Pirouette: Higher-Order Typed Functional Choreographies

ANDREW K. HIRSCH, Max Planck Institute for Software Systems, Germany
DEEPAK GARG, Max Planck Institute for Software Systems, Germany

We present Pirouette, a language for typed higher-order functional choreographic programming. Pirouette offers programmers the ability to write a centralized functional program and compile it via *endpoint projection* into programs for each node in a distributed system. Moreover, Pirouette is defined generically over a (local) language of messages, and lifts guarantees about the message type system to its own. Message type soundness also guarantees deadlock freedom. All of our results are verified in Coq.

CCS Concepts: • **Theory of computation → Functional constructs**; *Type structures*; • **Computing methodologies → Concurrent programming languages**.

Additional Key Words and Phrases: Concurrency, Choreographies, Functional programming

ACM Reference Format:
Andrew K. Hirsch and Deepak Garg. 2022. Pirouette: Higher-Order Typed Functional Choreographies. *Proc. ACM Program. Lang.* 6, POPL, Article 23 (January 2022), 27 pages. https://doi.org/10.1145/3498684

1 INTRODUCTION

Higher-order typed functional programming has proven to be a powerful technique for writing single-machine programs. It allows for strong guarantees through types along with code reuse through higher-order programming. However, currently, writing *distributed* programs using functional programming requires writing separate code for each node in the distributed system, then using send and receive expressions to transmit data between nodes. This makes it easy to write code that *deadlocks*, or gets stuck because patterns of sends and receives do not match.

Session types [Carbone et al. 2007; Dardha et al. 2012; DeYoung et al. 2012; Scalas and Yoshida 2019; Toninho et al. 2012; Wadler 2012] offer one solution. Session types describe the pattern of sends and receives in a program, allowing a compiler to catch the possibility of deadlock. However, session types are complicated to work with, and the programmer is still left trying to match up send and receive patterns by hand.

Choreographic programming [Cruz-Filipe and Montesi 2017a,b; Dalla Preda et al. 2015; Lanese et al. 2013; Montesi 2013, 2020] offers another solution. This is a programming paradigm that writes the distributed program as a single program, ensuring that sends and receives match by combining send and receive into one construct. Choreographic languages guarantee *deadlock freedom by design*, so the programmer gets strong guarantees on the communication patterns in their program. Until now, however, choreographic programming forced the user into a lower-order, imperative, and un(i)typed universe. This paper presents Pirouette, the first language for choreographic programming which is also higher-order, functional, and typed.

Authors' addresses: Andrew K. Hirsch, Max Planck Institute for Software Systems, Saarland Informatics Campus, Saarbrücken, Germany, akhirsch@mpi-sws.org; Deepak Garg, Max Planck Institute for Software Systems, Saarland Informatics Campus, Saarbrücken, Germany, dg@mpi-sws.org.

2475-1421/2022/1-ART23
https://doi.org/10.1145/3498684

Consider a standard example: the bookseller protocol. In this protocol, a buyer is looking to buy a book, so they send the book's title to a bookseller. The bookseller returns its price, and the buyer checks if that price is within their budget. If they can afford the book, they inform the bookseller who tells them a delivery date for their book; otherwise, they tell the seller they will not buy the book. After this, the protocol ends.

To write this program with session types, we first create a type describing the interactions from both points of view. That is, we create a type for the buyer that includes "send the title to the seller, and then receive back a price" along with a (dual) type for the seller that includes "receive the title from the buyer, then send them the price." First, we must ensure that the two types match: when the buyer sends to the seller, the seller receives from the buyer. These types do not describe details like "if the book is within budget, tell the seller yes." Instead, it says "either send the seller 'yes' and receive a delivery date, or send them 'no' and end the protocol". We then write a program for the buyer and another for the seller, which contain all the nasty details, and a type-checker ensures that these programs have the correct communication patterns.

In contrast, a choreographic programmer would write a single program, and expect the language and compiler to ensure that the resulting programs for the buyer and bookseller are deadlock free. For instance, such a programmer might write the following:

```
Buyer.book_title ⇝ Seller.b;
Seller.prices[b] ⇝ Buyer.p;
if Buyer.(p < budget)
then Buyer[L] ⇝ Seller;
     Seller.get_delivery_date(b) ⇝ Buyer.d_date;
     Buyer.(Some d_date)
else Buyer[R] ⇝ Seller;
     Buyer.None
```

Note that the choreography is a program, not merely the type of another program. In this program, Buyer sends a string book_title to Seller, who binds that string to the variable b. Seller then sends a message to Buyer, which is the result of looking up the price of the book in a map that Seller holds. Note that this message is written in a separate programming language, which defines the map-lookup syntax. Once Buyer has the price, they decide whether or not to buy based on their budget. Either way, they inform Seller whether they took the left (L) or the right (R) branch. In the left branch, Seller calls some function in the local language to get the delivery date. (We use the syntax $f(x)$ for function calls in the local syntax to emphasize that these are function calls in the local language, whereas Pirouette uses the syntax $F\ X$ for its function calls.) Seller transmits the result of this function call to Buyer, who returns the date, wrapped so that the entire program returns a value on buyer of type option date. In the right branch, Buyer simply returns None.

Note that communication is always matched up: sends and receives are combined, which guarantees deadlock freedom. The return value on Buyer makes this closer to functional programming than previous work on choreographies. Note that choreographies returning values and having standard simple types (i.e., not session types) are both innovations of this work.

Choreographies excel when more than two parties must interact. For instance, Carbone and Montesi [2013] suggest a change to the bookseller protocol similar to the following: Seller sends the price to two buyers who want to share the purchase of a book, Buyer_1 and Buyer_2. Buyer_2 then tells Buyer_1 how much they are willing to contribute to the purchase, who then responds to Seller

as before. We can write a modified choreography as follows:

```
Buyer1.book_title ⇝ Seller.b;
Seller.prices[b] ⇝ Buyer1.p;
Seller.prices[b] ⇝ Buyer2.p;
Buyer2.(p/2) ⇝ Buyer1.contrib;
if Buyer1.(p − contrib < budget)
then Buyer1[L] ⇝ Seller;
     Seller.get_delivery_date(b) ⇝ Buyer1.d_date;
     Buyer1.(Some d_date)
else Buyer1[R] ⇝ Seller;
     Buyer1.None
```

In order to use session types to describe this communication pattern, we must use *multiparty session types*, a major increase in complexity [Carbone et al. 2007; Scalas and Yoshida 2019]. However, choreographies handle this without difficulty.

The choreographies explained so far can be expressed in prior work. This paper introduces *functional choreographies*. To understand the need for these, notice how we can abstract out a pattern from our two protocols. A buyer (say Buyer) sends a book title to Seller, who looks up its price. There is then some process, possibly involving communications, which results in a decision by Buyer, who then informs Seller of that decision. If Buyer decides to buy the book, they get back a delivery date, otherwise they get nothing. We can modify the choreography so that the decision is its own function as follows:

```
fun Bookseller(F) := Buyer.book_title ⇝ Seller.b;
                     let Buyer.decision := F Seller.(prices[b])
                     in if Buyer.decision
                        then Buyer[L] ⇝ Seller;
                             Seller.get_delivery_date(b) ⇝ Buyer.the_date;
                             Buyer.(Some the_date)
                        else Buyer[R] ⇝ Seller;
                             Buyer.None
```

Here, F is a function with a choreographic body which takes a price on Seller as input and outputs a Boolean on Buyer. We can implement either of the previous protocols by changing F:

```
fun F(Seller.p) := Seller.p ⇝ Buyer.p;
                   Buyer.(p < budget)
```

```
fun F(Seller.p) := Seller.p ⇝ Buyer.p;
                   Seller.p ⇝ Buyer2.p;
                   Buyer2.(p/2) ⇝ Buyer.contrib;
                   Buyer.(p − contrib < budget)
```

Moreover, we can give F a type which enforces that it takes a price located on Seller to a Boolean value on Buyer, as desired. We write this type price@Seller → Buyer.bool. Then we can give Bookseller the type (price@Seller → Buyer.bool) → Buyer.date, indicating that it is a higher-order function which expects an input that abstracts the decision in the bookseller example. This illustrates the programming convenience gained by combining higher-order functional programming with the choreographic-programming paradigm. For instance, previous works on choreographies could not treat choreographic functions as data, preventing this form of abstraction [Cruz-Filipe and Montesi 2017b].

Contributions. As stated before, Pirouette is the first (higher-order) choreographic functional programming language. Pirouette features simple located types of the form Buyer.bool and familiar

type constructs like function spaces. Following the choreography philosophy, Pirouette guarantees deadlock freedom by design, without the use of complicated session types. Previously, higher-order choreographic programming was only supported through the informal development in Choral [Giallorenzo et al. 2020], which is useful despite offering no guarantees. In the following, we point out some salient technical contributions of our design.

First, Pirouette is *generic* in the language of messages. Note that, sometimes, locations send non-atomic messages like $p/2$, which can be arbitrary expressions. In previous work, messages were either from a particular (and very simple) language [Cruz-Filipe and Montesi 2017a] or were assumed to compute to a value in finite time [Carbone and Montesi 2013]. Pirouette is defined generically on top of *any* message language (which we refer to as the *local* language), with very few syntactic constraints. We further show that a sound type system for the expression language can be lifted to the level of choreographies.

Two key features of choreographies are *out-of-order execution* and *endpoint projection*. We can think of a choreography as an arbitrary interleaving of communication between programs running at different locations. However, programs with the same communication pattern can have other interleavings as well. Choreographies, despite *syntactically* being one arbitrary interleaving are able to *semantically* represent all interleavings by allowing out-of-order execution. We follow previous work by defining an equivalence relation $\equiv$ on choreographies to reason about out-of-order execution. However, unlike previous work, we are able to show that defining our operational semantics by appealing to $\equiv$ gives a weaker-than-desired semantics. Pirouette's semantics provides a stronger form of out-of-order execution via a novel combination of a labeled-transition system and a *block set*, which guarantees that out-of-order execution does not violate causality.

Endpoint projection formalizes the intuition of a choreography as representing a collection of programs running at different locations by extracting a program for each location (later, we use the term *control program* for the projected program at each location). This justifies choreographies as a way of writing distributed programs, and allows us to state and prove that Pirouette programs are deadlock free. We follow a new design principle for choreographies: *equivalence begets equality*. That is, equivalent choreographies always project to exactly the same program for each location. In previous work, equivalence was used more liberally which prevented such a clean theorem. For instance, Cruz-Filipe and Montesi [2017a] use equivalence to reason about recursion unfolding, so equivalent programs may project to programs where unfolding has and has not been applied.

Moreover, somewhat surprisingly, we show that deadlock freedom is a corollary to the soundness of our type system as well as the soundness and completeness of endpoint projection. Previous work was able to take advantage of the assumption that messages always produce a value along with the lower-order nature of their choreographies. Because our choreographies are higher-order and our messages are not assumed to compute, we have to appeal to the soundness of our type system to ensure that our choreographies are always able to take a step.

