $\mathsf{p} \in \mathrm{pn}(M)$ then by the induction hypothesis $[\![C[M_1]]\!]_{\mathsf{p}} = [\![M]\!]_{\mathsf{p}}\ [\![C'[M_1]]\!]_{\mathsf{p}} = [\![M]\!]_{\mathsf{p}}\ [\![C'[M_2]]\!]_{\mathsf{p}} = [\![C[M_2]]\!]_{\mathsf{p}}$. Otherwise $[\![C[M_1]]\!]_{\mathsf{p}} = [\![C[M_2]]\!]_{\mathsf{p}} = \bot$.

(4) $C = \mathsf{if}\ C'\ \mathsf{then}\ M_3\ \mathsf{else}\ M_4$.

1. Assume $\mathsf{p} \in \mathrm{type}(C'[M_1])$. Then $\mathsf{p} \in \mathrm{type}(C'[M_2])$. By the induction hypothesis, $[\![\mathsf{if}\ C'[M_1]\ \mathsf{then}\ M_3\ \mathsf{else}\ M_4]\!]_{\mathsf{p}} = \mathsf{if}\ [\![C'[M_1]]\!]_{\mathsf{p}}\ \mathsf{then}\ [\![M_3]\!]_{\mathsf{p}}\ \mathsf{else}\ [\![M_4]\!]_{\mathsf{p}} = \mathsf{if}\ [\![C'[M_2]]\!]_{\mathsf{p}}\ \mathsf{then}\ [\![M_3]\!]_{\mathsf{p}}\ \mathsf{else}\ [\![M_4]\!]_{\mathsf{p}} = [\![\mathsf{if}\ C'[M_2]\ \mathsf{then}\ M_3\ \mathsf{else}\ M_4]\!]_{\mathsf{p}}$.

2. Assume $\mathsf{p} \in \mathrm{pn}(C'[M_1]) \setminus \mathrm{pn}(\mathrm{type}(C'[M_1]))$. Then $\mathsf{p} \in \mathrm{pn}(C'[M_2]) \setminus \mathrm{pn}(\mathrm{type}(C'[M_2]))$. By induction hypothesis, $[\![\mathsf{if}\ C'[M_1]\ \mathsf{then}\ M_3\ \mathsf{else}\ M_4]\!]_{\mathsf{p}} = (\lambda x : \bot. [\![M_3]\!]_{\mathsf{p}} \sqcup [\![M_4]\!]_{\mathsf{p}})\ [\![C'[M_1]]\!]_{\mathsf{p}} = (\lambda x : \bot. [\![M_3]\!]_{\mathsf{p}} \sqcup [\![M_4]\!]_{\mathsf{p}})\ [\![C'[M_2]]\!]_{\mathsf{p}} = [\![\mathsf{if}\ C'[M_2]\ \mathsf{then}\ M_3\ \mathsf{else}\ M_4]\!]_{\mathsf{p}}$.

3. Otherwise, $\mathsf{p} \notin C'$. Then $\mathsf{p} \notin \mathrm{pn}(C'[M_1])$ and $\mathsf{p} \notin \mathrm{pn}(C'[M_2])$. Hence $[\![C[M_1]]\!]_{\mathsf{p}} = [\![M_3]\!]_{\mathsf{p}} \sqcup [\![M_4]\!]_{\mathsf{p}} = [\![C[M_2]]\!]_{\mathsf{p}}$.

(5) $C = \mathsf{select}_{\mathsf{q},\mathsf{r}}\ l\ C'$.

1. Assume $\mathsf{p} = \mathsf{q}$. By the induction hypothesis, $[\![C[M_1]]\!]_{\mathsf{p}} = \oplus_{\mathsf{r}}\ l\ [\![C'[M_1]]\!]_{\mathsf{p}} = \oplus_{\mathsf{r}}\ l\ [\![C'[M_2]]\!]_{\mathsf{p}} = [\![C[M_2]]\!]_{\mathsf{p}}$.

2. Assume $\mathsf{p} = \mathsf{r}$. By the induction hypothesis,

$$[\![C[M_1]]\!]_{\mathsf{p}} = \&_{\mathsf{r}}\{l : [\![C'[M_1]]\!]_{\mathsf{p}}\} = \&_{\mathsf{r}}\{l : [\![C'[M_2]]\!]_{\mathsf{p}}\} = [\![C[M_2]]\!]_{\mathsf{p}}.$$

3. Assume $\mathsf{p} \notin \{\mathsf{q}, \mathsf{r}\}$. By the induction hypothesis,

$$[\![C[M_1]]\!]_{\mathsf{p}} = [\![C'[M_1]]\!]_{\mathsf{p}} = [\![C'[M_2]]\!]_{\mathsf{p}} = [\![C[M_2]]\!]_{\mathsf{p}}.$$

(6) $C = \lambda x : T.C'$. By the induction hypothesis,

$$[\![C[M_1]]\!]_{\mathsf{p}} = \lambda x : [\![T]\!]_{\mathsf{p}}.[\![C'[M_1]]\!]_{\mathsf{p}} = \lambda x : [\![T]\!]_{\mathsf{p}}.[\![C'[M_2]]\!]_{\mathsf{p}} = [\![C[M_2]]\!]_{\mathsf{p}}.$$

□

Lemma 4.1. *If $V$ is a choreographic value then $[\![V]\!]_{\mathsf{p}}$ is a network-level value.*

Proof. Immediate by induction on the structure of $V$. □

Lemma A.8. *Let $\Gamma; \Theta \vdash M : T$ and $\mathsf{p} \in \Theta$. Then $x \in \mathrm{fv}([\![M]\!]_{\mathsf{p}})$ if and only if $x \in \mathrm{fv}(M)$ and $\mathsf{p} \in \mathrm{pn}(x)$.*

Proof. By structural induction on $M$.

(1) $M = x$. Then $[\![M]\!]_{\mathsf{p}} = x$ if $\mathsf{p} \in \mathrm{pn}(x)$ and $[\![M]\!]_{\mathsf{p}} = \bot$ otherwise.

(2) $M = \lambda x : T.M'$.

1. By the induction hypothesis, $y \in \mathrm{fv}([\![M']\!]_{\mathsf{p}})$ if and only if $y \in \mathrm{fv}(M')$ and $\mathsf{p} \in \mathrm{pn}(y)$.
2. By definition of projection, $[\![M]\!]_{\mathsf{p}} = \lambda x : [\![T]\!]_{\mathsf{p}}.[\![M']\!]_{\mathsf{p}}$. Hence

$$y \in \mathrm{fv}([\![M]\!]_{\mathsf{p}}) \Leftrightarrow y \in \mathrm{fv}([\![M']\!]_{\mathsf{p}}) \Leftrightarrow y \in \mathrm{fv}(M') \land \mathsf{p} \in \mathrm{pn}(y) \Leftrightarrow y \in \mathrm{fv}(M) \land \mathsf{p} \in \mathrm{pn}(y).$$

(3) $M = \mathsf{let}\ x : T = M_1\ \mathsf{in}\ M_2$.

1. By the induction hypothesis, $y \in \mathrm{fv}([\![M_2]\!]_{\mathsf{p}})$ if and only if $y \in \mathrm{fv}(M_2)$ and $\mathsf{p} \in \mathrm{pn}(y)$.
2. Assume $\mathsf{p} \in \mathrm{pn}(M_1)$. By definition of projection, $[\![M]\!]_{\mathsf{p}} = (\lambda x : [\![T]\!]_{\mathsf{p}}.[\![M_2]\!]_{\mathsf{p}})\ [\![M_1]\!]_{\mathsf{p}}$. Hence

$$y \in \mathrm{fv}([\![M]\!]_{\mathsf{p}}) \Leftrightarrow y \in \mathrm{fv}([\![M_2]\!]_{\mathsf{p}}) \Leftrightarrow y \in \mathrm{fv}(M_2) \land \mathsf{p} \in \mathrm{pn}(y) \Leftrightarrow y \in \mathrm{fv}(M) \land \mathsf{p} \in \mathrm{pn}(y).$$

3. Otherwise, $\mathsf{p} \notin \mathrm{pn}(M_1)$.
   a. Then $\mathsf{p} \notin \mathrm{pn}(T)$ by Lemma A.7.
   b. By definition of projection, $[\![M]\!]_{\mathsf{p}} = [\![M_2]\!]_{\mathsf{p}}$.
   c. Hence $y \in \mathrm{fv}([\![M]\!]_{\mathsf{p}})$ if and only if $y \in \mathrm{fv}(M)$ and $\mathsf{p} \in \mathrm{pn}(y)$.

(4) In all other cases, the result follows by routine induction.

□

Lemma A.9. *Let $\Gamma; \Theta \vdash \mathsf{let}\ x : T = V\ \mathsf{in}\ M : T'$. For any $\mathsf{p}$, $[\![M[x := V]]\!]_{\mathsf{p}} = [\![M]\!]_{\mathsf{p}}[x := [\![V]\!]_{\mathsf{p}}]$.*

Proof. By cases.

(1) Assume $\mathsf{p} \notin \mathrm{pn}(V)$.

1. Then $[\![M[x := V]]\!]_{\mathsf{p}} = [\![M]\!]_{\mathsf{p}}$ by Lemma 4.3.
2. By Lemma A.7, $\mathsf{p} \notin \mathrm{pn}(T)$.
3. By Lemma A.8, $x \notin \mathrm{fv}([\![M]\!]_{\mathsf{p}})$.
4. Hence $[\![M]\!]_{\mathsf{p}}[x := [\![V]\!]_{\mathsf{p}}] = [\![M]\!]_{\mathsf{p}}$.

(2) Otherwise, $\mathsf{p} \in \mathrm{pn}(V)$.

1. By Lemma A.7, $\mathsf{p} \in \mathrm{pn}(T)$. Hence $\mathsf{p} \in \mathrm{pn}(x)$.
2. The rest of the proof follows by routine induction on the structure of $M$.

□

Lemma 4.4. *Let $\mathcal{A}$ be an answering context.*

*(1) If* $\mathsf{r} \notin \mathrm{pn}(\mathcal{A})$ *then* $[\![\mathcal{A}[M]]\!]_\mathsf{r} = [\![M]\!]_\mathsf{r}$ *for any* $M$.

*(2) If* $\mathsf{r} \in \mathrm{pn}(\mathcal{A})$ *then there exists* $\mathcal{B}$ *such that, for any* $M$, $[\![\mathcal{A}[M]]\!]_\mathsf{r} = \mathcal{B}[\,[\![M]\!]_\mathsf{r}\,]$.

Proof. By case analysis on $\mathcal{A}$.

(1) Assume $\mathcal{A} = \mathsf{let}\ x : T = M'\ \mathsf{in}\ \bullet$.
1. Assume $\mathsf{r} \notin \mathrm{pn}(\mathcal{A})$. By definition of $\mathrm{pn}(-)$, $\mathsf{r} \notin \mathrm{pn}(M')$. By definition of projection, $[\![\mathcal{A}[M]]\!]_\mathsf{r} = [\![M]\!]_\mathsf{r}$.
2. Assume $\mathsf{r} \in \mathrm{pn}(\mathcal{A})$. By definition of $\mathrm{pn}(-)$, $\mathsf{r} \in \mathrm{pn}(M')$. By definition of projection, $[\![\mathcal{A}[M]]\!]_\mathsf{r} = \mathcal{B}[\,[\![M]\!]_\mathsf{r}\,]$ where $\mathcal{B} = (\lambda x : [\![T]\!]_\mathsf{r}.\bullet)\ [\![M']\!]_\mathsf{r}$.