We have formalized our entire development in Coq and mechanically verified proofs of all of our theorems. As mentioned by Cruz-Filipe et al. [2019], there have been several instances of flaws found in proofs of major theorems in concurrency theory in recent years. Therefore, in order to show that Pirouette's guarantees about deadlock freedom are trustworthy, we formalize our arguments. In soon-to-be-published work, a small choreography language has been formalized along with its endpoint-projection operation [Cruz-Filipe et al. 2021b,c]. However, Pirouette is a much more substantial language than was formalized in that work.

To summarize, we make the following contributions:

- We introduce Pirouette, the first functional choreography language. We present its operational semantics and a (simple) type system. The operational semantics allow for out-of-order

execution, mimicking the execution of a distributed program. These semantics are based on a novel idea of blocking sets – locations that are blocked on other operations and cannot reduce (Section 3).
- We describe a general set of constraints on the local (message) language, allowing almost any expression-based language to be used as the local language (Section 2). Type soundness for functional choreographies lifts from type soundness for the local language (Section 3.2).
- We study equivalence for functional choreographies, which allows for reasoning about out-of-order execution. We show that defining out-of-order execution based on this equivalence leads to a weaker-than-desired operational semantics when local reduction is allowed (Section 4).
- We show how endpoint projection extracts programs with explicit send and receive constructs from functional choreographies. This translation is sound and complete. Moreover, we show that well-typed, projected systems are deadlock free by design, and that this deadlock freedom follows from the soundness of the type system and the soundness and completeness of endpoint projection (Section 5).
- All our metatheory has been formalized in Coq and all our theorems have been mechanically verified. We discuss the particularities of our Coq implementation in Section 6.

Concurrent, independent work [Cruz-Filipe et al. 2021a] also explores higher-order functional choreographies. However, they have a very different design philosophy, which leads to a very different technical setup. For more comparison, see the discussion of related work in Section 7.

2 SYSTEM MODEL

We begin by discussing the assumptions we make about the system that Pirouette programs run on. We have made these assumptions as lightweight as possible. In particular, we assume that the system consists of a collection of nodes, each of which can run local programs and which can send and receive messages from other nodes. We also allow local programs to be typed, so that we can lift the type system to the choreography level.

2.1 Locations

We assume a set $\mathcal{L}$ of *locations*, which we write as ℓ, ℓ_1, ℓ_2, and so on. These names are treated atomically, so we do not assume any additional operations on locations. However, we do assume that location equality is decidable, so we can distinguish different locations.

Intuitively, locations refer to nodes in a distributed system. However, it's worth noting that there is nothing that prevents a node from being a thread, or a process, or any other entity that can run Turing-complete programs and send and receive messages.

2.2 Communication

We assume that every location can communicate with every other location synchronously. That is, if ℓ_1 sends a message to ℓ_2, then ℓ_1 does not continue until ℓ_2 has received the message, and then ℓ_1 may continue. Message sending is instantaneous and certain: messages do not get "lost in the air."

Nodes should be able to send and receive two kinds of messages: values of local programs (described below) and two special *synchronization messages*, written L and R. These will be used in the choreography language to ensure that different locations stay in lock-step with each other.

We also require that each node be able to run a functional *control* program which can send and receive messages, while also running programs in the local language. We describe the precise requirements in Section 5.1, when we have the necessary background.

2.3 Local Programs

We assume that every node runs programs in a *local* expression-based language. Our design treats this language generically, requiring only that it allows certain operations and equations.

Our first requirement is that expressions include *variables*. We model messages as values, and receipt as binding a value to a variable. We implement variable binding via substitution, which we write $e_1[x \mapsto e_2]$. Substitution must satisfy three standard equations:

- $x[x \mapsto e] = e$
- $e[x \mapsto x] = e$
- $e_1[x \mapsto e_2][y \mapsto e_3] = e_1[y \mapsto e_3][x \mapsto (e_2[y \mapsto e_3])]$ whenever $x \notin \mathrm{FV}(e_3)$

We require a function $FV(e)$ which returns the set of free variables in e. We require that if $x \notin \mathrm{FV}(e_1)$, then $e_1[x \mapsto e_2] = e_1$.

We only send values as messages, so we assume a predicate $\mathrm{Value}(e)$ which determines whether e is a value. We require two special values, true and false, which we use for branching in choreographies. We additionally require that all values are closed, in order to allow them to be sent. To see why, imagine that we send some open expression e from ℓ_1 to ℓ_2. Since e is open, e contains some free variable x, which refers to some data on ℓ_1. However, when we send e to ℓ_2, this information is lost and x might be captured by a binder in ℓ_2's program.

Finally, we require that an operational semantics be defined relationally for local expressions. We write $e_1 \Rightarrow_{\mathrm{e}} e_2$ to denote that e_1 steps to e_2 in the operational semantics. The only requirement on this semantics is that values do not take steps: if $\mathrm{Value}(v)$, then $v \not\Rightarrow_{\mathrm{e}} e$ for any e.

Examples. Simply-typed functional languages easily satisfy the requirements above. However, any expression-based language can be used, not only ones which define functions. We discuss two examples here.

Example 1 (Call-by-Value λ-Calculus)**.** The call-by-value λ-calculus, extended with recursive functions, Boolean values and if-then-else expression, almost fits our requirements. However, we must restrict values to be closed.

Example 2 (A Natural-Number Language)**.** We provide a language with the following syntax:

$$\text{Expressions} \quad e \quad ::= \quad x \mid 0 \mid S\,e \mid \mathsf{true} \mid \mathsf{false}$$

Intuitively, 0 stands for 0 as a natural number, S is the successor operation on natural numbers, and true and false stand for Boolean truth and falsity, respectively. Since there are no binders, substitution is easy to define. Any closed term is a value.

This is similar to the language of messages in Cruz-Filipe and Montesi [2017a], but modified to fit our requirements. In particular, we (a) allow more than one variable, which is treated via substitution instead of as a reference to state, and (b) add the true and false terms.

We include this language to demonstrate that our requirements do not force the choice of λ-calculus as an expression language. Indeed, we will see later that we can equip this language with a type system which results in a sound choreographic type system.

2.4 Typed Local Programs

The guarantees we provide for Pirouette depend on the guarantees provided by the local language's type system. If we do not consider the local type system, then we are able to provide a sound and complete translation to a language with explicit send and receive constructs. If the local type system guarantees preservation, but not progress (as in a unityped system), then we are also able to prove preservation of the choreographic type system. Finally, soundness of the local type system implies not only type soundness for Pirouette, but also deadlock freedom.

$$\textsc{Var}\ \frac{}{\Gamma, x : t \vdash x : t} \qquad \textsc{True}\ \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \textsc{False}\ \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

$$\textsc{Exchange}\ \frac{\Gamma, x : t_1, y : t_2 \vdash e : t_3}{\Gamma, y : t_2, x : t_1 \vdash e : t_3} \qquad \textsc{Weakening}\ \frac{\Gamma \vdash e : t_1}{\Gamma, y : t_2 \vdash e : t_1}$$

$$\textsc{Strengthening}\ \frac{\Gamma, x : t \vdash e : t \qquad x \notin \mathrm{FV}(e)}{\Gamma \vdash e : t} \qquad \textsc{Substitution}\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma, x : t_1 \vdash e_2 : t_2}{\Gamma \vdash e_2[x \mapsto e_1] : t_2}$$

Fig. 1. Required-Admissible Rules for Local Type Systems

We allow the types for the local language to be any language of simple types. We require that true and false have the same type, which we refer to as bool. Note, however, that this type may not be called bool, as we will see in Example 5.

We assume that the type system can be presented in terms of a judgment $\Gamma \vdash e : t$, where Γ is a sequence of variable-type pairs written $x : t$. We additionally require that typing be unique: if $\Gamma \vdash e : t_1$ and $\Gamma \vdash e : t_2$, then $t_1 = t_2$. We only use this requirement to show that types play well with equational reasoning (Theorem 3). While this requirement is unusual, we note that it is usually satisfied by simple-type systems. We conjecture that this requirement could be removed with a small change to our choreographic language; we return to this point in Section 4.

The other requirements can all be framed as admissible rules; those rules can be found in Figure 1. Since these rules should be admissible, we do *not* require that these are actual rules in the type system; merely that we can use them to build typing proofs. This will be important when building typing proofs for choreographies on top of typing proofs for the local language.

First, we require that variables be typed according to the context, and that true and false be typed in the bool type. We also require that the standard structural rules of Exchange and Weakening be allowed. The Strengthening rule is unusually-presented, but is common in most type systems: non-free variables can be safely removed from the typing context. Finally, we require the standard property of Substitution: substituting a well-typed variable into a well-typed expression yields a well-typed expression.

Sound Type Systems. We say that a local type system is *sound* if it additionally satisfies the following three requirements:

- (Boolean Invertibility) The type bool is invertible for values: if v is a value and $\vdash v : \mathsf{bool}$ then either $v = \mathsf{true}$ or $v = \mathsf{false}$.
- (Preservation) If $e_1 \Rightarrow_{\mathrm{e}} e_2$ and $\Gamma \vdash e_1 : t$, then $\Gamma \vdash e_2 : t$.
- (Progress) If e_1 is closed and $\vdash e_1 : t$, then either e_1 is a value or there is an e_2 such that $e_1 \Rightarrow_{\mathrm{e}} e_2$.

Examples. Each of the languages that served as examples above can be given type systems that satisfy the requirements above. In fact, we can give the λ-calculus two type systems, though one is not sound.

Example 3 (Simply-Typed λ-Calculus)**.** This is the paradigmatic example of a typed local language. None of the rules in Figure 1 are difficult; most of the proofs are completely standard. Moreover, it is sound: the progress and preservation proofs are standard, and the invertability proof is easy.

$$
\begin{array}{lllll}
\text{Locations} & \ell \in \mathcal{L} & & & \\
\text{Synchronization Labels} & d & ::= & \mathsf{L} \mid \mathsf{R} & \\
\text{Choreographies} & C & ::= & X \mid \ell.e \mid \ell_1.e \rightsquigarrow \ell_2.x;\ C & \\
 & & \mid & \mathsf{if}\ \ell.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \mid \ell_1[d] \rightsquigarrow \ell_2;\ C & \\
 & & \mid & \mathsf{let}\ \ell.x := C_1\ \mathsf{in}\ C_2 \mid \mathsf{fun}\ F(\ell.x) := C \mid \mathsf{fun}\ F(X) := C & \\
 & & \mid & C\ \ell.e \mid C_1\ C_2 &
\end{array}
$$

Fig. 2. Functional Choreographies Syntax

Example 4 (Typed Natural-Numbers)**.** We use two types: int and bool. Then, we have the following rules:

$$\frac{x : t \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash 0 : \mathsf{int}} \qquad \frac{\Gamma \vdash t : \mathsf{int}}{\Gamma \vdash S\,t : \mathsf{int}} \qquad \frac{}{\Gamma \vdash \mathsf{true} : \mathsf{bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} : \mathsf{bool}}$$

This makes the rules in Figure 1 easy to prove, and soundness is trivial.