(2) Assume $\mathcal{A} = \mathsf{select}_{\mathsf{p},\mathsf{q}}\ l\ \bullet$.
1. Assume $\mathsf{r} \notin \mathrm{pn}(\mathcal{A})$. By definition of $\mathrm{pn}(-)$, $\mathsf{r} \notin \{\mathsf{p},\mathsf{q}\}$. By definition of projection, $[\![\mathcal{A}[M]]\!]_\mathsf{r} = [\![M]\!]_\mathsf{r}$.
2. Assume $\mathsf{r} \in \mathrm{pn}(\mathcal{A})$. By definition of $\mathrm{pn}(-)$, $\mathsf{r} = \mathsf{p}$ or $\mathsf{r} = \mathsf{q}$.
   a. If $\mathsf{r} = \mathsf{p}$ then $[\![\mathcal{A}[M]]\!]_\mathsf{r} = \mathcal{B}[\,[\![M]\!]_\mathsf{r}\,]$ where $\mathcal{B} = \oplus_\mathsf{q}\ l\ \bullet$.
   b. If $\mathsf{r} = \mathsf{q}$ then $[\![\mathcal{A}[M]]\!]_\mathsf{r} = \mathcal{B}[\,[\![M]\!]_\mathsf{r}\,]$ where $\mathcal{B} = \&_\mathsf{p}\{l : \bullet\}$.

□

Lemma 4.5. *Let* $M = \mathcal{F}[N]$ *where* $\Theta;\Gamma \vdash N : T$. *Then there exists a network-level frame* $\mathcal{F}$ *such that, for any* $N'$ *where* $\Theta;\Gamma \vdash N' : T$,

*(1) If* $\mathsf{r} \in \mathrm{pn}(N')$ *then* $\mathcal{F}[\,[\![N']\!]_\mathsf{r}\,] = [\![\mathcal{F}[N']]\!]_\mathsf{r}$.

*(2) If* $\mathsf{r} \notin \mathrm{pn}(N')$ *then* $\mathcal{F}[\,[\![N']\!]_\mathsf{r}\,] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![\mathcal{F}[N']]\!]_\mathsf{r}$.

Proof. By case analysis on $\mathcal{F}$.

(1) Assume $\mathcal{F} = \bullet\ M'$. We show that $\mathcal{F} = \bullet\ [\![M']\!]_\mathsf{r}$.
1. Assume $\mathsf{r} \in \mathrm{pn}(N')$. By definition of projection, $[\![\mathcal{F}[N']]\!]_\mathsf{r} = [\![N']\!]_\mathsf{r}\ [\![M']\!]_\mathsf{r}$. Hence $[\![\mathcal{F}[N']]\!]_\mathsf{r} = \mathcal{F}[\,[\![N']\!]_\mathsf{r}\,]$.
2. Otherwise, $\mathsf{r} \notin \mathrm{pn}(N')$. By Lemma 4.2, $[\![N']\!]_\mathsf{r} = \bot$.
   a. Assume $\mathsf{r} \in \mathrm{pn}(M')$. By definition of projection, $[\![\mathcal{F}[N']]\!]_\mathsf{r} = \bot\ [\![M']\!]_\mathsf{r}$. Hence $\mathcal{F}[\bot] = [\![\mathcal{F}[N']]\!]_\mathsf{r}$.
   b. Otherwise, $\mathsf{r} \notin \mathrm{pn}(M')$. By definition of projection, $[\![\mathcal{F}[N']]\!]_\mathsf{r} = \bot$. Notice $\mathcal{F}[\bot] = \bot\ \bot$. Hence $\mathcal{F}[\bot] \rightarrow [\![\mathcal{F}[N']]\!]_\mathsf{r}$.

(2) Assume $\mathcal{F} = V\ \bullet$. We show that $\mathcal{F} = [\![V]\!]_\mathsf{r}\ \bullet$.
1. Assume $\mathsf{r} \in \mathrm{pn}(N')$. By definition of projection, $[\![\mathcal{F}[N']]\!]_\mathsf{r} = [\![V]\!]_\mathsf{r}\ [\![N']\!]_\mathsf{r}$. Hence $[\![\mathcal{F}[N']]\!]_\mathsf{r} = \mathcal{F}[\,[\![N']\!]_\mathsf{r}\,]$.
2. Otherwise, $\mathsf{r} \notin \mathrm{pn}(N')$. By Lemma 4.2, $[\![N']\!]_\mathsf{r} = \bot$.
   a. Assume $\mathsf{r} \in \mathrm{pn}(V)$. By definition of projection, $[\![\mathcal{F}[N']]\!]_\mathsf{r} = [\![V]\!]_\mathsf{r}\ \bot$. Hence $\mathcal{F}[\bot] = [\![\mathcal{F}[N']]\!]_\mathsf{r}$.
   b. Otherwise, $\mathsf{r} \notin \mathrm{pn}(V)$. By definition of projection, $[\![\mathcal{F}[N']]\!]_\mathsf{r} = \bot$ Notice $\mathcal{F}[\bot] = \bot\ \bot$. Hence $\mathcal{F}[\bot] \rightarrow [\![\mathcal{F}[N']]\!]_\mathsf{r}$.

(3) Assume $\mathcal{F} = \mathsf{let}\ x : T = \bullet\ \mathsf{in}\ M'$. We show that $\mathcal{F} = (\lambda x : [\![T]\!]_\mathsf{r}.[\![M']\!]_\mathsf{r})\ \bullet$.
1. Assume $\mathsf{r} \in \mathrm{pn}(N')$. By definition of projection, $[\![\mathcal{F}[N']]\!]_\mathsf{r} = (\lambda x : [\![T]\!]_\mathsf{r}.[\![M']\!]_\mathsf{r})\ [\![N']\!]_\mathsf{r}$. Hence $[\![\mathcal{F}[N']]\!]_\mathsf{r} = \mathcal{F}[\,[\![N']\!]_\mathsf{r}\,]$.
2. Otherwise, $\mathsf{r} \notin \mathrm{pn}(N')$. By definition of projection, $[\![\mathcal{F}[N']]\!]_\mathsf{r} = [\![M']\!]_\mathsf{r}$. Hence $\mathcal{F}[\bot] = (\lambda x : [\![T]\!]_\mathsf{r}.[\![M']\!]_\mathsf{r})\ \bot \rightarrow [\![M']\!]_\mathsf{r} = [\![\mathcal{F}[N']]\!]_\mathsf{r}$.

(4) Assume $\mathcal{F} = \mathsf{if}\ \bullet\ \mathsf{then}\ M_1\ \mathsf{else}\ M_2$ and $\mathsf{r} \in \mathrm{pn}(T)$. By definition of projection, $[\![\mathcal{F}[N']]\!]_\mathsf{r} = \mathsf{if}\ [\![N']\!]_\mathsf{r}\ \mathsf{then}\ [\![M_1]\!]_\mathsf{r}\ \mathsf{else}\ [\![M_2]\!]_\mathsf{r}$. We show $\mathcal{F} = \mathsf{if}\ \bullet\ \mathsf{then}\ [\![M_1]\!]_\mathsf{r}\ \mathsf{else}\ [\![M_2]\!]_\mathsf{r}$. By Lemma A.7, $\mathsf{r} \in \mathrm{pn}(\mathrm{type}(N'))$ is only satisfied when $\mathsf{r} \in \mathrm{pn}(N')$. By definition, $[\![\mathcal{F}[N']]\!]_\mathsf{r} = \mathcal{F}[\,[\![N']\!]_\mathsf{r}\,]$.

(5) Assume $\mathcal{F}$ = if $\bullet$ then $M_1$ else $M_2$ and $\mathsf{r} \notin \mathrm{pn}(T)$. We show $\mathcal{F} = (\lambda x : \bot.[\![M_1]\!]_\mathsf{r} \sqcup [\![M_2]\!]_\mathsf{r})\ \bullet$.

1. Assume $\mathsf{r} \in \mathrm{pn}(N')$. By definition of projection, $[\![\mathcal{F}[N']]\!]_\mathsf{r} = (\lambda x : \bot.[\![M_1]\!]_\mathsf{r} \sqcup [\![M_2]\!]_\mathsf{r})\ [\![N']\!]_\mathsf{r}$. Hence $[\![\mathcal{F}[N']]\!]_\mathsf{r} = \mathcal{F}[\ [\![N']\!]_\mathsf{r}\ ]$.
2. Otherwise, $\mathsf{r} \notin \mathrm{pn}(N')$. By definition of projection, $[\![\mathcal{F}[N']]\!]_\mathsf{r} = [\![M_1]\!]_\mathsf{r} \sqcup [\![M_2]\!]_\mathsf{r}$. Notice $\mathcal{F}[\bot] = (\lambda x : \bot.[\![M_1]\!]_\mathsf{r} \sqcup [\![M_2]\!]_\mathsf{r})\ \bot$. Hence $\mathcal{F}[\bot] \to [\![\mathcal{F}[N']]\!]_\mathsf{r}$.

□

Lemma 4.6. *Let $\Delta$ be a redex at* r *that is not an* (app) *or* (commute) *redex. Then $[\![\Delta]\!]_\mathsf{r} = \delta$ for some network-level redex $\delta$.*

Proof. By case analysis on choreographic redexes at r.

(1) Assume $\Delta = \mathsf{com}_{\mathsf{p},\mathsf{q}}\ M$.

1. Assume $\mathsf{r} = \mathsf{p}$ and $M = c@\mathsf{p}$. By definition of projection, $[\![\Delta]\!]_\mathsf{r} = \mathsf{send}_\mathsf{q}\ c$, which is a redex for *(send)*.
2. Otherwise, $\mathsf{r} = \mathsf{q}$. By definition of projection and Lemma 4.2, $[\![\Delta]\!]_\mathsf{r} = \mathsf{recv}_\mathsf{p}\ \bot$, which is a redex for *(receive)*.

(2) Assume $\Delta = \mathsf{select}_{\mathsf{p},\mathsf{q}}\ l\ M'$ where $\mathsf{r} \in \{\mathsf{p}, \mathsf{q}\}$.

1. Assume $\mathsf{r} = \mathsf{p}$. By definition of projection, $[\![\Delta]\!]_\mathsf{r} = \oplus_\mathsf{q}\ l\ [\![M']\!]_\mathsf{r}$, which is a redex for *(choice)*.
2. Otherwise, $\mathsf{r} = \mathsf{q}$. By definition of projection, $[\![\Delta]\!]_\mathsf{r} = \&_\mathsf{p}\{l : [\![M']\!]_\mathsf{r}\}$, which is a redex for *(offer)*.

(3) Assume $\Delta = \mathsf{let}\ x : T = V\ \mathsf{in}\ M$ where $\mathsf{r} \in \mathrm{pn}(V)$.

1. By definition of projection, $[\![\Delta]\!]_\mathsf{r} = (\lambda x : [\![T]\!]_\mathsf{r}.[\![M]\!]_\mathsf{r})\ [\![V]\!]_\mathsf{r}$. By Lemma 4.1, $[\![V]\!]_\mathsf{r}$ is a value $L$. Hence $[\![\Delta]\!]_\mathsf{r}$ is a redex for *(p-app)*.

(4) Assume $\Delta = \mathsf{if}\ M\ \mathsf{then}\ M_{\mathsf{true}}\ \mathsf{else}\ M_{\mathsf{false}}$ where $\mathsf{r} \in \mathrm{pn}(\Delta)$.

1. Assume $\mathsf{p} \in \mathrm{pn}(M)$. Then $M = c@\mathsf{p}$ for some $c$. By definition of projection, $[\![\Delta]\!]_\mathsf{r} = \mathsf{if}\ c\ \mathsf{then}\ [\![M_{\mathsf{true}}]\!]_\mathsf{r}\ \mathsf{else}\ [\![M_{\mathsf{false}}]\!]_\mathsf{r}$, which is a redex for *(p-if)*.
2. Otherwise, $\mathsf{r} \neq \mathsf{p}$. By definition of projection, $[\![\Delta]\!]_\mathsf{r} = [\![M_1]\!]_\mathsf{r} \sqcup [\![M_2]\!]_\mathsf{r}$. We assumed $\mathsf{r} \in \mathrm{pn}(\Delta)$, so $\mathsf{r} \in \mathrm{pn}(M_1) \cup \mathrm{pn}(M_2)$. By Lemma 4.2, $[\![M_1]\!]_\mathsf{r} \neq \bot$ or $[\![M_2]\!]_\mathsf{r} \neq \bot$. Hence, by definition of ($\sqcup$), $[\![M_1]\!]_\mathsf{r} \sqcup [\![M_2]\!]_\mathsf{r}$ has the form $\&_\mathsf{q}\{l_1 : R_1, \ldots, l_n : R_n\}$ for some $n$, which is a redex for *(offer)*.

(5) Assume $\Delta = f(\overline{\mathsf{p}})$.