Example 5 (Unityped λ-Calculus)**.** Using the idea of "untyped is unityped", we develop a type system for λ-calculus with only one type, $*$, and only one typing rule:

$$\textsc{Trivial}\ \frac{}{\Gamma \vdash e : *}$$

Here, we satisfy the requirement to have a type for Boolean values by setting bool $= *$. The rules in Figure 1 are trivial, as is Preservation. However, this system is *not* sound: bool is not invertible, and uni-typed λ-calculus does not satisfy progress (programs may get stuck).

3 FUNCTIONAL CHOREOGRAPHIES

We introduce Pirouette, our language for writing distributed programs in a functional, choreographic style. We guarantee deadlock freedom along with the standard static guarantees of simple-type systems. Moreover, we allow higher-order computations by allowing choreographies to be passed to other choreographies as inputs.

The syntax of Pirouette can be found in Figure 2. Note that choreographies contain two types of variables: choreography variables, which stand for the result of a distributed computation (i.e., a choreography), and local variables which are the variables of the local language seen in Section 2. We write choreography variables in upper case (X, Y, etc.) and choreography variables in lower case (x, y, etc.).

Every location has its own namespace of local variables, so $\ell_1.x \neq \ell_2.x$. This is reflected in the defintion of substitution: substitution of choreography variables has the standard form $C_1[X \mapsto C_2]$, while substitution of local variables has the form $C[\ell|\ x \mapsto e]$. The first notion of substitution is standard. The second walks through a term looking for a subterm of the form $\ell.e'$, and then replaces e' with $e'[x \mapsto e]$. (Note that the location name ℓ must be the same as the one in the substitution!)

We write $\ell.e$ to represent the choreography that returns the result of running e on ℓ. To use this value in a future choreography, we use the syntax $\mathsf{let}\ \ell.x := C_1\ \mathsf{in}\ C_2$. This runs C_1 until it returns a value on ℓ, and then binds that value to x under ℓ in C_2.

We write $\ell_1.e \rightsquigarrow \ell_2.x;\ C$ to represent ℓ_1 evaluating e and then sending the resulting value v to ℓ_2. The variable $\ell_2.x$ is bound to v in the continuation choreography C, representing ℓ_2's receipt of the message.

The distributed program can branch based on the result of a test on a local machine. We write this if $\ell.e$ then C_1 else C_2. However, this can quickly break causality. To see why, consider the program if $\ell_1.e$ then $\ell_2.3$ else $\ell_2.4$. In this program, ℓ_2 behaves differently depending on the behavior of a test performed on ℓ_1, but ℓ_2 has never been told about the result of that test! In order to fix this problem, we require that ℓ_1 inform ℓ_2 of which branch was taken before ℓ_2 can behave differently in the two branches. We do this using the syntax $\ell_1[d] \rightsquigarrow \ell_2;\ C$. Here d can either be L or R, where L represents taking the then branch, while R represents taking the else branch. Thus, we can safely write the following program:

$$\begin{array}{l}\text{if } \ell_1.e\\ \text{then } \ell_1[\mathsf{L}] \rightsquigarrow \ell_2;\ \ell_2.3\\ \text{else } \ell_1[\mathsf{R}] \rightsquigarrow \ell_2;\ \ell_2.4\end{array}$$

Note that this is only required because ℓ_2*'s behavior differs* in the two branches. If ℓ_2 behaves the same, no synchronization is required. Thus, the following program is okay:

$$\begin{array}{l}\text{if } \ell_1.e\\ \text{then } \ell_3.e' \rightsquigarrow \ell_2.x;\\ \qquad \ell_1[\mathsf{L}] \rightsquigarrow \ell_2;\ \ell_2.x + 2\\ \text{else } \ell_3.e' \rightsquigarrow \ell_2.x;\\ \qquad \ell_1[\mathsf{R}] \rightsquigarrow \ell_2;\ \ell_2.x + 3\end{array}$$

Because it behaves the same in both branches, ℓ_3 never needs to be informed about which branch is taken, even though it appears in the branches. Moreover, ℓ_2 only needs to be informed about which branch is taken after it receives its message from ℓ_3.

There are two types of functions available in Pirouette, local and global functions. Local functions expect a local value as input, stored on some particular node, whereas global functions expect a choreography as input. Both types of functions may be recursive. We write fun $F(\ell.x) := C$ for the (recursive) function named F which expects a local value on ℓ as input, and has body C. We write $C\ \ell.e$ for the application of function C to input e which is stored on ℓ. For the global function named F which takes an input named X and has body C, we write fun $F(X) := C$. As is traditional, we write $C_1\ C_2$ for the application of function C_1 to C_2.

We adopt a call-by-value semantics, so we evaluate inputs to values before applying functions. As described in Section 2, local values are defined by the local language and are always closed. Choreography values are programs of any of the forms: (a) $\ell.v$ (where v is a local value), (b) fun $F(\ell.x) := C$ (where the only free variables in C are F and $\ell.x$), or (c) fun $F(X) := C$ (where the only free variables in C are F and X).

We define an operation $\mathsf{LN}(C)$ which collects all of the location names in C in a set. We write $\mathrm{FCV}(C)$ for the free choreography variables in C, and $\mathrm{FEV}(C)$ for the collection of free expression variables in C, tagged with the locations that own them. We write $\mathrm{FEV}_\ell(C)$ for the collection of expression variables free under ℓ in C.

3.1 Operational Semantics

Intuitively, if two locations ℓ_1 and ℓ_2 both take actions, they should be able to do this in either order. For instance, consider the Pirouette program $\ell_1.2 + 3 \rightsquigarrow \ell_2.x;\ \ell_3.3 * 4 \rightsquigarrow \ell_2.y;\ C$. Here, ℓ_1 and ℓ_3 are both working on computations whose results they expect to send to ℓ_2. Since these are different locations, both should be able to work on their programs at the same time. Thus in the semantics we should be able to reduce this program to either $\ell_1.5 \rightsquigarrow \ell_2.x;\ \ell_3.3 * 4 \rightsquigarrow \ell_2.y;\ C$ or $\ell_1.2 + 3 \rightsquigarrow \ell_2.x;\ \ell_3.12 \rightsquigarrow \ell_2.y;\ C$, performing the local reductions in either order. However, ℓ_2 is just one location, so it cannot listen for two messages at once. Thus, even if we reduce the above

program to $\ell_1.5 \rightsquigarrow \ell_2.x;\ \ell_3.12 \rightsquigarrow \ell_2.y;\ C$ we are forced to reduce the send from ℓ_1 before the send from ℓ_3, since the second is waiting on ℓ_2.

In order to allow for this behavior, we keep track of a set of *blocked* locations in our operational semantics. Intuitively, blocked locations cannot take a step. By keeping track of what locations are blocked, we can allow out-of-order-execution among non-blocked locations.

However, this is not quite enough to get the behavior we want. Consider this program, which represents a system where ℓ_1 branches on e and then does nothing, while ℓ_2 returns the result of $3+5$ independent of ℓ_1's choice: $\text{if } \ell_1.e \text{ then } \ell_2.3 + 5 \text{ else } \ell_2.3 + 5$. Here, ℓ_2 ought to be able to make progress, reducing this program to $\text{if } \ell_1.e \text{ then } \ell_2.8 \text{ else } \ell_2.8$. Note that this progress was atomic. Thus, it would be illegitimate to reduce different programs in the two branches. For instance, reducing $\text{if } \ell_1.e \text{ then } \ell_2.3 + 5 \text{ else } \ell_2.3 * 4$ to $\text{if } \ell_1.e \text{ then } \ell_2.8 \text{ else } \ell_2.12$ requires reducing different programs in the two branches, whereas, in reality, only one of those two programs should be reduced and that program should be chosen by ℓ_1.

In order to prevent this type of reduction, we track *what* reduction is happening, resulting in a labeled transition system. We refer to labels in Pirouette as "redices," and to a single label as a "redex." In the second-to-last example above, we note that in each branch, we reduce a 3+5 to 8 on ℓ_2. Since we are doing the same reduction in each branch, we can reduce the whole program.

Selected rules from the operational semantics can be found in Figure 3. The full rules, along with the syntax of redices, can be found in the accompanying technical report [Hirsch and Garg 2021].

Each rule has the form $C_1 \overset{R\ B}{\Longrightarrow}_{\text{c}} C_2$, where R is a redex, B is a set of locations, and C_1 and C_2 are choreographies. Intuitively, this says that C_1 can reduce to C_2 using redex R even if every location in B is blocked. With this interpretation, the rule SendE allows a location to reduce a message to a value before sending it. The notation $\ell_1.(e_1 \Rightarrow e_2) \rightsquigarrow \ell_2$ is the redex for this reduction rule. Note that we check both that $\ell_1 \notin B$ and that $\ell_1 \neq \ell_2$ in the premise of this rule. The first check ensures that ℓ_1 is not blocked, since the reduction happens at that location. The second check reflects the fact that sends from a node to itself is not meaningful.

The rule SendI allows reductions under a send construct, but only when the reduction can take place with both the sender and the receiver blocked. This formalizes the intuition that while some locations are waiting, other locations can take actions. Sending is formalized with the rule SendV. This removes a send entirely. Note that in the substitution, ℓ_2's variable is substituted with the message. This formalizes the intuition that sends are modeled by binding the message to a variable in the receiver's program.

The IfI rule is the only rule that makes use of redices. By ensuring that the same redex is reduced on each branch of the if, we make sure that the only reductions that can be made are those which are invariant under which branch is taken. Note that we again ensure that ℓ_1 is not taking an action in this reduction step, since it is currently occupied with the if itself.

The rules SyncI and Sync demonstrate the rather subtle effect that Sync has on the semantics. The first rule demonstrates that synchronization induces blocking on the relevant locations. However, Sync shows that this is the only effect that it has on the semantics of the choreography.

Finally, the rules AppGlobalFun, AppGlobalArg, and AppGlobal demonstrate how functions are treated. Both the function and its argument can be evaluated in any order, but the redex is tagged with which choice is made. This ensures that if statements don't reduce a function in one branch and its argument in the other. Finally, the AppGlobal rule shows that the semantics of a function is given as standard, with a parallel substitution. However, we require that there be no blockers. This comes from the fact that all of the locations work together to reduce functions at the same time.