1. Then $\mathsf{r} \in \overline{\mathsf{p}}$ because we assumed $\mathsf{r} \in \mathrm{pn}(\Delta)$. By definition of projection, $[\![f(\overline{\mathsf{p}})]\!]_\mathsf{r} = f_i(\overline{\mathsf{q}})$ for some $\overline{\mathsf{q}} \subseteq \overline{\mathsf{p}}$ and some $i$.

□

Lemma 4.7. *Let $M = \mathcal{E}_\mathsf{r}[N]$ where $\Theta; \Gamma \vdash N : T$. Then there exists a network-level evaluation context $\mathcal{E}$ such that, for any $N'$ where $\Theta; \Gamma \vdash N' : T$,*

*(1) If $\mathsf{r} \in \mathrm{pn}(N')$ then $\mathcal{E}[\ [\![N']\!]_\mathsf{r}\ ] = [\![\mathcal{E}_\mathsf{r}[N']]\!]_\mathsf{r}$.*

*(2) If $\mathsf{r} \notin \mathrm{pn}(N')$ then $\mathcal{E}[\ [\![N']\!]_\mathsf{r}\ ] \overset{\tau}{\twoheadrightarrow} [\![\mathcal{E}_\mathsf{r}[N']]\!]_\mathsf{r}$.*

Proof. Let $M = \mathcal{E}_\mathsf{r}[N]$ where $\Theta; \Gamma \vdash N : T$. Proceed by induction on the structure of $\mathcal{E}_\mathsf{r}$.

(1) Assume $\mathcal{E}_\mathsf{r} = \bullet$. We show $\mathcal{E} = \bullet$. Notice that $[\![\mathcal{E}_\mathsf{r}[N']]\!]_\mathsf{r} = [\![N']\!]_\mathsf{r} = \mathcal{E}[\ [\![N']\!]_\mathsf{r}\ ]$ and $\mathcal{E}[\bot] = \bot$.

1. Assume $\mathsf{r} \in \mathrm{pn}(N')$. Then $[\![\mathcal{E}_\mathsf{r}[N']]\!]_\mathsf{r} = \mathcal{E}[\ [\![N']\!]_\mathsf{r}\ ]$.
2. Assume $\mathsf{r} \notin \mathrm{pn}(N')$. By Lemma 4.2, $[\![N']\!]_\mathsf{r} = \bot$. Hence $\mathcal{E}[\bot] = [\![\mathcal{E}_\mathsf{r}[N']]\!]_\mathsf{r}$.

(2) Otherwise, $\mathcal{E}_\mathsf{r} = \mathcal{F}[\mathcal{E}'_\mathsf{r}]$.

1. In the type derivation of $M$, let $T'$ be the type of $\mathcal{E}'_\mathsf{r}[N]$.
2. By the induction hypothesis, there exists $\mathcal{E}'$ such that, for any $N'$ of type $T$, (1) if $\mathsf{r} \in \mathrm{pn}(N')$ then $[\![\mathcal{E}'_\mathsf{r}[N']]\!]_\mathsf{r} = \mathcal{E}'[\ [\![N']\!]_\mathsf{r}\ ]$; and (2) if $\mathsf{r} \notin \mathrm{pn}(N')$ then $\mathcal{E}'[\bot] \overset{\tau}{\twoheadrightarrow} [\![\mathcal{E}'_\mathsf{r}[N']]\!]_\mathsf{r}$.

3. By Lemma 4.5, there exists $\mathcal{E}''$ such that, for any $N''$ of type $T'$, (1) if $\mathsf{r} \in \mathrm{pn}(N'')$ then $[\![\mathcal{F}[N'']]\!]_{\mathsf{r}} = \mathcal{E}''[\,[\![N'']\!]_{\mathsf{r}}\,]$; and (2) if $\mathsf{r} \notin \mathrm{pn}(N'')$ then $\mathcal{E}''[\bot] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![\mathcal{F}[N'']]\!]_{\mathsf{r}}$.
4. We show $\mathcal{E} = \mathcal{E}''[\mathcal{E}']$.
5. Assume $\mathsf{r} \in \mathrm{pn}(N')$.
   a. Then $\mathsf{r} \in \mathrm{pn}(\mathcal{E}'_{\mathsf{r}}[N])$.
   b. Letting $N'' := \mathcal{E}'_{\mathsf{r}}[N']$, we have $[\![\mathcal{F}[\mathcal{E}'_{\mathsf{r}}[N']]]\!]_{\mathsf{r}} = \mathcal{E}''[[\![\mathcal{E}'_{\mathsf{r}}[N']]\!]_{\mathsf{r}}\,] = \mathcal{E}''[\mathcal{E}'[\,[\![N']\!]_{\mathsf{r}}\,]]$.
   c. Hence $[\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{r}} = \mathcal{E}[\,[\![N']\!]_{\mathsf{r}}\,]$.
6. Assume $\mathsf{r} \notin \mathrm{pn}(N')$.
   a. Assume $\mathsf{r} \in \mathrm{pn}(\mathcal{E}'_{\mathsf{r}})$. Then $\mathsf{r} \in \mathrm{pn}(\mathcal{E}'_{\mathsf{r}}[N])$. By instantiating $N'' := \mathcal{E}'_{\mathsf{r}}[N']$, we have $[\![\mathcal{F}[\mathcal{E}'_{\mathsf{r}}[N']]]\!]_{\mathsf{r}} = \mathcal{E}''[\,[\![\mathcal{E}'_{\mathsf{r}}[N']]\!]_{\mathsf{r}}\,]$. Since $\mathcal{E}'[\bot] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![\mathcal{E}'_{\mathsf{r}}[N']]\!]_{\mathsf{r}}$, we also have $\mathcal{E}''[\,\mathcal{E}'[\bot]\,] \xrightarrow{\tau}\!\!\!\!\rightarrow \mathcal{E}''[\,[\![\mathcal{E}'_{\mathsf{r}}[N']]\!]_{\mathsf{r}}\,]$. Hence $\mathcal{E}[\bot] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![\mathcal{F}[\mathcal{E}'_{\mathsf{r}}[N']]]\!]_{\mathsf{r}}$.
   b. Otherwise, $\mathsf{r} \notin \mathrm{pn}(\mathcal{E}'_{\mathsf{r}})$. By Lemma 4.2, $[\![\mathcal{E}'_{\mathsf{r}}[N']]\!]_{\mathsf{r}} = \bot$. By instantiating $N'' := \mathcal{E}'_{\mathsf{r}}[N']$, we have $\mathcal{E}''[\bot] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![\mathcal{F}[\mathcal{E}'_{\mathsf{r}}[N']]]\!]_{\mathsf{r}}$. Since $\mathcal{E}'[\bot] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![\mathcal{E}'_{\mathsf{r}}[N']]\!]_{\mathsf{r}}$, we have $\mathcal{E}'[\bot] \xrightarrow{\tau}\!\!\!\!\rightarrow \bot$. Hence $\mathcal{E}''[\mathcal{E}'[\bot]] \xrightarrow{\tau}\!\!\!\!\rightarrow \mathcal{E}''[\bot]$. Hence $\mathcal{E}[\bot] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![\mathcal{F}[\mathcal{E}'_{\mathsf{r}}[N']]]\!]_{\mathsf{r}}$.

(3) Otherwise, $\mathcal{E}_{\mathsf{r}} = \mathcal{A}[\mathcal{E}'_{\mathsf{r}}]$ where $\mathsf{r} \notin \mathrm{pn}(\mathcal{A})$.
1. By Lemma 4.4, $[\![\mathcal{A}[\mathcal{E}'_{\mathsf{r}}[N]]]\!]_{\mathsf{r}} = [\![\mathcal{E}'_{\mathsf{r}}[N]]\!]_{\mathsf{r}}$.
2. By the induction hypothesis for $\mathcal{E}'_{\mathsf{r}}[N]$, there exists $\mathcal{E}'$ such that for any $N'$ of type $T$, (1) $\mathsf{r} \in \mathrm{pn}(N')$ implies $[\![\mathcal{E}'_{\mathsf{r}}[N]]\!]_{\mathsf{r}} = \mathcal{E}'[\,[\![N]\!]_{\mathsf{r}}\,]$ and (2) $\mathsf{r} \notin \mathrm{pn}(N')$ implies $\mathcal{E}'[\bot] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![\mathcal{E}'_{\mathsf{r}}[N]]\!]_{\mathsf{r}}$.
3. We show $\mathcal{E} = \mathcal{E}'$.
4. Assume $\mathsf{r} \in \mathrm{pn}(N')$. Then $[\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{r}} = [\![\mathcal{E}'_{\mathsf{r}}[N]]\!]_{\mathsf{r}} = \mathcal{E}'[\,[\![N]\!]_{\mathsf{r}}\,]$. Hence $[\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{r}} = \mathcal{E}[\,[\![N]\!]_{\mathsf{r}}\,]$.
5. Otherwise, $\mathsf{r} \notin \mathrm{pn}(N')$. Then $\mathcal{E}[\bot] = \mathcal{E}'[\bot] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![\mathcal{E}'_{\mathsf{r}}[N']]\!]_{\mathsf{r}} = [\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{r}}$. Hence $\mathcal{E}[\bot] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{r}}$.

□

Lemma A.10. *Let* p, r *be roles such that* $\mathsf{p} \neq \mathsf{r}$. *Let* $M = \mathcal{E}_{\mathsf{r}}[N]$ *where* $\Theta; \Gamma \vdash N : T$. *Then there exists a degenerate context* $\mathcal{O}$ *such that, for any* $N'$ *where* $\Theta; \Gamma \vdash N' : T$,

*(1) If* $\mathsf{p} \in \mathrm{pn}(N')$ *then* $\mathcal{O}[\,[\![N']\!]_{\mathsf{p}}\,] = [\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{p}}$.
*(2) If* $\mathsf{p} \notin \mathrm{pn}(N')$ *then* $\mathcal{O}[\,[\![N']\!]_{\mathsf{p}}\,] \rightsquigarrow\!\!\!\!\rightarrow [\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{p}}$.

Proof. Let $M = \mathcal{E}_{\mathsf{r}}[N]$ where $N$ is given type $T$ by the judgment $\Theta; \Gamma \vdash N : T$. Proceed by induction on the structure of $\mathcal{E}_{\mathsf{r}}$.

(1) Assume $\mathcal{E}_{\mathsf{r}} = \bullet$. We show $\mathcal{O} = \bullet$.
1. Notice that $[\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{p}} = [\![N']\!]_{\mathsf{p}} = \mathcal{O}[\,[\![N']\!]_{\mathsf{p}}\,]$ and $\mathcal{O}[\bot] = \bot$.
2. Assume $\mathsf{r} \in \mathrm{pn}(N')$. Then $[\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{r}} = \mathcal{O}[\,[\![N']\!]_{\mathsf{r}}\,]$.
3. Assume $\mathsf{r} \notin \mathrm{pn}(N')$. By Lemma 4.2, $[\![N']\!]_{\mathsf{r}} = \bot$. Hence $\mathcal{O}[\bot] = [\![\mathcal{E}_{\mathsf{r}}[N']]\!]_{\mathsf{r}}$.

(2) Otherwise, $\mathcal{E}_{\mathsf{r}} = \mathcal{F}[\mathcal{E}'_{\mathsf{r}}]$.
1. In the type derivation of $M$, let $T'$ be the type of $\mathcal{E}'_{\mathsf{r}}[N]$.
2. By the induction hypothesis, there exists $\mathcal{O}'$ such that, for any $N'$ of type $T$, (1) if $\mathsf{p} \in \mathrm{pn}(N')$ then $[\![\mathcal{E}'_{\mathsf{r}}[N']]\!]_{\mathsf{p}} = \mathcal{O}'[\,[\![N']\!]_{\mathsf{p}}\,]$; and (2) if $\mathsf{p} \notin \mathrm{pn}(N')$ then $\mathcal{O}'[\bot] \rightsquigarrow\!\!\!\!\rightarrow [\![\mathcal{E}'_{\mathsf{r}}[N']]\!]_{\mathsf{p}}$.
3. By Lemma 4.5, there exists $\mathcal{E}''$ such that, for any $N''$ of type $T'$, (1) if $\mathsf{p} \in \mathrm{pn}(N'')$ then $[\![\mathcal{F}[N'']]\!]_{\mathsf{p}} = \mathcal{E}''[\,[\![N'']\!]_{\mathsf{p}}\,]$; and (2) if $\mathsf{p} \notin \mathrm{pn}(N'')$ then $\mathcal{E}''[\bot] \xrightarrow{\tau}\!\!\!\!\rightarrow [\![\mathcal{F}[N'']]\!]_{\mathsf{p}}$.
4. We show $\mathcal{O} = \mathcal{E}''[\mathcal{O}']$.
5. Assume $\mathsf{p} \in \mathrm{pn}(N')$.
   a. Then $\mathsf{p} \in \mathrm{pn}(\mathcal{E}'_{\mathsf{r}}[N])$.
   b. Letting $N'' := \mathcal{E}'_{\mathsf{r}}[N']$, we have $[\![\mathcal{F}[\mathcal{E}'_{\mathsf{r}}[N']]]\!]_{\mathsf{p}} = \mathcal{E}''[[\![\mathcal{E}'_{\mathsf{r}}[N']]\!]_{\mathsf{p}}\,] = \mathcal{E}''[\mathcal{O}'[\,[\![N']\!]_{\mathsf{p}}\,]]$.

c. Hence $[\![\mathcal{E}_r[N']]\!]_p = \mathcal{E}[\,[\![N']\!]_p\,]$.
6. Assume $p \notin pn(N')$.
a. Assume $p \in pn(\mathcal{E}'_r)$.
(1) Then $p \in pn(\mathcal{E}'_r[N])$.
(2) By instantiating $N'' := \mathcal{E}'_r[N']$, we have $[\![\mathcal{F}[\mathcal{E}'_r[N']]]\!]_p = \mathcal{E}''[\,[\![\mathcal{E}'_r[N']]\!]_p\,]$.
(3) Since $O'[\bot] \rightsquigarrow\!\!\!\!\!\rightarrow [\![\mathcal{E}'_r[N']]\!]_p$, we also have $\mathcal{E}''[\,O'[\bot]\,] \rightsquigarrow\!\!\!\!\!\rightarrow \mathcal{E}''[\,[\![\mathcal{E}'_r[N']]\!]_p\,]$.
(4) Hence $O[\bot] \rightsquigarrow\!\!\!\!\!\rightarrow [\![\mathcal{F}[\mathcal{E}'_r[N']]]\!]_p$.
b. Otherwise, $p \notin pn(\mathcal{E}'_r)$.
(1) By Lemma 4.2, $[\![\mathcal{E}'_r[N']]\!]_p = \bot$.
(2) By instantiating $N'' := \mathcal{E}'_r[N']$, we have $\mathcal{E}''[\bot] \overset{\tau}{\twoheadrightarrow} [\![\mathcal{F}[\mathcal{E}'_r[N']]]\!]_p$.
(3) Since $O'[\bot] \rightsquigarrow\!\!\!\!\!\rightarrow [\![\mathcal{E}'_r[N']]\!]_p$, we have $O'[\bot] \rightsquigarrow\!\!\!\!\!\rightarrow \bot$.
(4) Hence $\mathcal{E}''[O'[\bot]] \rightsquigarrow\!\!\!\!\!\rightarrow \mathcal{E}''[\bot]$.
(5) Hence $\mathcal{E}[\bot] \rightsquigarrow\!\!\!\!\!\rightarrow [\![\mathcal{F}[\mathcal{E}'_r[N']]]\!]_r$.

(3) Otherwise, $\mathcal{E}_r = \mathcal{A}[\mathcal{E}'_r]$ where $p \in pn(\mathcal{A})$.
1. By Lemma 4.4, $[\![\mathcal{A}[\mathcal{E}'_r[N]]]\!]_p = \mathcal{B}[\,[\![\mathcal{E}'_r[N]]\!]_p\,]$ for some $\mathcal{B}$.
2. By the induction hypothesis for $\mathcal{E}'_r[N]$, there exists $O'$ such that for any $N'$ of type $T$, (1) $p \in pn(N')$ implies $[\![\mathcal{E}'_r[N]]\!]_p = O'[\,[\![N]\!]_p\,]$ and (2) $p \notin pn(N')$ implies $O'[\bot] \rightsquigarrow\!\!\!\!\!\rightarrow [\![\mathcal{E}'_r[N]]\!]_p$.
3. We show $O = \mathcal{B}[O']$.
4. Assume $p \in pn(N')$. Then $[\![\mathcal{E}_r[N']]\!]_p = \mathcal{B}[\,[\![\mathcal{E}'_r[N]]\!]_p\,] = \mathcal{B}[O'[\,[\![N]\!]_p\,]]$. Hence $[\![\mathcal{E}_r[N']]\!]_p = O[\,[\![N]\!]_p\,]$.
5. Otherwise, $p \notin pn(N')$. Then $O[\bot] = \mathcal{B}[O'[\bot]] \rightsquigarrow\!\!\!\!\!\rightarrow \mathcal{B}[\,[\![\mathcal{E}'_r[N']]\!]_p\,] = [\![\mathcal{E}_r[N']]\!]_p$. Hence $O[\bot] \rightsquigarrow\!\!\!\!\!\rightarrow [\![\mathcal{E}_r[N']]\!]_p$.

□

Lemma 4.8. *Let $(\rightsquigarrow\!\!\!\!\!\rightarrow)$ be the reflexive transitive closure of $(\rightsquigarrow)$. If $M \overset{\tau}{\twoheadrightarrow} M'$ only by (commute) and (app) reductions, then $[\![M]\!] \rightsquigarrow\!\!\!\!\!\rightarrow [\![M']\!]$.*

Proof. First we show that for any r, $[\![\mathcal{F}[\mathcal{A}[M]]]\!]_r \rightsquigarrow\!\!\!\!\!\rightarrow [\![\mathcal{A}[\mathcal{F}[M]]]\!]_r$.

(1) Assume $r \in pn(M)$.
1. Assume $r \in pn(\mathcal{A})$.
a. By lemma 4.4, there exists $\mathcal{B}$ such that $[\![\mathcal{A}[N]]\!]_r = \mathcal{B}[\,[\![N]\!]_r\,]$ for any $N$.
b. By Lemma 4.5, there exists $\mathcal{F}$ such that both (1) $[\![\mathcal{F}[\mathcal{A}[M]]]\!]_r = \mathcal{F}[\mathcal{B}[[\![M]\!]_r]]$ and (2) $[\![\mathcal{A}[\mathcal{F}[M]]]\!]_r = \mathcal{B}[\mathcal{F}[[\![M]\!]_r]]$.
c. Hence $[\![M]\!]_r \rightsquigarrow [\![M']\!]_r$.
2. Assume $r \notin pn(\mathcal{A})$.
a. By Lemma 4.4, $[\![\mathcal{A}[N]]\!]_r = [\![N]\!]_r$ for any $N$.
b. Hence $[\![\mathcal{A}[\mathcal{F}[M]]]\!]_r = [\![\mathcal{F}[M]]\!]_r$.
c. By Lemma 4.5, there exists $\mathcal{F}$ such that both (1) $[\![\mathcal{F}[\mathcal{A}[M]]]\!]_r = \mathcal{F}[[\![M]\!]_r]$ and (2) $[\![\mathcal{A}[\mathcal{F}[M]]]\!]_r = \mathcal{F}[[\![M]\!]_r]$.
d. Hence $[\![\mathcal{F}[\mathcal{A}[M]]]\!]_r = \mathcal{F}[[\![M]\!]_r] = [\![\mathcal{A}[\mathcal{F}[M]]]\!]_r$.

(2) Assume $r \notin pn(M)$.
1. By Lemma 4.2, $[\![M]\!]_r = \bot$.
2. Assume $r \in pn(\mathcal{A})$.
a. By Lemma 4.4 there exists $\mathcal{B}$ such that $[\![\mathcal{A}[N]]\!]_r = \mathcal{B}[\,[\![N]\!]_r\,]$ for any $N$.
b. By Lemma 4.5, there exists $\mathcal{F}$ such that both (1) $[\![\mathcal{F}[\mathcal{A}[M]]]\!]_r = \mathcal{F}[\mathcal{B}[\bot]]$, and (2) $\mathcal{F}[\bot] \overset{\tau}{\twoheadrightarrow} [\![\mathcal{F}[M]]\!]_r$.

c. Hence $[\![\mathcal{F}[\mathcal{A}[M]]]\!]_\mathsf{r} = \mathcal{F}[\mathcal{B}[\bot]] \rightsquigarrow \mathcal{B}[\mathcal{F}[\bot]] \rightsquigarrow \mathcal{B}[\,[\![\mathcal{F}[M]]\!]_\mathsf{r}\,] = [\![\mathcal{A}[\mathcal{F}[M]]]\!]_\mathsf{r}$.

3. Assume $\mathsf{r} \notin \mathrm{pn}(\mathcal{A})$.
   a. By Lemma 4.4, $[\![\mathcal{A}[N]]\!]_\mathsf{r} = [\![N]\!]_\mathsf{r}$ for any $N$.
   b. Hence $[\![\mathcal{A}[\mathcal{F}[M]]]\!]_\mathsf{r} = [\![\mathcal{F}[M]]\!]_\mathsf{r}$.
   c. Observe $\mathsf{r} \notin \mathrm{pn}(M)$ and $\mathsf{r} \notin \mathrm{pn}(\mathcal{A}[M])$.
   d. By Lemma 4.3, $[\![\mathcal{F}[M]]\!]_\mathsf{r} = [\![\mathcal{F}[\mathcal{A}[M]]]\!]_\mathsf{r}$.
   e. Hence $[\![\mathcal{F}[\mathcal{A}[M]]]\!]_\mathsf{r} = [\![\mathcal{F}[M]]\!]_\mathsf{r} = [\![\mathcal{A}[\mathcal{F}[M]]]\!]_\mathsf{r}$.

Next, we show that for any r, $[\![(\lambda x : T.M)\ M']\!]_\mathsf{r} \rightsquigarrow\!\!\!\!\!\rightarrow [\![\mathsf{let}\ x : T = M'\ \mathsf{in}\ M]\!]_\mathsf{r}$.

(1) Assume $\mathsf{r} \in \mathrm{pn}(M')$.
1. Assume $\mathsf{r} \in \mathrm{pn}(\lambda x : T.M)$.
   a. By definition of projection, $[\![(\lambda x : T.M)\ M']\!]_\mathsf{r} = (\lambda x : [\![T]\!]_\mathsf{r}.[\![M]\!]_\mathsf{r})\ [\![M']\!]_\mathsf{r}$.
   b. By definition of projection, $[\![\mathsf{let}\ x : T = M'\ \mathsf{in}\ M]\!]_\mathsf{r} = (\lambda x : [\![T]\!]_\mathsf{r}.[\![M]\!]_\mathsf{r})\ [\![M']\!]_\mathsf{r}$.
   c. Hence $[\![(\lambda x : T.M)\ M']\!]_\mathsf{r} = [\![\mathsf{let}\ x : T = M'\ \mathsf{in}\ M]\!]_\mathsf{r}$.
2. Otherwise, $\mathsf{r} \notin \mathrm{pn}(\lambda x : T.M)$.
   a. By Lemma 4.2, $[\![\lambda x : T.M]\!]_\mathsf{r} = \bot$ and $[\![M]\!]_\mathsf{r} = \bot$.
   b. By definition of projection, $[\![(\lambda x : T.M)\ M']\!]_\mathsf{r} = \bot\ [\![M']\!]_\mathsf{r}$.
   c. By definition of projection, $[\![\mathsf{let}\ x : T = M'\ \mathsf{in}\ M]\!]_\mathsf{r} = (\lambda x : \bot.\ \bot)\ [\![M']\!]_\mathsf{r}$.
   d. Hence $[\![(\lambda x : T.M)\ M']\!]_\mathsf{r} \rightsquigarrow [\![\mathsf{let}\ x : T = M'\ \mathsf{in}\ M]\!]_\mathsf{r}$ by *(bottom$_2$)*.