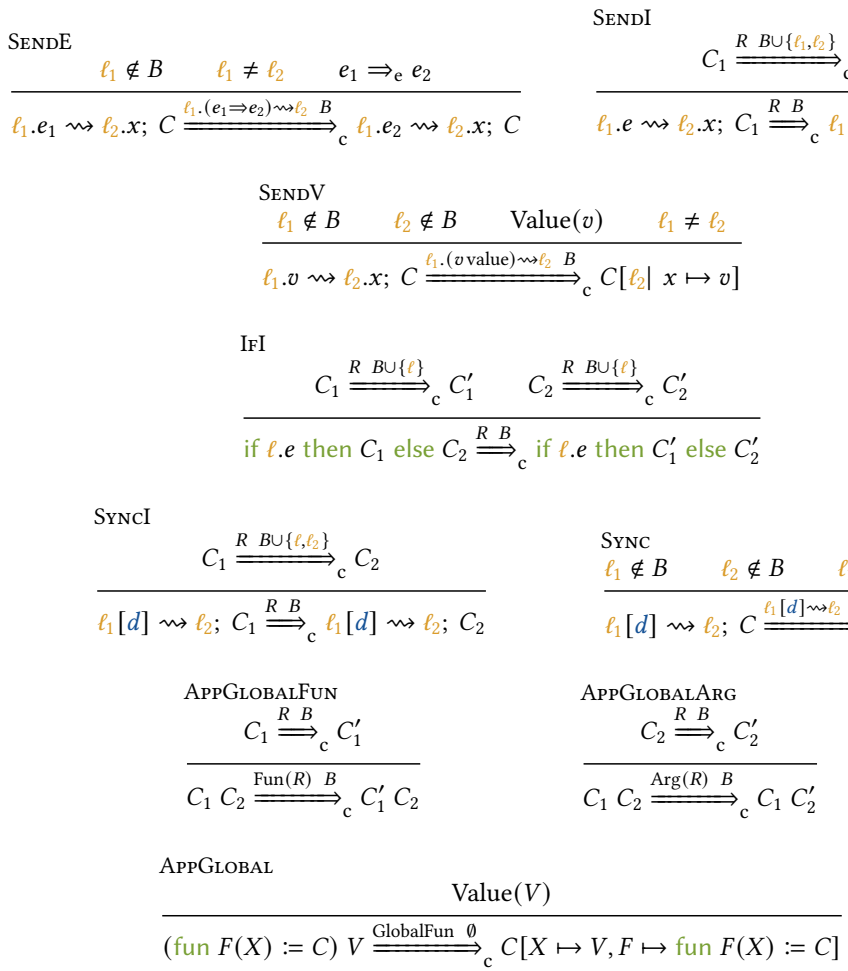

Fig. 3. Selected Choreography Operational Semantics

3.2 Type System

Assuming that the local expression language has a type system as described in Section 2.4, we can develop a language of types for Pirouette. The syntax of Pirouette types can be found in Figure 4, along with selected typing rules. The full typing rules are in the accompanying technical report [Hirsch and Garg 2021].

There are three categories of Pirouette types, corresponding to the three categories of choreographic values. The first category is a local type at some location, which we write $\ell.t$. This is the type given to values of the form $\ell.v$. Then there is the type of local functions, which we write $t_1@\ell \to \tau_2$. Here, t_1 is an expression type, corresponding to the type of the input to the function, which is expected to be located on ℓ. The function then returns a τ_2, which is a choreography type. Finally, there is the type of global functions which take an arbitrary choreographic type as an input. Thus, we write $\tau_1 \to \tau_2$, where τ_1 and τ_2 are both Pirouette types.

$$
\begin{array}{lrcl}
\text{Choreography Types} & \tau & ::= & \ell_1.t \mid t_1@\ell \to \tau_2 \mid \tau_1 \to \tau_2 \\
\text{Local Contexts} & \Gamma & ::= & \cdot \mid \Gamma, \ell.x : t \\
\text{Choreography Contexts} & \Delta & ::= & \cdot \mid \Delta, X : \tau
\end{array}
$$

$$
\textsc{Send}\ \frac{\Gamma|_{\ell_1} \vdash e : t_1 \qquad \Gamma, \ell_2.x : t_1;\ \Delta \vdash C : \tau_2 \qquad \ell \neq \ell_2}{\Gamma;\ \Delta \vdash \ell_1.e \rightsquigarrow \ell_2.x;\ C : \tau_2}
\qquad
\textsc{DefLocal}\ \frac{\Gamma;\ \Delta \vdash C_1 : \ell.t_1 \qquad \Gamma, \ell.x : t_1;\ \Delta \vdash C_2 : \tau_2}{\Gamma;\ \Delta \vdash \mathsf{let}\ \ell.x := C_1\ \mathsf{in}\ C_2 : \tau_2}
$$

$$
\textsc{FunLocal}\ \frac{\Gamma, \ell.x : t_1;\ \Delta, F : t_1@\ell \to \tau_2 \vdash C : \tau_2}{\Gamma;\ \Delta \vdash \mathsf{fun}\ F(\ell.x) := C : t_1@\ell \to \tau_2}
\qquad
\textsc{FunGlobal}\ \frac{\Gamma;\ \Delta, F : \tau_1 \to \tau_2, X : \tau_1 \vdash C : \tau_2}{\Gamma;\ \Delta \vdash \mathsf{fun}\ F(X) := C : \tau_1 \to \tau_2}
$$

Fig. 4. Pirouette Types (Syntax and Selected Rules)

The choreographic type judgment is of the form $\Gamma;\ \Delta \vdash C : \tau$ where Γ is a local context, Δ is a global context, C is a choreography, and τ is a choreographic type. Global contexts are normal typing contexts relating choreography variables to choreography types. However, local contexts have to relate variables to types, but must also keep track of the location of the variable. Since each location is its own namespace, we relate variables paired with locations to local types (see the syntax of Γ in Figure 4).

For any location ℓ, we can recursively *project* the typing context of the location from a local context Γ as follows:

$$
\Gamma|_{\ell_1} = \begin{cases} \cdot & \text{if } \Gamma = \cdot \\ \Gamma'|_{\ell_1}, x : t & \text{if } \Gamma = \Gamma', \ell_1.x : t \\ \Gamma'|_{\ell_1} & \text{if } \Gamma = \Gamma', \ell_2.x : t \text{ where } \ell_1 \neq \ell_2 \end{cases}
$$

Intuitively, this gives the context for the namespace ℓ.

We can see projection in action in the Send rule. Here, we check that e has the local type t at the location ℓ. We then check the remainder of the choreography under the assumption that x has type t *at* ℓ_2, since after the send x will be bound to the result of evaluating e. Note that this works because values are closed; otherwise, v might not typecheck in ℓ_2's namespace.

The rule DefLocal shows how the program $\mathsf{let}\ \ell.x := C_1\ \mathsf{in}\ C_2$ acts as an elimination rule for the type $\ell.t$. We ensure that C_1 has the type $\ell.t$, and then we bind x to t locally in C_2. Finally, FunLocal and FunGlobal produce local and global function types, respectively. Both also bind the function name to the appropriate function type, allowing for recursive function definitions.

The choreographic type system enjoys progress and preservation if the local type system is sound. However, we can get more-precise guarantees, which we call *relative progress* and *relative preservation.*

Theorem 1 (Relative Preservation). *If the local type system enjoys Preservation, then for every choreography C_1 such that $\Gamma;\ \Delta \vdash C_1 : \tau$ and $C_1 \overset{R\ B}{\Longrightarrow}_c C_2$, $\Gamma;\ \Delta \vdash C_2 : \tau$.*

Theorem 2 (Relative Progress). *If the local type system enjoys Boolean Invertability and Progress, then for every choreography C_1 such that $\cdot;\ \cdot \vdash C_1 : \tau$, either C_1 is a choreography value or there are some R, B, and C_2 such that $C_1 \overset{R\ B}{\Longrightarrow}_c C_2$.*

Corollary 1 (Relative Soundness). *If the local type system is sound, as defined in Section 2.4, then the choreographic type system enjoys progress and preservation.*

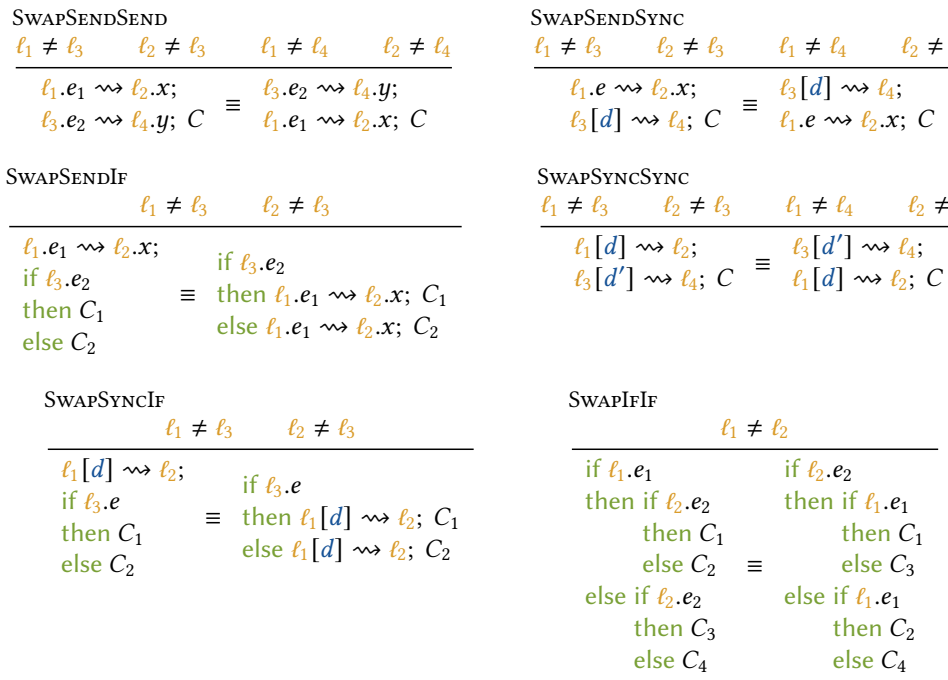

Fig. 5. Selected Choreography Equivalence Rules

By dividing up the result into more-precise theorems, we are able to get some guarantees even when the local type system is not sound. For instance, the unityped λ-calculus example (Example 5) is not sound, but does guarantee preservation. Thus, Theorem 1 allows us to lift preservation to the choreographic system. In this case, we do not get choreographic progress, intuitively because we may get stuck when trying to evaluate a local expression, or when an if expression tries to discriminate on a non-boolean value.