(2) Assume $\mathsf{r} \notin \mathrm{pn}(M')$.
1. By Lemma 4.2, $[\![M']\!]_\mathsf{r} = \bot$.
2. Assume $\mathsf{r} \in \mathrm{pn}(M)$.
   a. By definition of projection, $[\![(\lambda x : T.M)\ M']\!]_\mathsf{r} = (\lambda x : [\![T]\!]_\mathsf{r}.[\![M]\!]_\mathsf{r})\ \bot$.
   b. By definition of projection, $[\![\mathsf{let}\ x : T = M'\ \mathsf{in}\ M]\!]_\mathsf{r} = [\![M]\!]_\mathsf{r}$.
   c. Hence $[\![(\lambda x : T.M)\ M']\!]_\mathsf{r} \rightarrow [\![\mathsf{let}\ x : T = M'\ \mathsf{in}\ M]\!]_\mathsf{r}$.
3. Otherwise, $\mathsf{r} \notin \mathrm{pn}(M)$.
   a. By Lemma 4.2, $[\![M]\!]_\mathsf{r} = \bot$.
   b. By definition of projection, $[\![(\lambda x : T.M)\ M']\!]_\mathsf{r} = \bot$.
   c. By definition of projection, $[\![\mathsf{let}\ x : T = M'\ \mathsf{in}\ M]\!]_\mathsf{r} = [\![M]\!]_\mathsf{r} = \bot$.
   d. Hence $[\![(\lambda x : T.M)\ M']\!]_\mathsf{r} = [\![\mathsf{let}\ x : T = M'\ \mathsf{in}\ M]\!]_\mathsf{r}$.

□

Lemma A.11. *If* $(\lambda x : T.\ P)\ L \rightsquigarrow (\lambda x : T.\ P')\ L$ *then* $P[x := L] \rightsquigarrow P'[x := L]$.

Proof. Assume $(\lambda x : T.\ P)\ L \rightsquigarrow (\lambda x : T.\ P')\ L$. Then $\mathcal{O}[\Delta] = (\lambda x : T.\ P)\ L$ for some $\mathcal{O}, \Delta$. This is only possible if $\mathcal{O} = (\lambda x : T.\ \mathcal{O}')\ L$ for some $\mathcal{O}'$. Notice that for every $\mathcal{F}$ and $\mathcal{B}$, the substitutions $\mathcal{F}[x := L]$ and $\mathcal{B}[x := L]$ are valid frames and degenerate contexts, respectively. Proceeding by induction on $\mathcal{O}'$, we prove that $(\lambda x : T.\ \mathcal{O}'[\Delta])\ L \rightsquigarrow (\lambda x : T.\ \mathcal{O}'[\Delta'])\ L$ implies $(\mathcal{O}'[\Delta])[x := L] \rightsquigarrow (\mathcal{O}'[\Delta'])[x := L]$.

(1) Assume $\mathcal{O}' = \bullet$. By case analysis on the transition $(\lambda x : T.\ P)\ L \rightsquigarrow (\lambda x : T.\ P')\ L$.
1. Case 1: *(prune)*. Then $\Delta = \&_\mathsf{q}\{l_i : P_i\}_{i \in \mathcal{I}}$ and $\Delta' = \&_\mathsf{q}\{l_j : P_j\}_{j \in \mathcal{J}}$ where $\mathcal{J} \subseteq \mathcal{I}$. Therefore $P[x := L] = \&_\mathsf{q}\{l_i : P_i[x := L]\}_{i \in \mathcal{I}}$ and $P'[x := L] = \&_\mathsf{q}\{l_j : P_j[x := L]\}_{j \in \mathcal{J}}$ and also $P[x := L] \rightsquigarrow P'[x := L]$.
2. Case 2: *(commute)*. Then $\Delta = \mathcal{F}[\mathcal{B}[R]]$ for some $\mathcal{F}, \mathcal{B}, R$. Let $\mathcal{F}' = \mathcal{F}[x := L]$ and let $\mathcal{B}' = \mathcal{B}[x := L]$. Then $P[x := L] = \mathcal{F}'[\mathcal{B}'[R[x := L]]]$ and $P'[x := L] = \mathcal{B}'[\mathcal{F}'[R[x := L]]]$. Hence $P[x := L] \rightsquigarrow P'[x := L]$.
3. Case 3: *(bottom$_2$)*. Then $\Delta = \bot\ Q$ and $\Delta' = (\lambda x : \bot.\ \bot)\ Q$. Therefore $P[x := L] = \bot\ Q[x := L]$ and $P'[x := L] = (\lambda x : \bot.\ \bot)\ Q[x := L]$. Hence $P[x := L] \rightsquigarrow P'[x := L]$ by *(bottom$_2$)*.
4. Case 4: *(compute)*. Proceed by case analysis on the redex $\Delta$.

a. Assume $\Delta = (\lambda x : S.\ R)\ L'$. Then $P[x := L] = (\lambda x : S.\ R)\ (L'[x := L])$ and $P'[x := L] = R[x := L'][x := L]$. Then $R[x := L'][x := L] = R[x := (L'[x := L])]$ implies $P[x := L] \rightsquigarrow P'[x := L]$.
b. Assume $\Delta = (\lambda y : S.\ R)\ L'$ where $x \neq y$ and $y \notin \text{fv}(L)$. Then it must follow that $P[x := L] = (\lambda y : S.\ R[x := L])\ (L'[x := L])$ and $P'[x := L] = R[y := L'][x := L]$. Since $R[y := L'][x := L] = R[x := L][y := L'[x := L]]$, we have $P[x := L] \rightsquigarrow P'[x := L]$.
c. Assume $\Delta = \text{if } c \text{ then } R_1 \text{ else } R_2$, where $c = \text{true}$ without loss of generality. Then $P[x := L] = \text{if true then } R_1[x := L] \text{ else } R_2[x := L]$ and $P'[x := L] = R_1[x := L]$. Hence $P[x := L] \rightsquigarrow P'[x := L]$.
d. Assume $\Delta = f(\overline{\mathsf{p}})$. Then $\Delta'$ is some closed term $R$. Hence $P[x := L] = \Delta$ and $P'[x := L] = \Delta'$, so $P[x := L] \rightsquigarrow P'[x := L]$.
e. Assume $\Delta = \bot\ \bot$. Then $P[x := L] = \Delta$ and $P'[x := L] = \Delta'$, so $P[x := L] \rightsquigarrow P'[x := L]$.

(2) Assume $\mathcal{O}' = \mathcal{F}[\mathcal{O}'']$ for some $\mathcal{F}$. Let $\mathcal{F}' = \mathcal{F}[x := L]$. Then $P[x := L] = \mathcal{F}'[\mathcal{O}''[\Delta]]$ and $P'[x := L] = \mathcal{F}'[\mathcal{O}''[\Delta']]$ so $P[x := L] \rightsquigarrow P'[x := L]$ by the induction hypothesis.
(3) Assume $\mathcal{O}' = \mathcal{B}[\mathcal{O}'']$ for some $\mathcal{B}$. As above.

□

Lemma A.12. *Let $P \rightsquigarrow \tilde{P}$.*

*(1) If $P = \mathcal{E}[\&_{\mathsf{q}}\{l_i : P_i\}_{i \in \mathcal{I}}]$ then $\tilde{P} = \tilde{\mathcal{E}}[\&_{\mathsf{q}}\{l_j : \tilde{P}_j\}_{j \in \mathcal{J}}]$ for some $\tilde{\mathcal{E}}$, $\mathcal{J} \subseteq \mathcal{I}$, and $\{\tilde{P}_j\}_{j \in \mathcal{J}}$.*
*(2) If $P = \mathcal{E}[\oplus_{\mathsf{q}}\ l\ P']$ then $\tilde{P} = \tilde{\mathcal{E}}[\oplus_{\mathsf{q}}\ l\ \tilde{P}']$ for some $\tilde{\mathcal{E}}, \tilde{P}'$.*

Proof. Assume $P \rightsquigarrow \tilde{P}$. Then $P = \mathcal{O}[\Delta]$ and $\tilde{P} = \mathcal{O}[\Delta']$ for some $\mathcal{O}, \Delta, \Delta'$. Proceed by induction on $\mathcal{O}$.

(1) Assume $\mathcal{O} = \bullet$.
1. Assume $P \rightsquigarrow \tilde{P}$ by *(prune)*. Then $\mathcal{E} = \bullet$ and $\tilde{P} = \tilde{\mathcal{E}}[\&_{\mathsf{q}}\{l_j : P_j\}_{j \in \mathcal{J}}]$ where $\tilde{\mathcal{E}} = \bullet$ and $\mathcal{J} \subseteq \mathcal{I}$.
2. Assume $P \rightsquigarrow \tilde{P}$ by *(compute)*. Then $P$ is neither $\mathcal{E}[\&_{\mathsf{q}}\{l_i : P_i\}_{i \in \mathcal{I}}]$ nor $\mathcal{E}[\oplus_{\mathsf{q}}\ l\ P']$.
3. Assume $P \rightsquigarrow \tilde{P}$ by *(bottom$_2$)*. Then $\mathcal{E} = \bot\ \mathcal{E}'$ for some $\mathcal{E}'$. Hence for any $\delta$, if $P = \mathcal{E}[\delta]$ then $\tilde{P} = \tilde{\mathcal{E}}[\delta]$ where $\tilde{\mathcal{E}} = (\lambda x : \bot.\ \bot)\ \mathcal{E}'$.
4. Assume $P \rightsquigarrow \tilde{P}$ by *(commute)*. Then $P = \mathcal{F}[\mathcal{B}[R]]$ for some $\mathcal{F}, \mathcal{B}, R$. Proceed by case analysis on $\mathcal{B}$.
a. Assume $\mathcal{B} = \oplus_{\mathsf{r}}\ l'\ \bullet$. Then $\mathcal{E} = \mathcal{F}$ and $P = \mathcal{E}[\oplus_{\mathsf{q}}\ l\ P']$. Hence $\tilde{P} = \oplus_{\mathsf{q}}\ l\ \mathcal{F}[P']$.
b. Assume $\mathcal{B} = \&_{\mathsf{r}}\{l_k : \bullet\}$ for some $k$. Then $\mathcal{E} = \mathcal{F}$ and $P = \mathcal{E}[\&_{\mathsf{q}}\{l_i : P_i\}_{i \in \mathcal{I}}]$ where $\mathcal{I} = \{k\}$. Hence $\tilde{P} = \&_{\mathsf{q}}\{l_k : \mathcal{F}[P_k]\}$.
c. Assume $\mathcal{B} = (\lambda x : T.\ \bullet)\ R'$. Then $\mathcal{E} = \mathcal{F}[(\lambda x : T.\ R)\ \mathcal{E}']$ for some $\mathcal{E}'$. Hence for any $\delta$, if $P = \mathcal{E}[\delta]$ then $\tilde{P} = \tilde{\mathcal{E}}[\delta]$ where $\tilde{\mathcal{E}} = (\lambda x : T.\ \mathcal{F}[R])\ \mathcal{E}'$.

(2) Assume $\mathcal{O} = \mathcal{F}'[\mathcal{O}']$.
1. Assume $\mathcal{F}' = \bullet\ R$. Notice $\mathcal{O}'[\Delta]$ cannot be a value, so $\mathcal{E} = \mathcal{F}'[\mathcal{E}']$ for some $\mathcal{E}'$. The result then follows by induction.
2. Assume $\mathcal{F}' = L\ \bullet$ and $L$ is a value. Then $\mathcal{E} = \mathcal{F}'[\mathcal{E}']$ for some $\mathcal{E}'$ and the result follows by induction.
3. Assume $\mathcal{F}' = \text{if } \bullet \text{ then } R_1 \text{ else } R_2$. As above.

(3) Assume $\mathcal{O} = \mathcal{B}[\mathcal{O}']$.
1. Assume $\mathcal{B} = \oplus_{\mathsf{r}}\ l'\ \bullet$. Then $P = \mathcal{E}[\oplus_{\mathsf{q}}\ l\ P']$ where $\mathcal{E} = \bullet$. Hence $\tilde{P} = \oplus_{\mathsf{q}}\ l\ \mathcal{O}'[\Delta']$.
2. Assume $\mathcal{B} = \&_{\mathsf{r}}\{l_k : \bullet\}$ for some $k$. Then $P = \mathcal{E}[\&_{\mathsf{q}}\{l_i : P_i\}_{i \in \mathcal{I}}]$ where $\mathcal{E} = \bullet$ and $\mathcal{I} = \{k\}$. Hence $\tilde{P} = \&_{\mathsf{q}}\{l_k : \mathcal{O}'[\Delta']\}$.

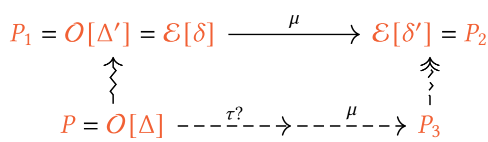
3. Assume $\mathcal{B} = (\lambda x : T.\ \bullet)\ R$. Then $\mathcal{E} = (\lambda x : T.\ \mathcal{O}'[\Delta])\ \mathcal{E}'$ for some $\mathcal{E}'$. Hence for any $\delta$, if $P = \mathcal{E}[\delta]$ then $\tilde{P} = \tilde{\mathcal{E}}[\delta]$ where $\tilde{\mathcal{E}} = (\lambda x : T.\ \mathcal{O}'[\Delta'])\ \mathcal{E}'$.

□

Theorem 4.10 (Prophecy Theorem). *Let $M$ be a choreography, $\mathcal{N}$ a network, and $\mathcal{N} \rightsquigarrow\!\!\twoheadrightarrow [\![M]\!]$.*

- *(Completeness) If $[\![M]\!] \xrightarrow{\mu} \tilde{\mathcal{N}}'$ then there exists $\mathcal{N}'$ such that $\mathcal{N} \overset{\tau}{\twoheadrightarrow}\xrightarrow{\mu} \mathcal{N}' \rightsquigarrow\!\!\twoheadrightarrow \tilde{\mathcal{N}}'$.*
- *(Soundness) If $\mathcal{N} \xrightarrow{\mu} \mathcal{N}'$ where $\mu \neq \tau$ then there exists $\tilde{\mathcal{N}}'$ such that $[\![M]\!] \xrightarrow{\mu} \tilde{\mathcal{N}}'$ and $\mathcal{N}' \rightsquigarrow\!\!\twoheadrightarrow \tilde{\mathcal{N}}'$. If $\mathcal{N} \xrightarrow{\tau} \mathcal{N}'$ then either $\mathcal{N}' \rightsquigarrow\!\!\twoheadrightarrow [\![M]\!]$ or there exists $\tilde{\mathcal{N}}'$ such that $[\![M]\!] \xrightarrow{\tau} \tilde{\mathcal{N}}'$ and $\mathcal{N}' \rightsquigarrow\!\!\twoheadrightarrow \tilde{\mathcal{N}}'$.*

Proof (Completeness). Assuming $P \rightsquigarrow P_1$ and $P_1 \xrightarrow{\mu} P_2$, we find $P_3$ such that the following diagram commutes:




We proceed by induction on the structure of $\mathcal{O}$.

(1) Assume $\mathcal{O} = \bullet$.

1. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(compute)* rule.
   a. Notice $\mathcal{O}$ is an evaluation context and $\Delta$ is a redex.
   b. Hence $P \xrightarrow{\tau} P_1 \xrightarrow{\mu} P_3$ where $P_3 = P_2$.
2. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(bottom$_2$)* rule.
   a. Since $\mathcal{O} = \bullet$, this implies $P = \bot\ R$ and $P_1 = (\lambda x : \bot.\ \bot)\ R$.
   b. If $R = \bot$ then $\mu = \tau$, $P_2 = \bot$, and $P \xrightarrow{\mu} P_3 \rightsquigarrow\!\!\twoheadrightarrow P_2$ where $P_3 = P_3$. Otherwise, $\mathcal{E} = (\lambda x : \bot.\ \bot)\ \mathcal{E}'$ for some $\mathcal{E}'$ where $\mathcal{E}'[\delta] \xrightarrow{\mu} \mathcal{E}'[\delta']$. Hence $P \xrightarrow{\mu} P_3 \rightsquigarrow\!\!\twoheadrightarrow P_2$ where $P_3 = \bot\ \mathcal{E}'[\delta']$ and $P_2 = (\lambda x : \bot.\ \bot)\ \mathcal{E}'[\delta']$.
3. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(prune)* rule.
   a. Since $\mathcal{O} = \bullet$, this implies $P = \&_{\mathrm{q}}\{l_i : R_i\}_{i \in \mathcal{I}}$ and $P_1 = \&_{\mathrm{q}}\{l_j : R_j\}_{j \in \mathcal{J}}$ for some q, $\mathcal{I}, \mathcal{J}$, $\{l_i\}_{i \in \mathcal{I}}, \{R_i\}_{i \in \mathcal{I}}$, where $\mathcal{J} \subseteq \mathcal{I}$.
   b. In order to have $P_1 = \mathcal{E}[\delta] \xrightarrow{\mu} \mathcal{E}[\delta']$, we must have $\mu = \mathrm{select}_{\mathrm{q,p}}\ l_{j^*}$ for some $j^* \in \mathcal{J}$ and $\mathcal{E} = \bullet$ and $\delta' = R_{j^*}$.
   c. Since $\mathcal{O}$ is an evaluation context and $j^* \in \mathcal{J} \subseteq \mathcal{I}$, we have $P \xrightarrow{\mu} P_3 = P_2$ where $P_3 = R_{j^*}$.
4. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(commute)* rule. Then $P = \mathcal{F}[\mathcal{B}[R]]$ for some $\mathcal{F}, \mathcal{B}, R$. We proceed by case analysis on $\mathcal{B}$.
   a. Assume $\mathcal{B} = \oplus_{\mathrm{q}}\ l\ \bullet$.
      (1) Then $P_1 = \oplus_{\mathrm{q}}\ l\ \mathcal{F}[R]$. Since $P_1 \xrightarrow{\mu} P_2$, we must have $\mu = \mathrm{select}_{\mathrm{p,q}}\ l$ and $P_2 = \mathcal{F}[R]$.
      (2) Notice $\mathcal{F}$ is an evaluation context. Hence $P \xrightarrow{\mu} P_3$ where $P_3 = P_2$.
   b. Assume $\mathcal{B} = \&_{\mathrm{q}}\{l : \bullet\}$.
      (1) Then $P_1 = \&_{\mathrm{q}}\{l : \mathcal{F}[R]\}$. Since $P_1 \xrightarrow{\mu} P_2$, we must have $\mu = \mathrm{select}_{\mathrm{q,p}}\ l$ and $P_2 = \mathcal{F}[R]$.
      (2) Again, $\mathcal{F}$ is an evaluation context. Hence $P \xrightarrow{\mu} P_3$ where $P_3 = P_2$.
   c. Assume $\mathcal{B} = (\lambda x : T.\ \bullet)\ L$, where $L$ is a value.
      (1) Then $P_1 = (\lambda x : T.\ \mathcal{F}[R])\ L$. Since $P_1 \xrightarrow{\mu} P_2$, we must have $P_2 = \mathcal{F}[R][x := L]$ and $\mu = \tau$.
      (2) Observe $P = \mathcal{F}[(\lambda x : T.\ R)\ L]$ and $P \xrightarrow{\mu} P_3$ where $P_3 = \mathcal{F}[R[x := L]]$.
      (3) Since $P$ is closed, $\mathcal{F}$ has no free variables. Hence $P_3 = P_2$.
   d. Assume $\mathcal{B} = (\lambda x : T.\ \bullet)\ R'$, where $R'$ is not a value.

(1) Then $P_1 = (\lambda x : T.\, \mathcal{F}[R])\ R'$ and $P_2 = (\lambda x : T.\, \mathcal{F}[R])\ R''$ for some $R''$. Then $P \xrightarrow{\mu} P_3$ and $P_3 \rightsquigarrow P_2$ where $P_3 = \mathcal{F}[(\lambda x : T.\, R)\ R'']$.

(2) Assume $\mathcal{O} = \mathcal{F}'[\mathcal{O}']$ where $\mathcal{F}' = \bullet\ R$.

1. Assume $\mathcal{E} = \bullet$. Here it suffices to show $P \xrightarrow{\tau} P_1$, since $P \xrightarrow{\tau} P_1$ and the assumption $P_1 \xrightarrow{\mu} P_2$ implies $P \xrightarrow{\tau}\xrightarrow{\mu} P_3 \rightsquigarrow\!\!\!\!\rightarrow P_2$ where $P_3 = P_2$.
   a. Since $P_1 = \mathcal{O}'[\Delta']\ R$ is an application term, the redex $\delta$ must have the form $\mathsf{send}_\mathsf{q}\ c$, $\mathsf{recv}_\mathsf{q}\ \bot$, $(\lambda x : T.\, Q)\ L$, or $\bot\ \bot$. In all four cases, $\mathcal{O}'[\Delta']$ and $R$ are both values.
   b. Since $\mathcal{O}'[\Delta']$ is a value, $\mathcal{O}'[\Delta] \rightsquigarrow \mathcal{O}'[\Delta']$ cannot have proceeded by a *(prune)*, *(commute)*, or *(bottom$_2$)* step; it must have been a *(compute)* step. Since $\mathcal{O}'[\Delta] \rightsquigarrow \mathcal{O}'[\Delta']$ was a *(compute)* step resulting in a value, we must have $\mathcal{O}' = \bullet$ and $\Delta$ must be a redex that contracts into a value.
   c. Since $\mathcal{O}' = \bullet$, it follows that $\mathcal{F}'[\mathcal{O}']$ is an evaluation context. Hence $\mathcal{F}'[\mathcal{O}'[\Delta]] \xrightarrow{\tau} \mathcal{F}[\mathcal{O}'[\Delta']]$, i.e. $P \xrightarrow{\tau} P_1$.
2. Assume $\mathcal{E} = \mathcal{F}[\mathcal{E}']$ where $\mathcal{F} = \bullet\ Q$. Then the result follows by induction.
3. Assume $\mathcal{E} = \mathcal{F}[\mathcal{E}']$ where $\mathcal{F} = L\ \bullet$. Again, it suffices to show $P \xrightarrow{\tau} P_1$.
   a. Notice $\mathcal{O}'[\Delta'] = L$ is a value. Hence $\mathcal{O}'[\Delta] \rightsquigarrow \mathcal{O}'[\Delta']$ must have been a *(compute)* step where $\mathcal{O}' = \bullet$ and $\Delta$ is a redex that contracts into a value. Since $\mathcal{O}' = \bullet$, it follows that $\mathcal{F}'[\mathcal{O}']$ is an evaluation context. Hence $\mathcal{F}'[\mathcal{O}'[\Delta]] \xrightarrow{\tau} \mathcal{F}[\mathcal{O}'[\Delta']]$, i.e. $P \xrightarrow{\tau} P_1$.
4. Assume $\mathcal{E} = \mathcal{F}[\mathcal{E}']$ where $\mathcal{F} = \mathsf{if}\ \bullet\ \mathsf{then}\ Q_1\ \mathsf{else}\ Q_2$. Impossible, because we must have $\mathcal{F}[\mathcal{E}'[\delta]] = \mathcal{F}'[\mathcal{O}'[\Delta']]$.

(3) Assume $\mathcal{O} = \mathcal{F}'[\mathcal{O}']$ where $\mathcal{F}' = L\ \bullet$ where $L$ is a value.

1. Assume $\mathcal{E} = \bullet$. Again, it suffices to show $P \xrightarrow{\tau} P_1$.
   a. Since $P_1 = L\ \mathcal{O}'[\Delta']$ is an application term and $P_1 = \mathcal{E}[\delta]$, the definition of redexes $\delta$ implies $\mathcal{O}'[\Delta']$ is a value. Hence $\mathcal{O}'[\Delta] \rightsquigarrow \mathcal{O}'[\Delta']$ must have been a *(compute)* step where $\mathcal{O}' = \bullet$ and $\Delta$ is a redex that contracts into a value. Since $\mathcal{O}' = \bullet$, it follows that $\mathcal{F}'[\mathcal{O}']$ is an evaluation context. Hence $\mathcal{F}'[\mathcal{O}'[\Delta]] \xrightarrow{\tau} \mathcal{F}[\mathcal{O}'[\Delta']]$, i.e. $P \xrightarrow{\tau} P_1$.
2. Assume $\mathcal{E} = \mathcal{F}[\mathcal{E}']$ where $\mathcal{F} = \bullet\ Q$. Impossible, because $\mathcal{F}[\mathcal{E}'[\delta]] = \mathcal{F}'[\mathcal{O}'[\Delta]]$ implies $L = \mathcal{E}'[\delta]$; there is no $\mathcal{E}'$ such that $\mathcal{E}'[\delta]$ is a value.
3. Assume $\mathcal{E} = \mathcal{F}[\mathcal{E}']$ where $\mathcal{F} = L'\ \bullet$. Then the result follows by induction.
4. Assume $\mathcal{E} = \mathcal{F}[\mathcal{E}']$ where $\mathcal{F} = \mathsf{if}\ \bullet\ \mathsf{then}\ Q_1\ \mathsf{else}\ Q_2$. Impossible, because we must have $\mathcal{F}[\mathcal{E}'[\delta]] = \mathcal{F}'[\mathcal{O}'[\Delta']]$.