4 EQUATIONAL REASONING

Choreographies represent collections of programs running in parallel. In order to represent these programs serially, we are forced to decide what behavior to write first. For instance, consider the program $\ell_1.2 + 3 \rightsquigarrow \ell_2.x;\ \ell_3.3 * 4 \rightsquigarrow \ell_4.y;\ C$. The program represents ℓ_1 sending a message to ℓ_2 while ℓ_3 sends a message to ℓ_4. We could have just as well represented that program as $\ell_3.3 * 4 \rightsquigarrow \ell_4.y;\ \ell_1.2 + 3 \rightsquigarrow \ell_2.x;\ C$. Since these choreographies represent the same collection of programs, all of our constructs treat them the same way. In prior work on choreographies, this fact is typically formalized using a separate notion of *equivalence*, which says when two choreographies represent the same collection of programs [Cruz-Filipe and Montesi 2017a,b; Lanese et al. 2013; Montesi 2013]. Following that tradition, we define a notion of structural equivalence for Pirouette and study its properties.

We define choreography equivalence as the smallest equivalence relation which is also a congruence and satisfies the rules in Figure 5. The complete formal definition can be found in the accompanying technical report [Hirsch and Garg 2021].

Choreography equivalence respects types, even for open programs:

Theorem 3 (Equivalence Respects Types). *If* $\Gamma;\ \Delta \vdash C_1 : \tau$ *and* $C_1 \equiv C_2$, *then* $\Gamma;\ \Delta \vdash C_2 : \tau$

Interestingly, Theorem 3 has an outsized influence on our system model. In particular, in order to prove that the rule SwapSendIf respects types, we need to know that expressions have unique types. After all, if e_2 is given type t_1 in the true branch, but type t_2 in the false branch, there might not be a type that we could assign to e_2 that makes both branches type check. Including a typing annotation on send statements might solve this problem. However, this would require mixing the type system with the syntax of choreographies. While this is a reasonable choice in many situations, here we choose to keep them separate, since that makes it easy to tell when our results rely on the type system and when they do not.

Finally, our operational semantics respects equivalence, allowing us to prove the following simulation theorem:

Theorem 4 (Operational Semantics Simulates Equivalence). *If* $C_1 \overset{R\ B}{\Longrightarrow}_c C_2$ *and* $C_1 \equiv C_1'$, *then there is a* C_2' *such that* $C_2 \equiv C_2'$ *and* $C_1' \overset{R\ B}{\Longrightarrow}_c C_2'$.

Equivalence can be used to define a new, seemingly simpler operational semantics for choreographies. This new semantics, which we write $\Rightarrow_{\equiv}$, has no redices or block sets and it replaces internal steps from our semantics with the following rule:

$$\textsc{EquivStep}\quad \frac{C_1 \equiv C_1' \qquad C_1' \Rightarrow_{\equiv} C_2' \qquad C_2' \equiv C_2}{C_1 \Rightarrow_{\equiv} C_2}$$

We formalize this semantics in the accompanying technical report [Hirsch and Garg 2021].

In fact, much of the prior work on choreographies defined their operational semantics in precisely this way [Carbone et al. 2014; Cruz-Filipe and Montesi 2017a,b; Lanese et al. 2013; Montesi 2013]. While this new semantics is good for prior work, it is too weak for Pirouette, which allows reduction of local expressions. To see why, consider the program $\ell_1.e_1 \rightsquigarrow \ell_2.x;\ \ell_3.e_2$. It should be possible for ℓ_3 to evaluate its return value, even though ℓ_1 has not yet sent its message to ℓ_2. However, under the equivalence-based semantics, there is no way to reduce e_2, since we cannot use an equivalence to bring it up to the top.

A similar problem appears for sends. Consider the program $\ell_1.e_1 \rightsquigarrow \ell_2.x;\ \ell_3.e_2 \rightsquigarrow \ell_2.x;\ C$. In this example, both ℓ_1 and ℓ_3 are trying to send a message to ℓ_2. If e_2 can be reduced further, then ℓ_3 ought to be able to reduce e_2 while waiting for ℓ_2 to be ready to receive the second message. However, in the equivalence-based semantics, we cannot use SwapSendSend to bring up the second send, since that would swap the order of ℓ_2's receives.

Finally, consider programs that contain function applications, such as

$$\ell_1.e_1 \rightsquigarrow \ell_2.x;\\ (\text{fun } F(X) := X)\ (\ell_3.e_2 \rightsquigarrow \ell_4.y;\ \ell_4.y)$$

This program allows ℓ_1 to send a message to ℓ_2 and then applies the identity function to another choreography. Importantly, the argument choreography does not mention ℓ_1 or ℓ_2. Therefore, ℓ_3 should be able to reduce e_2 before ℓ_1 completes its send. However, it is not possible to do this in the equivalence-based definition mentioned before.

This makes reasoning with $\equiv$ much less powerful here than in previous choreographic systems. However, the difficulty is limited to the problems mentioned above.

$$
\begin{array}{lrcl}
\text{Control Expression} & E & ::= & X \mid \mathsf{fun}_{\mathsf{l}}\ F(x) \coloneqq E \mid \mathsf{fun}_{\mathsf{g}}\ F(X) \coloneqq E \mid E\ e \mid E_1\ E_2 \\
& & \mid & () \mid \mathsf{ret}(e) \mid \mathsf{let}\ \mathsf{ret}(x) \coloneqq E_1\ \mathsf{in}\ E_2 \\
& & \mid & \mathsf{send}\ e\ \mathsf{to}\ \ell;\ E \mid \mathsf{receive}\ x\ \mathsf{from}\ \ell;\ E \\
& & \mid & \mathsf{if}\ e\ \mathsf{then}\ E_1\ \mathsf{else}\ E_2 \mid \mathsf{choose}\ d\ \mathsf{for}\ \ell;\ E \\
& & \mid & \mathsf{allow}\ \ell\ \mathsf{choice} \mid \mathsf{L} \Rightarrow E_1 \mid \mathsf{R} \Rightarrow E_2 \\
\text{Systems} & \Pi & ::= & \ell_1 \triangleright E_1 \parallel \cdots \parallel \ell_n \triangleright E_n
\end{array}
$$

Fig. 6. Control Language Syntax

Theorem 5 (Weak Semantics). *Let $\overset{R\ B}{\Longrightarrow}_w$ be the relation obtained by modifying the relation $\overset{R\ B}{\Longrightarrow}_c$ as follows:*

- *$\ell.e$ can only reduce when the block set is empty,*
- *messages can only be reduced when neither the sender nor the receiver are in the block set, and*
- *in $\mathsf{let}\ \ell.x \coloneqq C_1\ \mathsf{in}\ C_2$ and function application, subchoreographies can only reduce when the block set is empty.*

(The semantics $\overset{R\ B}{\Longrightarrow}_w$ is formalized in the accompanying technical report [Hirsch and Garg 2021].) Then whenever $C_1 \overset{R\ B}{\Longrightarrow}_w C_2$, $C_1 \Rightarrow_{\equiv} C_2$. Moreover, whenever $C_1 \Rightarrow_{\equiv} C_2$, there is a redex R and choreography C_2' such that $C_1 \overset{R\ \emptyset}{\Longrightarrow}_w C_2'$ and $C_2 \equiv C_2'$.

While this makes our connection to previous work more clear, we prefer to work with the $\overset{R\ B}{\Longrightarrow}_c$ relation defined in Section 3 due to its extra power.

5 ENDPOINT PROJECTION

While Pirouette programs are designed to represent a collection of concurrently-executing programs, so far that has been a guiding intuition rather than a formal property. In order to change that, we define the *endpoint projection* operation, which *extracts a program* for each location from a choreography, if it is possible to do so. This extracted program is expressed in a language called the *control language* which features local execution and explicit constructs for message passing. The extracted programs of all locations are composed in parallel. Here, we explain the control language, then explain the extraction and finally show that the parallel composition of all extracted programs reflects and preserves the operational semantics of the choreography.

5.1 The Control Language

Our control language (Figure 6) is a concurrent λ-calculus where messages are values of local programs. It is inspired both by work on process calculi and by concurrent ML.

Like with Pirouette, control programs have two types of variable: local variables and control variables. We write control variables with capital letters, because they play a role similar to that played by choreography variables. Local variables are the variables of local programs. There are correspondingly two types of functions, local functions and global functions (written $\mathsf{fun}_{\mathsf{l}}\ F(x) \coloneqq E$ and $\mathsf{fun}_{\mathsf{g}}\ F(X) \coloneqq E$, respectively). Note that because in the control language—unlike in Pirouette—every local program is at the same location, local substitution does not take location into account.

Control programs can return the result of evaluating a local program, which we write $\mathsf{ret}(e)$. We can use the result of such a program in another program using the syntax $\mathsf{let}\ \mathsf{ret}(x) \coloneqq E_1\ \mathsf{in}\ E_2$.

However, unlike choreographies, control programs can also return the trivial value $()$. This is used for control programs that do not have a return value on them.

Communication happens between control programs composed in parallel through explicit send and receive commands. We will see later how parallel composition works, and how communication takes place.

There are two forms of branching in the control language. "If" statements are standard, and are a sequential form of branching. We also have *external choice*, which is a distributed form of branching. The program $\mathsf{allow}\ \ell\ \mathsf{choice} \mid \mathsf{L} \Rightarrow E_1 \mid \mathsf{R} \Rightarrow E_2$ represents allowing ℓ to choose which branch to take: E_1, labeled L, or E_2, labeled R. The syntax $\mathsf{choose}\ d\ \mathsf{for}\ \ell;\ E$ represents telling ℓ to take the branch labeled d.

We refer to the parallel composition of a control program for each location as a *system* (we use the symbol Π to refer to systems). The notation $\ell \triangleright E$ says that program E is running on the node ℓ. As can be seen in Figure 6, a system is a finite parallel compositions of such $\ell \triangleright E$.

We often use the syntax $\Vert_{\ell \in \mathfrak{L}} E_\ell$ to refer to a system where $\mathfrak{L}$ is a finite set of locations and $E_\_$ is a function from locations to control program expressions.

Location semantics. In defining the operational semantics of systems, two syntactic operations will be useful. The first is *system lookup* (written $\Pi(\mathsf{A})$), which refers to the control program bound to a particular location A. The second is *system update* (written $\Pi[\mathsf{A} \mapsto E]$), which replaces the program bound to A. We define them as follows:

$$\left(\Vert_{\ell \in \mathfrak{L}} E_\ell\right)(\mathsf{A}) = E_{\mathsf{A}} \qquad \left(\Vert_{\ell \in \mathfrak{L}} E_\ell\right)[\mathsf{A} \mapsto E] = \Vert_{\ell \in \mathfrak{L}} E'_\ell \text{ where } E'_\ell = \begin{cases} E & \ell = \mathsf{A} \\ E_\ell & \text{otherwise} \end{cases}$$

The semantics of control programs is given via a labeled transition system. This allows systems to match up corresponding rules in their semantics. The syntax of labels and selected rules can be found in Figure 7, and the full set of rules can be found in the accompanying technical report [Hirsch and Garg 2021].

Internal steps that do not interact with the outside are given the label ι[1]. For instance, the rule SendE takes a local step in a message to be sent to ℓ, which is an internal step. Every step of a local program corresponds to an ι step.