(4) Assume $\mathcal{O} = \mathcal{F}'[\mathcal{O}']$ where $\mathcal{F}' = \mathsf{if}\ \bullet\ \mathsf{then}\ R_1\ \mathsf{else}\ R_2$.

1. Assume $\mathcal{E} = \bullet$. Again, it suffices to show $P \xrightarrow{\tau} P_1$.
   a. Since $P_1 = \mathsf{if}\ \mathcal{O}'[\Delta']\ \mathsf{then}\ R_1\ \mathsf{else}\ R_2$, the step $\mathcal{E}[\delta] \xrightarrow{\tau} \mathcal{E}[\delta']$ can only be a *(p-if)* reduction. Hence $\mathcal{O}'[\Delta']$ must be a value. Hence $\mathcal{O}'[\Delta] \rightsquigarrow \mathcal{O}'[\Delta']$ must have been a *(compute)* step where $\mathcal{O}' = \bullet$ and $\Delta$ is a redex that contracts into a value. Since $\mathcal{O}' = \bullet$, it follows that $\mathcal{F}'[\mathcal{O}']$ is an evaluation context. Hence $\mathcal{F}'[\mathcal{O}'[\Delta]] \xrightarrow{\tau} \mathcal{F}[\mathcal{O}'[\Delta']]$, i.e. $P \xrightarrow{\tau} P_1$.
2. Assume $\mathcal{E} = \mathcal{F}[\mathcal{E}']$ where $\mathcal{F} = \bullet\ Q$. Impossible, because we must have $\mathcal{F}[\mathcal{E}'[\delta]] = \mathcal{F}'[\mathcal{O}'[\Delta']]$.
3. Assume $\mathcal{E} = \mathcal{F}[\mathcal{E}']$ where $\mathcal{F} = L\ \bullet$. Impossible, because we must have $\mathcal{F}[\mathcal{E}'[\delta]] = \mathcal{F}'[\mathcal{O}'[\Delta']]$.
4. Assume $\mathcal{E} = \mathcal{F}[\mathcal{E}']$ where $\mathcal{F} = \mathsf{if}\ \bullet\ \mathsf{then}\ Q_1\ \mathsf{else}\ Q_2$. Then the result follows by induction.

(5) Assume $\mathcal{O} = \mathcal{B}[\mathcal{O}']$ where $\mathcal{B} = \oplus_\mathsf{q}\ l\ \bullet$. Impossible, because there is no $\mathcal{E}$ such that $\mathcal{B}[\mathcal{O}'[\Delta']] = \mathcal{E}[\delta]$.

(6) Assume $\mathcal{O} = \mathcal{B}[\mathcal{O}']$ where $\mathcal{B} = \&_\mathsf{q}\{l : \bullet\}$. Impossible, because there is no $\mathcal{E}$ such that $\mathcal{B}[\mathcal{O}'[\Delta']] = \mathcal{E}[\delta]$.

(7) Assume $\mathcal{O} = \mathcal{B}[\mathcal{O}']$ where $\mathcal{B} = (\lambda x : T.\, \bullet)\ L$ and $L$ is a value.

1. Then $P \rightsquigarrow P_1 \xrightarrow{\mu} P_2$ where $P = (\lambda x : T.\, \mathcal{O}'[\Delta])\ L$ and $P_1 = (\lambda x : T.\, \mathcal{O}'[\Delta'])\ L$ and $P_2 = \mathcal{O}'[\Delta'][x := L]$ and $\mu = \tau$.
2. Notice $(\lambda x : T.\, \mathcal{O}'[\Delta])\ L \xrightarrow{\tau} \mathcal{O}'[\Delta][x := L]$.
3. It suffices to show that $\mathcal{O}'[\Delta][x := L] \rightsquigarrow \mathcal{O}'[\Delta'][x := L]$; this follows from Lemma A.11.

(8) Assume $\mathcal{O} = \mathcal{B}[\mathcal{O}']$ where $\mathcal{B} = (\lambda x : T.\, \bullet)\ R$ and $R$ is not a value.

1. Then $P = (\lambda x : T.\, \mathcal{O}'[\Delta])\ R$ and $P_1 = (\lambda x : T.\mathcal{O}'[\Delta'])\ R$ and $P_2 = (\lambda x : T.\mathcal{O}'[\Delta'])\ R'$ for some $R'$. Hence $P \xrightarrow{\mu} P_3$ and $P_3 \rightsquigarrow P_2$ where $P_3 = (\lambda x : T.\mathcal{O}'[\Delta])\ R'$.

□

Proof (Soundness). First we handle the special case. Assuming $\mathcal{N} \xrightarrow{\mathsf{select}_{\mathsf{p},\mathsf{q}}\ l} \mathcal{N}'$ and $\mathcal{N} \rightsquigarrow \tilde{\mathcal{N}}$ by *(prune)*, we find $\tilde{\mathcal{N}}'$ such that $\mathcal{N}' \xrightarrow{\mathsf{select}_{\mathsf{p},\mathsf{q}}\ l} \tilde{\mathcal{N}}'$ and $\mathcal{N}' \rightsquigarrow \tilde{\mathcal{N}}'$ by *(prune)*:

(1) The transition $\mathcal{N} \xrightarrow{\mathsf{select}_{\mathsf{p},\mathsf{q}}\ l} \mathcal{N}'$ implies $\mathcal{N}(\mathsf{p}) = \mathcal{E}[\oplus_{\mathsf{q}}\ l\ P]$ and $\mathcal{N}(\mathsf{q}) = \mathcal{E}'[\&_{\mathsf{p}}\{l_i : Q_i\}_{i \in \mathcal{I}}]$ for some $\mathcal{E}, \mathcal{E}', P, \mathcal{I}, \{Q_i\}_{i \in \mathcal{I}}$ where $l \in \{l_i\}_{i \in \mathcal{I}}$.

(2) The relation $\mathcal{N} \rightsquigarrow \tilde{\mathcal{N}}$ implies $\tilde{\mathcal{N}}(\mathsf{p}) = \mathcal{E}[\oplus_{\mathsf{q}}\ l\ P']$ and $\mathcal{N}(\mathsf{q}) = \mathcal{E}'[\&_{\mathsf{p}}\{l_j : Q'_j\}_{j \in \mathcal{J}}]$ for some $P', \mathcal{J} \subseteq \mathcal{I}, \{Q'_j\}_{j \in \mathcal{J}}$.

(3) It suffices to prove $l \in \{l_j\}_{j \in \mathcal{J}}$. Let $\tilde{M}$ be the normal form of $M$; our approach is to prove that $\tilde{M}$ take a $\mathsf{select}_{\mathsf{p},\mathsf{q}}\ l$ step.

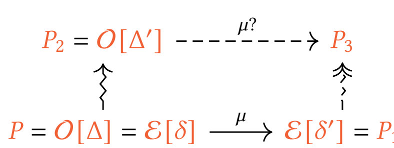
(4) Recall $\tilde{\mathcal{N}} \rightsquigarrow\!\!\!\!\rightarrow [\![M]\!]$. By Lemma A.12, $[\![\tilde{M}]\!]_{\mathsf{p}} = \tilde{\mathcal{E}}[\oplus_{\mathsf{q}}\ l\ \tilde{P}']$ and $[\![\tilde{M}]\!]_{\mathsf{q}} = \tilde{\mathcal{E}}'[\&_{\mathsf{q}}\{l_k : \tilde{Q}'_k\}_{k \in \mathcal{K}}]$ for some $\tilde{\mathcal{E}}, \tilde{\mathcal{E}}', \tilde{P}', \mathcal{K} \subseteq \mathcal{J}, \{\tilde{Q}'_k\}_{k \in \mathcal{K}}$.

(5) By Lemmas 3.4, 4.1 and 4.4 there exist $\mathcal{E}_{\mathsf{p}}, \Delta$ such that $\tilde{M} = \mathcal{E}_{\mathsf{p}}[\Delta]$. By Lemmas 4.6 and 4.7, $[\![\mathcal{E}_{\mathsf{p}}[\Delta]]\!]_{\mathsf{p}} = \tilde{\mathcal{E}}[\ [\![\Delta]\!]_{\mathsf{p}}\ ]$ with $[\![\Delta]\!]_{\mathsf{p}} = \oplus_{\mathsf{q}}\ l\ \tilde{P}'$ and $[\![\Delta]\!]_{\mathsf{q}} = \&_{\mathsf{p}}\{l_k : \tilde{Q}'_k\}_{k \in \mathcal{K}}$. Hence $\Delta$ can only be $\mathsf{select}_{\mathsf{p},\mathsf{q}}\ l\ M'$ for some $M'$.

(6) Since $[\![\Delta]\!]_{\mathsf{q}} = \&_{\mathsf{p}}\{l_k : \tilde{Q}'_k\}_{k \in \mathcal{K}}$ and $\Delta = \mathsf{select}_{\mathsf{p},\mathsf{q}}\ l\ M'$, by definition of projection $\mathcal{K}$ must be the singleton set $\{l\}$.

(7) Since $\mathcal{K} = \{l\}$ and $\mathcal{K} \subseteq \mathcal{J}$, we must have $l \in \{l_j\}_{j \in \mathcal{J}}$ as desired.

Now for the general case. Assuming $P \xrightarrow{\mu} P_1$ and $P \rightsquigarrow P_2$, we find $P_3$ such that the following diagram commutes:




We proceed by induction on the structure of $\mathcal{E}$.

(1) Assume $\mathcal{E} = \bullet$.

1. Assume $\mathcal{O} = \bullet$.
   a. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(prune)* rule. Then $P$ has the form $\&_{\mathsf{q}}\{l_i : Q_i\}_{i \in \mathcal{I}}$. But then $\mu$ could only be $\mathsf{select}_{\mathsf{q},\mathsf{p}}\ l$, a case we already handled separately above.
   b. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(commute)* rule. Then $P$ has the form $\mathcal{F}[\mathcal{B}[P']]$. But $\mathcal{E} = \bullet$ implies $P$ is a redex, and there are no redexes of that form.
   c. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(compute)* rule. Then $\mu = \tau$ and $P_2 = P_1$, so it suffices to let $P_3 = P_1$.
   d. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(bottom$_2$)* rule. Then $P = \bot\ \bot$ and $P \xrightarrow{\mu} P_1$ by *(bottom)*, with $\mu = \tau$ and $P_1 = \bot$. Likewise $P \rightsquigarrow P_2$ by *(bottom$_2$)* with $P_2 = (\lambda x : \bot.\, \bot)\ \bot$. Hence $P_2 \xrightarrow{\mu} P_3$ and $P_1 \rightsquigarrow\!\!\!\!\rightarrow P_3$ with $P_3 = \bot$.
2. Assume $\mathcal{O} = \mathcal{B}[\mathcal{O}']$ where $\mathcal{B} = \oplus_{\mathsf{q}}\ l\ \bullet$. Then $\mu = \mathsf{select}_{\mathsf{p},\mathsf{q}}\ l$ and $P_1 = Q$ and $Q = \mathcal{O}'[\Delta] \rightsquigarrow \mathcal{O}'[\Delta']$. Hence $P_2 \xrightarrow{\mu} P_3$ and $P_1 \rightsquigarrow P_3$ where $P_3 = \mathcal{O}'[\Delta']$.