Sends and receives are labeled with matching labels: $v \rightsquigarrow \ell_2$ for sends, and $\ell_1 : v \rightsquigarrow$ for receives. Note that from the perspective of a single control-language program, receives are treated nondeterministically—any value could be received. Our system semantics will force sends and receives to match up. Similarly, external choice and its resolution have matching labels: $\ell : [d]$ for external choice and $[d] \rightsquigarrow \ell$ for its resolution.

In choreographies, all participants must β-reduce function applications together. For instance, consider reducing a local function:

$$(\mathsf{fun}\ F(\ell.x) := C)\ \ell.e \xrightarrow[\mathsf{c}]{\mathrm{LocalFun}(\ell.v)\ \emptyset} C[\ell|\ x \mapsto v][F \mapsto \mathsf{fun}\ F(\ell.x) := C]$$

Because the choreography steps from a choreography with a function application to one without, *every* location's control program changes. In order to accommodate this, we use a new label, ι_{sync}. The only three rules labeled with ι_{sync} are LetRet, AppLocal, and AppGlobal, all of which can be found in Figure 7.

[1]It is standard to use τ to refer to internal steps. However, τ already represents Pirouette types. We thus use ι for "internal."

$$\text{Label}\quad l \quad ::= \quad \iota \mid v \rightsquigarrow \ell \mid \ell : v \rightsquigarrow \mid [d] \rightsquigarrow \ell \mid \ell : [d] \mid \iota_{\text{sync}} \mid \text{Fun}(l) \mid \text{Arg}(l)$$

$$\textsc{SendE}\ \frac{e_1 \Rightarrow_{\text{e}} e_2}{\text{send } e_1 \text{ to } \ell;\ E \overset{\iota}{\Rightarrow}_{\text{E}} \text{send } e_2 \text{ to } \ell;\ E}$$

$$\textsc{SendV}\ \frac{\text{Value}(v)}{\text{send } v \text{ to } \ell;\ E \overset{v \rightsquigarrow \ell}{\Longrightarrow}_{\text{E}} E} \qquad \textsc{RecvV}\ \frac{\text{Value}(v)}{\text{receive } x \text{ from } \ell;\ E \overset{\ell : v \rightsquigarrow}{\Longrightarrow}_{\text{E}} E[x \mapsto v]}$$

$$\textsc{Choose}\ \frac{}{\text{choose } d \text{ for } \ell;\ E \overset{[d] \rightsquigarrow \ell}{\Longrightarrow}_{\text{E}} E} \qquad \textsc{AllowChoiceL}\ \frac{}{\text{allow } \ell \text{ choice} \mid \text{L} \Rightarrow E_1 \mid \text{R} \Rightarrow E_2 \overset{\ell : [\text{L}]}{\Longrightarrow}_{\text{E}} E_1}$$

$$\textsc{LetRet}\ \frac{\text{Value}(v)}{\text{let ret}(x) := \text{ret}(v) \text{ in } E_2 \overset{\iota_{\text{sync}}}{\Longrightarrow}_{\text{E}} E_2[x \mapsto v]} \qquad \textsc{AppLocal}\ \frac{\text{Value}(v)}{(\text{fun}_{\text{l}}\ F(x) := E)\ v \overset{\iota_{\text{sync}}}{\Longrightarrow}_{\text{E}} E[x \mapsto v]}$$

$$\textsc{AppGlobal}\ \frac{\text{Value}(V)}{(\text{fun}_{\text{g}}\ F(X) := E)\ V \overset{\iota_{\text{sync}}}{\Longrightarrow}_{\text{E}} E[X \mapsto V]}$$

Fig. 7. Control Programs Semantics (Selected Rules)

System semantics. Systems are also given semantics via a labeled transition system. The system labels arise from a merging operator on control-language labels, where $l_1 \bowtie_L l_2$ ensures that labels l_1 and l_2 match, producing an output system label L. It also ensures that, in a function application, either both steps reduce the function or both reduce its argument. The syntax of system labels, label merge operator, and system semantics can all be found in Figure 8.

The labels ι and ι_{sync} both refer to internal steps of a production. Hence, they can be merged with themselves to yield the corresponding system label. The rule Internal allows any location ℓ to take an internal step without interfering with any other location. Synchronized steps require the use of the Synchronized Internal rule, which requires that *every* location take a ι_{sync} step.

Send and receive labels must be matched together. If ℓ_1 sends v to ℓ_2, their labels merge together to a system label $\ell_1.v \longrightarrow \ell_2$ in rule MergeComm. The rule Comm then allows both ℓ_1 and ℓ_2 to take their corresponding steps together, without interfering with any other locations. The Choice rule behaves similarly, but for choice-based branching.

5.2 Merging Control Programs

Our goal is to extract a control program for every location in a choreography *compositionally*. However, compositionality is greatly complicated by if branches. To see why, consider the following:

$$\text{System Label} \quad L \quad ::= \quad \iota \mid \ell_1.v \longrightarrow \ell_2 \mid \ell_1[d] \longrightarrow \ell_2 \mid \iota_{\text{sync}}$$

Label Merge

$$\textsc{MergeIota}\ \frac{}{\iota \bowtie_\iota \iota} \qquad \textsc{MergeSync}\ \frac{}{\iota_{\text{sync}} \bowtie_{\iota_{\text{sync}}} \iota_{\text{sync}}} \qquad \textsc{MergeComm}\ \frac{}{v \rightsquigarrow \ell_2 \bowtie_{\ell_1.v \longrightarrow \ell_2} \ell_1 : v \rightsquigarrow} \qquad \textsc{MergeChoice}\ \frac{}{[d] \rightsquigarrow \ell_2 \bowtie_{\ell_1[d] \longrightarrow \ell_2} \ell_1 : [d] \rightsquigarrow}$$

$$\textsc{MergeFun}\ \frac{l_1 \bowtie_L l_2}{\mathsf{Fun}(l_1) \bowtie_L \mathsf{Fun}(l_2)} \qquad \textsc{MergeArg}\ \frac{l_1 \bowtie_L l_2}{\mathsf{Arg}(l_1) \bowtie_L \mathsf{Arg}(l_2)}$$

System Steps

$$\textsc{Internal}\ \frac{\Pi(\ell) \overset{\iota}{\Rightarrow}_{\mathrm{E}} E}{\Pi \overset{\iota}{\Rightarrow}_{\mathrm{S}} \Pi[\ell \mapsto E]} \qquad \textsc{Synchronized Internal}\ \frac{l \bowtie_{\iota_{\text{sync}}} l \qquad \forall \ell \in \mathfrak{L},\ E_\ell \overset{l}{\Rightarrow}_{\mathrm{E}} E'_\ell}{\underset{\ell \in \mathfrak{L}}{\|} E_\ell \overset{\iota_{\text{sync}}}{\Longrightarrow}_{\mathrm{S}} \underset{\ell \in \mathfrak{L}}{\|} E'_\ell}$$

$$\textsc{Comm}\ \frac{\ell_1 \neq \ell_2 \qquad l_1 \bowtie_{\ell_1.v \longrightarrow \ell_2} l_2 \qquad \Pi(\ell_1) \overset{l_1}{\Rightarrow}_{\mathrm{E}} E_1 \qquad \Pi(\ell_2) \overset{l_2}{\Rightarrow}_{\mathrm{E}} E_2}{\Pi \overset{\ell_1.v \longrightarrow \ell_2}{\Longrightarrow}_{\mathrm{S}} \Pi[\ell_1 \mapsto E_1, \ell_2 \mapsto E_2]}$$

$$\textsc{Choice}\ \frac{\ell_1 \neq \ell_2 \qquad l_1 \bowtie_{\ell_1[d] \longrightarrow \ell_2} l_2 \qquad \Pi(\ell_1) \overset{l_1}{\Rightarrow}_{\mathrm{E}} E_1 \qquad \Pi(\ell_2) \overset{l_2}{\Rightarrow}_{\mathrm{E}} E_2}{\Pi \overset{\ell_1[d] \longrightarrow \ell_2}{\Longrightarrow}_{\mathrm{S}} \Pi[\ell_1 \mapsto E_1, \ell_2 \mapsto E_2]}$$

Fig. 8. System Semantics

```
if ℓ1.e
then ℓ2.3 ⇝ ℓ1.x; ℓ1[L] ⇝ ℓ2; ℓ2.0
else ℓ2.3 ⇝ ℓ1.x; ℓ1[R] ⇝ ℓ2; ℓ2.1
```

Intuitively, we want ℓ_2 to be associated with the control program

$$\text{send } 3 \text{ to } \ell_1;\ \text{allow } \ell_1 \text{ choice} \mid \mathsf{L} \Rightarrow \text{ret}(0) \mid \mathsf{R} \Rightarrow \text{ret}(1)$$

However, when defining our procedure formally, we want to extract a program for each branch of the if expression, and combine them together to get the final program.

This leads to two issues. First, we need to be able to define a program for each branch. Second, we need to be able to merge those two programs into a single program.

To see why it is difficult to define a program for each branch, consider the true branch of the program above. We know that ℓ_1 will send a synchronization message to ℓ_2. Thus, ℓ_2 must allow ℓ_1 to make a choice for it, as we saw earlier. However, here we only have the L branch available; the R branch will not be available until we merge.

To solve this problem, we add one-branch choice constructs to our control language:

$$\begin{array}{l}\text{allow } \ell \text{ choice} \\ \mid \mathsf{L} \Rightarrow E\end{array} \qquad \text{and} \qquad \begin{array}{l}\text{allow } \ell \text{ choice} \\ \mid \mathsf{R} \Rightarrow E\end{array}$$

These act precisely like the two-choice construct, except that they only allow their one branch to be taken. With these, we now have a program for ℓ_2 we intend to extract from the branch above:

$$\text{send } 3 \text{ to } \ell_1;\ \text{allow } \ell_1 \text{ choice} \mid \mathsf{L} \Rightarrow \text{ret}(0)$$

$$\left(\begin{array}{l}\text{allow } \ell \text{ choice}\\ \mid \mathsf{L} \Rightarrow E_1\end{array}\right) \sqcup \left(\begin{array}{l}\text{allow } \ell \text{ choice}\\ \mid \mathsf{L} \Rightarrow E_2\end{array}\right) \triangleq \begin{array}{l}\text{allow } \ell \text{ choice}\\ \mid \mathsf{L} \Rightarrow E_1 \sqcup E_2\end{array}$$

$$\left(\begin{array}{l}\text{allow } \ell \text{ choice}\\ \mid \mathsf{L} \Rightarrow E_1\end{array}\right) \sqcup \left(\begin{array}{l}\text{allow } \ell \text{ choice}\\ \mid \mathsf{R} \Rightarrow E_2\end{array}\right) \triangleq \begin{array}{l}\text{allow } \ell \text{ choice}\\ \mid \mathsf{L} \Rightarrow E_1\\ \mid \mathsf{R} \Rightarrow E_2\end{array}$$

$$\left(\begin{array}{l}\text{allow } \ell \text{ choice}\\ \mid \mathsf{L} \Rightarrow E_1\end{array}\right) \sqcup \left(\begin{array}{l}\text{allow } \ell \text{ choice}\\ \mid \mathsf{L} \Rightarrow E_{2,1}\\ \mid \mathsf{R} \Rightarrow E_{2,2}\end{array}\right) \triangleq \begin{array}{l}\text{allow } \ell \text{ choice}\\ \mid \mathsf{L} \Rightarrow E_1 \sqcup E_{2,1}\\ \mid \mathsf{R} \Rightarrow E_{2,2}\end{array}$$

Fig. 9. Merge Operator Definition (Selected Parts)

We now need a way to merge the extracted programs from each branch into a single program. We define a *partial* merge operator $\sqcup$, which ensures that two programs are the same until they allow both choices. We adopt notation from computability theory and write $f(x) \uparrow$ when f is a partial function to denote that the function is undefined. So we would write $E_1 \sqcup E_2 \uparrow$ if the merge of E_1 and E_2 is undefined. We further write $f(x) \downarrow$ (so $E_1 \sqcup E_2 \downarrow$) if $f(x)$ is defined, but we do not care about the value. In Figure 9, you can find the definition of the merge operator when the left-hand side is allow ℓ choice | L $\Rightarrow E_1$. The remaining parts of the definition can be found in the accompanying technical report [Hirsch and Garg 2021].

5.3 Endpoint Projection, Defined

We are now ready to define endpoint projection, or EPP. Merging is an important part of the definition of EPP, and EPP inherits partiality from merging. We continue to use $f(x) \uparrow$ if $f(x)$ is undefined and $f(x) \downarrow$ if $f(x)$ is defined, but we do not care about the value.

Endpoint projection is defined as follows:

$$[\![\ell_1.e]\!]_{\ell_2} = \begin{cases} \mathsf{ret}(e) & \text{if } \ell_1 = \ell_2 \\ () & \text{otherwise} \end{cases} \qquad [\![X]\!]_{\ell} = X$$

$$[\![\ell_1.e \rightsquigarrow \ell_2.x;\ C]\!]_{\ell_3} = \begin{cases} \uparrow & \text{if } \ell_1 = \ell_2 = \ell_3 \\ \mathsf{send}\ e\ \mathsf{to}\ \ell_2;\ [\![C]\!]_{\ell_3} & \text{if } \ell_1 = \ell_3 \neq \ell_2 \\ \mathsf{receive}\ x\ \mathsf{from}\ \ell_1;\ [\![C]\!]_{\ell_3} & \text{if } \ell_1 \neq \ell_3 = \ell_2 \\ [\![C]\!]_{\ell_3} & \text{if } \ell_1 \neq \ell_3 \text{ and } \ell_2 \neq \ell_3 \end{cases}$$

$$[\![\mathsf{if}\ \ell_1.e\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2]\!]_{\ell_2} = \begin{cases} \mathsf{if}\ e\ \mathsf{then}\ [\![C_1]\!]_{\ell_2}\ \mathsf{else}\ [\![C_2]\!]_{\ell_2} & \text{if } \ell_1 = \ell_2 \\ [\![C_1]\!]_{\ell_2} \sqcup [\![C_2]\!]_{\ell_2} & \text{otherwise} \end{cases}$$

$$[\![\ell_1[d] \rightsquigarrow \ell_2;\ C]\!]_{\ell_3} = \begin{cases} \uparrow & \text{if } \ell_1 = \ell_2 = \ell_3 \\ \mathsf{choose}\ d\ \mathsf{for}\ \ell_2;\ [\![C]\!]_{\ell_3} & \text{if } \ell_1 = \ell_3 \neq \ell_2 \\ \mathsf{allow}\ \ell_1\ \mathsf{choice} \mid \mathsf{L} \Rightarrow [\![C]\!]_{\ell_3} & \text{if } \ell_1 \neq \ell_3 = \ell_2 \text{ and } d = \mathsf{L} \\ \mathsf{allow}\ \ell_1\ \mathsf{choice} \mid \mathsf{R} \Rightarrow [\![C]\!]_{\ell_3} & \text{if } \ell_1 \neq \ell_3 = \ell_2 \text{ and } d = \mathsf{R} \\ [\![C]\!]_{\ell_3} & \text{otherwise} \end{cases}$$

$$[\![\text{let } \ell_1.x := C_1 \text{ in } C_2]\!]_{\ell_2} = \begin{cases} \text{let ret}(x) := [\![C_1]\!]_{\ell_2} \text{ in } [\![C_2]\!]_{\ell_2} & \text{if } \ell_1 = \ell_2 \\ (\text{fun}_{\mathsf{g}}\ F(X) := [\![C_2]\!]_{\ell_2})\ [\![C_1]\!]_{\ell_2} & \text{where } F, X \text{ are fresh, otherwise} \end{cases}$$

$$[\![\text{fun } F(\ell_1.x) := C]\!]_{\ell_2} = \begin{cases} \text{fun}_{\mathsf{l}}\ F(x) := [\![C]\!]_{\ell_2} & \text{if } \ell_1 = \ell_2 \\ \text{fun}_{\mathsf{g}}\ F(X) := [\![C]\!]_{\ell_2} & \text{where } X \text{ is fresh, otherwise} \end{cases}$$

$$[\![C\ \ell_1.e]\!]_{\ell_2} = \begin{cases} [\![C]\!]_{\ell_2}\ e & \text{if } \ell_1 = \ell_2 \\ [\![C]\!]_{\ell_2}\ () & \text{otherwise} \end{cases} \qquad [\![\text{fun } F(X) := C]\!]_{\ell} = \text{fun}_{\mathsf{g}}\ F(X) := [\![C]\!]_{\ell}$$

$$[\![C_1\ C_2]\!]_{\ell} = [\![C_1]\!]_{\ell}\ [\![C_2]\!]_{\ell}$$

Note a simple design principle: local expressions owned by a location other than the one being projected to are projected to $()$. This allows us to define a control program with the same control flow, but which does not know about the precise local expressions held at other locations.

This definition tells us what the control program for a single location is. However, we are interested in a system of control programs. We can lift the single-location definition to a multi-location system definition: $[\![C]\!]_{\mathfrak{L}} = \underset{\ell \in \mathfrak{L}}{\|}\ \ell \triangleright [\![C]\!]_{\ell}$.

One way of thinking about $[\![C]\!]_{\ell}$ is that it gives ℓ's view of C. From this perspective, it makes sense to ask what ℓ's view of a step of computation is. We can provide this by projecting a choreography redex to a control-language label, as follows:

$$[\![\ell_1.(e_1 \Rightarrow e_2)]\!]_{\ell_2} = \begin{cases} \iota & \text{if } \ell_1 = \ell_2 \\ \uparrow & \text{otherwise} \end{cases} \qquad [\![\text{if } \ell_1.(e_1 \Rightarrow e_2)]\!]_{\ell_2} = \begin{cases} \iota & \text{if } \ell_1 = \ell_2 \\ \uparrow & \text{otherwise} \end{cases}$$

$$[\![\text{if } \ell_1.\text{true}]\!]_{\ell_2} = \begin{cases} \iota & \text{if } \ell_1 = \ell_2 \\ \uparrow & \text{otherwise} \end{cases} \qquad [\![\text{if } \ell_1.\text{false}]\!]_{\ell_2} = \begin{cases} \iota & \text{if } \ell_1 = \ell_2 \\ \uparrow & \text{otherwise} \end{cases}$$

$$[\![\text{LocalArg}(\ell_1.(e_1 \Rightarrow e_2))]\!]_{\ell_2} = \begin{cases} \iota & \text{if } \ell_1 = \ell_2 \\ \uparrow & \text{otherwise} \end{cases}$$

$$[\![\ell_1.(e_1 \Rightarrow e_2) \rightsquigarrow \ell_2]\!]_{\ell_3} = \begin{cases} \tau & \text{if } \ell_1 = \ell_3 \\ \uparrow & \text{otherwise} \end{cases}$$

$$[\![\ell_1.(v\ \text{value}) \rightsquigarrow \ell_2]\!]_{\ell_3} = \begin{cases} v \rightsquigarrow \ell_2 & \text{if } \ell_1 = \ell_3 \neq \ell_2 \\ \ell_1 : v \rightsquigarrow & \text{if } \ell_1 \neq \ell_3 = \ell_2 \\ \uparrow & \text{otherwise} \end{cases}$$

$$[\![\ell_1[d] \rightsquigarrow \ell_2]\!]_{\ell_3} = \begin{cases} [d] \rightsquigarrow \ell_2 & \text{if } \ell_1 = \ell_3 \neq \ell_2 \\ \ell_1 : [d] & \text{if } \ell_1 \neq \ell_3 = \ell_2 \\ \uparrow & \text{otherwise} \end{cases} \qquad [\![\text{let } \ell_1 := (v\ \text{value})]\!]_{\ell_2} = \iota_{\text{sync}}$$

$$[\![\text{LocalFun}(\ell_1.v)]\!]_{\ell_2} = \iota_{\text{sync}} \qquad [\![\text{GlobalFun}]\!]_{\ell} = \iota_{\text{sync}} \qquad [\![\text{Arg}(R)]\!]_{\ell} = \text{Arg}([\![R]\!]_{\ell})$$

$$[\![\text{Fun}(R)]\!]_{\ell} = \text{Fun}([\![R]\!]_{\ell})$$

Just as we can ask what some location's view of a step of computation is, we can ask what a system's view of a step of computation is. Interestingly, this does not rely on which locations are in the system, and it is therefore a total operation.

$$[\![\ell_1.(e_1 \Rightarrow e_2)]\!] = \iota \qquad [\![\text{if } \ell_1.(e_1 \Rightarrow e_2)]\!] = \iota \qquad [\![\text{if } \ell_1.\text{true}]\!] = \iota \qquad [\![\text{if } \ell_1.\text{false}]\!] = \iota$$

$$[\![\text{LocalArg}(\ell_1.(e_1 \Rightarrow e_2))]\!] = \iota \qquad [\![\ell_1.(e_1 \Rightarrow e_2) \rightsquigarrow \ell_2]\!] = \iota \qquad [\![\ell_1.(v \text{ value}) \rightsquigarrow \ell_2]\!] = \ell_1.v \longrightarrow \ell_2$$

$$[\![\ell_1[d] \rightsquigarrow \ell_2]\!] = \ell_1[d] \longrightarrow \ell_2 \qquad [\![\text{let } \ell_1 := (v \text{ value})]\!] = \iota_{\text{sync}} \qquad [\![\text{LocalFun}(\ell_1.v)]\!] = \iota_{\text{sync}}$$

$$[\![\text{GlobalFun}]\!] = \iota_{\text{sync}} \qquad [\![\text{Arg}(R)]\!] = [\![R]\!] \qquad [\![\text{Fun}(R)]\!] = [\![R]\!]$$

5.4 Properties of Endpoint Projection

EPP is one of the most-important operations on choreographies. It is what gives them a ground-truth interpretation as a parallel composition of programs. In fact, without EPP it would be almost impossible to state one of our most-important theorems: deadlock freedom by construction (Theorem 12 below).