3. Assume $\mathcal{O} = \mathcal{B}[\mathcal{O}']$ where $\mathcal{B} = \&_q\{l : \bullet\}$. As above.
4. Assume $\mathcal{O} = \mathcal{B}[\mathcal{O}']$ where $\mathcal{B} = (\lambda x : T.\ \bullet)\ R$. Since $\mathcal{O}[\Delta]$ is a redex $\delta$, $R$ must be a value; call that value $L$. Then $P_1 = \mathcal{O}'[\Delta][x := L]$ and $\mu = \tau$. We also have $P_2 \xrightarrow{\mu} P_3$ where $P_3 = \mathcal{O}'[\Delta'][x := L]$. By Lemma A.11, $P_1 = P_3$.
5. Assume $\mathcal{O} = \mathcal{F}[\mathcal{O}']$. Impossible: since $P = \mathcal{E}[\delta] = \delta$ is a redex, $\mathcal{O}'[\Delta]$ would have to be a value. But there is no $\mathcal{O}', \Delta$ such that $\mathcal{O}'[\Delta]$ is a value.

(2) Assume $\mathcal{E} = \mathcal{F}[\mathcal{E}']$ where $\mathcal{F} = \bullet\ Q$.

1. Assume $\mathcal{O} = \bullet$.
   a. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(commute)* rule. Then $P = \Delta = \mathcal{F}'[\mathcal{B}[R]]$ for some $\mathcal{F}', \mathcal{B}, R$. Proceed by case analysis on $\mathcal{F}'$ and $\mathcal{B}$.
      (1) Assume $\mathcal{F}' = \bullet\ Q$ and $\mathcal{B} = \oplus_q\ l\ \bullet$. Then $\mu = \text{select}_{p,q}\ l$ and $P_1 = \mathcal{F}[R]$ and $P_2 = \oplus_q\ l\ \mathcal{F}[R]$. If $P_3 = \mathcal{F}[R]$, then $P_1 = P_3$ and $P_2 \xrightarrow{\mu} P_3$.
      (2) Assume $\mathcal{F}' = \bullet\ Q$ and $\mathcal{B} = \&_q\{l : \bullet\}$. Then $\mu = \text{select}_{q,p}\ l$ and $P_1 = \mathcal{F}[R]$ and $P_2 = \&_q\{l : \mathcal{F}[R]\}$. If $P_3 = \mathcal{F}[R]$, then $P_1 = P_3$ and $P_2 \xrightarrow{\mu} P_3$.
      (3) Assume $\mathcal{F}' = \bullet\ Q$ and $\mathcal{B} = (\lambda x : T.\ \bullet)\ L$, where $L$ is a value. Then $\mu = \tau$ and $P_1 = \mathcal{F}[\ R[x := L]\ ]$ and $P_2 = (\lambda x : T.\ \mathcal{F}[R])\ L$. Since $x \notin \text{fv}(\mathcal{F})$, if $P_3 = \mathcal{F}[\ R[x := L]\ ]$ then $P_1 = P_3$ and $P_2 \xrightarrow{\mu} (\mathcal{F}[R])[x := L] = P_3$.
      (4) Assume $\mathcal{F}' = \bullet\ Q$ and $\mathcal{B} = (\lambda x : T.\ \bullet)\ R'$, where $R'$ is not a value. Then $P = ((\lambda x : T.R)\ R')\ Q$ and $P_1 = ((\lambda x : T.R)\ R'')\ Q$ for some $R''$ and $P_2 = (\lambda x : T.R\ Q)\ R'$. Then $P_2 \xrightarrow{\mu} P_3$ and $P_1 \rightsquigarrow P_3$ where $P_3 = (\lambda x : T.\ R\ Q)\ R''$.
      (5) Assume $\mathcal{F}' = L\ \bullet$. Then $\mathcal{F}'[\mathcal{B}[R]] = \mathcal{F}[\mathcal{E}'[\delta]]$ implies $L = \mathcal{E}'[\delta]$. Impossible, because there is no such $\mathcal{E}'$.
      (6) Assume $\mathcal{F}' = \text{if}\ \bullet\ \text{then}\ R_1\ \text{else}\ R_2$. Impossible, since $\mathcal{F}'[\mathcal{B}[R]] = \mathcal{F}[\mathcal{E}'[\delta]]$.
   b. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(prune)* rule. Impossible, since $\mathcal{O}[\Delta] = \mathcal{F}[\mathcal{E}'[\delta]]$.
   c. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(compute)* rule. Then $P = \Delta$ is a redex, implying $\mathcal{E}'[\delta]$ is a value, which is impossible.
   d. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(bottom$_2$)* rule. Then $\mathcal{E}'[\delta] = \bot$, which is impossible.
2. Assume $\mathcal{O} = \mathcal{F}'[\mathcal{O}']$ where $\mathcal{F}' = \bullet\ R$. Then the result follows by induction.
3. Assume $\mathcal{O} = \mathcal{F}'[\mathcal{O}']$ where $\mathcal{F}' = L\ \bullet$. Impossible, because $\mathcal{E}'[\delta]$ cannot be a value $L$.
4. Assume $\mathcal{O} = \mathcal{B}[\mathcal{O}']$ where $\mathcal{B} = (\lambda x : T.\ \bullet)\ R$. Impossible, because $\mathcal{B}[\mathcal{O}'[\Delta]] = \mathcal{F}[\mathcal{E}'[\delta]]$ implies $\mathcal{E}'[\delta] = \lambda x : T.\ \mathcal{O}'[\Delta]$, and $\mathcal{E}'[\delta]$ cannot be a value.
5. The remaining cases are impossible, because we must have $\mathcal{F}'[\mathcal{O}'[\Delta]] = \mathcal{F}[\mathcal{E}'[\delta]]$.

(3) Assume $\mathcal{E} = \mathcal{F}[\mathcal{E}']$ where $\mathcal{F} = L\ \bullet$. This case follows by the same arguments as the one above.

1. Assume $\mathcal{O} = \bullet$. Proceed by case analysis on the transition $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$.
   a. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(commute)* rule. Then $P = \Delta = \mathcal{F}'[\mathcal{B}[R]]$ for some $\mathcal{F}', \mathcal{B}, R$. Proceed by case analysis on $\mathcal{F}'$ and $\mathcal{B}$.
      (1) Assume $\mathcal{F}' = L\ \bullet$ and $\mathcal{B} = \oplus_q\ l\ \bullet$. Then $\mu = \text{select}_{p,q}\ l$ and $P_1 = \mathcal{F}[R]$ and $P_2 = \oplus_q\ l\ \mathcal{F}[R]$. If $P_3 = \mathcal{F}[R]$, then $P_1 = P_3$ and $P_2 \xrightarrow{\mu} P_3$.
      (2) Assume $\mathcal{F}' = L\ \bullet$ and $\mathcal{B} = \&_q\{l : \bullet\}$. Then $\mu = \text{select}_{q,p}\ l$ and $P_1 = \mathcal{F}[R]$ and $P_2 = \&_q\{l : \mathcal{F}[R]\}$. If $P_3 = \mathcal{F}[R]$, then $P_1 = P_3$ and $P_2 \xrightarrow{\mu} P_3$.
      (3) Assume $\mathcal{F}' = L\ \bullet$ and $\mathcal{B} = (\lambda x : T.\ \bullet)\ L'$, where $L'$ is a value. Then $\mu = \tau$ and $P_1 = \mathcal{F}[\ R[x := L']\ ]$ and $P_2 = (\lambda x : T.\ \mathcal{F}[R])\ L'$. Since $x \notin \text{fv}(\mathcal{F})$, if $P_3 = \mathcal{F}[\ R[x := L']\ ]$ then $P_1 = P_3$ and $P_2 \xrightarrow{\mu} (\mathcal{F}[R])[x := L'] = P_3$.

(4) Assume $\mathcal{F}' = L \bullet$ and $\mathcal{B} = (\lambda x : T.\, \bullet)\ R'$, where $R'$ is not a value. Then $P = L\ ((\lambda x : T.R)\ R')$ and $P_1 = L\ ((\lambda x : T.R)\ R'')$ for some $R''$ and $P_2 = (\lambda x : T.L\ R)\ R'$. Then $P_2 \xrightarrow{\mu} P_3$ and $P_1 \rightsquigarrow P_3$ where $P_3 = (\lambda x : T.\ L\ R)\ R''$.

(5) Assume $\mathcal{F}' = \bullet\ Q$. Impossible, since $\mathcal{O}[\Delta] = \mathcal{F}[\mathcal{E}'[\delta]]$ would imply $\mathcal{B}[R] = L$.

(6) Assume $\mathcal{F}' = \mathsf{if} \bullet \mathsf{then}\ R_1\ \mathsf{else}\ R_2$. Impossible, since $\mathcal{F}'[\mathcal{B}[R]] = \mathcal{F}[\mathcal{E}'[\delta]]$.

b. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(prune)* rule. Impossible, since $\mathcal{O}[\Delta] = \mathcal{F}[\mathcal{E}'[\delta]]$.

c. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(compute)* rule. Then $P = \Delta$ is a redex, implying $\mathcal{E}'[\delta]$ is a value, which is impossible.

d. Assume $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ by the *(bottom$_2$)* rule. Then $P = \bot\ \mathcal{E}'[\delta]$ and $P_1 = \bot\ \mathcal{E}'[\delta']$ and $P_2 = (\lambda x : \bot.\ \bot)\ \mathcal{E}'[\delta]$. Hence $P_2 \xrightarrow{\mu} P_3$ and $P_1 \rightsquigarrow P_3$ with $P_3 = (\lambda x : \bot.\ \bot)\ \mathcal{E}'[\delta']$.

2. Assume $\mathcal{O} = \mathcal{F}'[\mathcal{O}']$ where $\mathcal{F}' = \bullet\ R$. Then the two steps commute: $P_2 \xrightarrow{\mu} P_3$ and $P_1 \rightsquigarrow P_3$ where $P_3 = \mathcal{O}'[\Delta']\ \mathcal{E}'[\delta']$.
3. Assume $\mathcal{O} = \mathcal{F}'[\mathcal{O}']$ where $\mathcal{F}' = L \bullet$. Then the result follows by induction.
4. Assume $\mathcal{O} = \mathcal{B}[\mathcal{O}']$ where $\mathcal{B} = (\lambda x : T.\ \bullet)\ R$. Then $P = (\lambda x : T.\ \mathcal{O}'[\Delta])\ \mathcal{E}'[\delta]$ and the two steps commute: $P_2 \xrightarrow{\mu} P_3$ and $P_1 \rightsquigarrow P_3$ where $P_3 = (\lambda x : T.\ \mathcal{O}'[\Delta'])\ \mathcal{E}'[\delta']$.
5. The remaining cases are impossible, because we must have $\mathcal{F}'[\mathcal{O}'[\Delta]] = \mathcal{F}[\mathcal{E}'[\delta]]$.

(4) Assume $\mathcal{E} = \mathcal{F}[\mathcal{E}']$ where $\mathcal{F} = \mathsf{if} \bullet \mathsf{then}\ Q_1\ \mathsf{else}\ Q_2$.

1. Assume $\mathcal{O} = \bullet$. Since $P$ is an if-expression, the transition $\mathcal{O}[\Delta] \rightsquigarrow \mathcal{O}[\Delta']$ cannot be a *(prune)*, *(commute)*, or *(bottom$_2$)* step; it must be a *(compute)* step. In particular, we must have $\Delta \mapsto \Delta'$ by a *(p-if)* reduction, and therefore $P = \mathsf{if}\ c\ \mathsf{then}\ Q_1\ \mathsf{else}\ Q_2$ for some $c$. But there is no $\mathcal{E}'$ such that $\mathcal{E}'[\delta] = c$, so this case is impossible.
2. Assume $\mathcal{O} = \mathcal{F}'[\mathcal{O}']$ where $\mathcal{F}' = \mathsf{if} \bullet \mathsf{then}\ R_1\ \mathsf{else}\ R_2$. Then the result follows by induction.
3. The remaining cases are impossible, because we must have $\mathcal{F}'[\mathcal{O}'[\Delta]] = \mathcal{F}[\mathcal{E}'[\delta]]$.

□

Received 2025-02-27; accepted 2025-06-27