The first property we examine is how EPP treats equivalence. As with every other operation on choreographies, we would like it if EPP treated equivalent choreographies the same. Note that we have no notion of equivalence on control-language programs, so we get a strong notion of "treating the same":

Theorem 6 (Equivalence Begets Equality). *If $C_1 \equiv C_2$, then $[\![C_1]\!]_\ell = [\![C_2]\!]_\ell$ for every $\ell \in \mathcal{L}$.*

Next we would like to examine the relationship between the semantics of a choreography and the semantics of its projection. However, there is still one remaining disconnect between the semantics of choreographies and that of systems that comes into play. Yet again, it has to do with the semantics of external choice. To see the issue, consider the following example:

$$C_1 \triangleq \begin{array}{l}\text{if } \ell_1.\text{true}\\ \text{then } \ell_1[\text{L}] \rightsquigarrow \ell_2;\ \ell_2.0\\ \text{else } \ell_1[\text{R}] \rightsquigarrow \ell_2;\ \ell_2.1\end{array} \qquad C_2 \triangleq \ell_1[\text{L}] \rightsquigarrow \ell_2;\ \ell_2.0 \qquad [\![C_1]\!]_{\ell_2} = \begin{array}{l}\text{allow } \ell_1 \text{ choice}\\ \mid \text{L} \Rightarrow \text{ret}(0)\\ \mid \text{R} \Rightarrow \text{ret}(1)\end{array}$$

$$[\![C_2]\!]_{\ell_2} = \begin{array}{l}\text{allow } \ell_1 \text{ choice}\\ \mid \text{L} \Rightarrow \text{ret}(0)\end{array} \qquad C_1 \xrightarrow{\text{if } \ell_1.\text{true } \emptyset}_{\text{c}} C_2$$

As you can see, by taking a choreography step which corresponds to a completely internal step on ℓ_1, we have lost information about a possible path on ℓ_2. This comes because the choice of the path is up to ℓ_1, who "makes up their mind" in that internal step.

One way to view this is from ℓ_2's point of view. In the program C_1, ℓ_2 has a nondeterministic program: a message will come in to tell them which of two branches to take. This is evident in the semantics of the control language, since $[\![C_1]\!]_{\ell_2}$ can take either of two steps. However, C_1 is deterministic from ℓ_2's point of view, since only one message is possible. Hence, this step has resolved some nondeterminism.

We formalize this notion of "the same program, but with some nondeterminism resolved" in a new relation called $\leq_{\text{nd}}$. This is *nearly* defined as the smallest relation that commutes with all of the control-language constructs and also fulfills the following extra rules:

$$\frac{E_1 \leq_{\text{nd}} E_{2,1}}{\begin{array}{l}\text{allow } \ell \text{ choice}\\ \mid \text{L} \Rightarrow E_1\end{array} \leq_{\text{nd}} \begin{array}{l}\text{allow } \ell \text{ choice}\\ \mid \text{L} \Rightarrow E_{2,1}\\ \mid \text{R} \Rightarrow E_{2,2}\end{array}} \qquad \frac{E_1 \leq_{\text{nd}} E_{2,2}}{\begin{array}{l}\text{allow } \ell \text{ choice}\\ \mid \text{R} \Rightarrow E_1\end{array} \leq_{\text{nd}} \begin{array}{l}\text{allow } \ell \text{ choice}\\ \mid \text{L} \Rightarrow E_{2,1}\\ \mid \text{R} \Rightarrow E_{2,2}\end{array}}$$

However, there is a small complication: functions are only related to themselves. The full definition can be found in the accompanying technical report [Hirsch and Garg 2021]. The relation $\leq_{\text{nd}}$ is a partial order.

We extend the $\leq_{\text{nd}}$ relation to systems pointwise, so $\Pi_1 \leq_{\text{nd}} \Pi_2$ if for every ℓ such that $\Pi_1(\ell)\downarrow$, $\Pi_2(\ell)\downarrow$ and $\Pi_1(\ell) \leq_{\text{nd}} \Pi_2(\ell)$. The following theorem relates $\leq_{\text{nd}}$ to the semantics of systems. It may look complicated, but is not. Its two bullet points respectively say: (1) If a less nondeterministic system takes a step, then that step is available to any more-nondeterministic system, and (2) If a more nondeterministic system takes a step but the less nondeterministic system is the result of projecting some choreography, then we can take advantage of the fact that choices are always paired in choreographies to mimic the step in the less nondeterministic system.

Theorem 7 (Lifting and Lowering System Steps Across $\leq_{\text{nd}}$)**.** *If $\Pi_1 \leq_{nd} \Pi_2$, then the following are both true:*

- *If $\Pi_1 \overset{L}{\Longrightarrow}_S \Pi_1'$, then there is a Π_2' such that $\Pi_1' \leq_{nd} \Pi_2'$ and $\Pi_2 \overset{L}{\Longrightarrow}_S \Pi_2'$.*
- *If $\Pi_2 \overset{L}{\Longrightarrow}_S \Pi_2'$ and $\Pi_1 = [\![C]\!]_{\mathfrak{L}}$, then there is a Π_1' such that $\Pi_1' \leq_{nd} \Pi_2'$ and $\Pi_1 \overset{L}{\Longrightarrow}_S \Pi_1'$.*

Finally, the less-nondeterminism relation allows us to connect the semantics of choreographies with the semantics of our control language. Steps in the choreographies correspond to actions by one or more locations. If a step involves a location, then that location's control program also takes a step; the label of that step is the projection of the redex of the choreographic step. However, if a step does not involve a location, then that location's control program does not take a step and the projection of the redex for that location is undefined. While an uninvolved location does not take a step, it may find its nondeterminism reduced as other locations "make up their minds" about what branch it should take in the future.

Theorem 8 (Local Completeness)**.** *If $C_1 \overset{R\ B}{\Longrightarrow}_c C_2$, then for any location ℓ, either (a) $[\![C_1]\!]_\ell \overset{[\![R]\!]_\ell}{\Longrightarrow}_E [\![C_2]\!]_\ell$, or (b) $[\![R]\!]_\ell \uparrow$ and $[\![C_2]\!]_\ell \leq_{nd} [\![C_1]\!]_\ell$.*

Theorem 9 (Global Completeness)**.** *If $C_1 \overset{R\ B}{\Longrightarrow}_c C_2$ and every location named in R is in $\mathfrak{L}$, then there is a Π such that $[\![C_1]\!]_{\mathfrak{L}} \overset{[\![R]\!]}{\Longrightarrow}_S \Pi$ and $[\![C_2]\!]_{\mathfrak{L}} \leq_{nd} \Pi$.*

The requirement that every location named in R is in $\mathfrak{L}$ is important: if R represents ℓ_1 sending a message to ℓ_2, but ℓ_2 is not in $\mathfrak{L}$, then there is no one in $[\![C]\!]_{\mathfrak{L}}$ to do the receiving. This blocks ℓ_1 from sending its message (since message passing is synchronous) and thus blocks the system step from taking place. However, if both ℓ_1 and ℓ_2 are in $\mathfrak{L}$, then Theorem 8 tells us that ℓ_2 is guaranteed to be ready to receive ℓ_1's message.

A similar difficulty comes when trying to go in the other direction: it is not enough to know that ℓ_1's projection sends a message for a choreography to take a step; we must also know that ℓ_2's projection receives it. This means that we have to state local soundness differently depending on the label of the control-program step:

Theorem 10 (Local Soundness)**.** *All of the following are true:*

- *If $[\![C_1]\!]_\ell \overset{l}{\Rightarrow}_E E$ where $l \bowtie_\iota l$, then there are R and C_2 such that $[\![R]\!]_\ell = l$, $[\![C_2]\!]_\ell = E$, and $C_1 \overset{R\ \emptyset}{\Longrightarrow}_c C_2$.*

- *If* $[\![C_1]\!]_{\ell_1} \overset{l_1}{\Longrightarrow}_E E_1$ *and* $[\![C_2]\!]_{\ell_2} \overset{l_2}{\Longrightarrow}_E E_2$ *where* $l_1 \bowtie_{\ell_1.v \longrightarrow \ell_2} l_2$, *then there are* R *and* C_2 *such that (a)* $[\![C_2]\!]_{\ell_1} = E_1$, *(b)* $[\![C_2]\!]_{\ell_2} = E_2$, *(c)* $[\![R]\!]_{\ell_1} = l_1$, *(d)* $[\![R]\!]_{\ell_2} = l_2$, *and (e)* $C_1 \overset{R\ \emptyset}{\Longrightarrow}_c C_2$.
- *If* $[\![C_1]\!]_{\ell_1} \overset{l_1}{\Longrightarrow}_E E_1$ *and* $[\![C_2]\!]_{\ell_2} \overset{l_2}{\Longrightarrow}_E E_2$ *where* $l_1 \bowtie_{\ell_1[d] \longrightarrow \ell_2} l_2$, *then there are* R *and* C_2 *such that (a)* $[\![C_2]\!]_{\ell_1} = E_1$, *(b)* $[\![C_2]\!]_{\ell_2} = E_2$, *(c)* $[\![R]\!]_{\ell_1} = l_1$, *(d)* $[\![R]\!]_{\ell_2} = l_2$, *and (e)* $C_1 \overset{R\ \emptyset}{\Longrightarrow}_c C_2$.
- *If* $\mathsf{LN}(C_1) \subseteq \mathfrak{L} \neq \emptyset$, $l \bowtie_{\iota_{sync}} l$, *and for every* $\ell \in \mathfrak{L}$ $[\![C_1]\!]_\ell \overset{l}{\Longrightarrow}_E E_\ell$, *then there are* R *and* C_2 *such that (a) for every* $\ell \in \mathfrak{L}$, $[\![R]\!]_\ell = l_\ell$, *and (b)* $C_1 \overset{R\ \emptyset}{\Longrightarrow}_c C_2$

The requirement that $\mathsf{LN}(C_1) \subseteq \mathfrak{L} \neq \emptyset$ may seem unusual in the last case of Theorem 10. In order to β-reduce a function call or a subexpression $\mathsf{let}\ \ell.x := C_1\ \mathsf{in}\ C_2$ inside a choreography C, every location in C must be able to perform the same β-reduction. However, if we only require that every location in $\mathsf{LN}(C)$ be able to make a step, the requirement might be trivial if C does not name any locations. Therefore, we require that every location in some nonempty set of locations $\mathfrak{L}$ which contains every location in $\mathsf{LN}(C)$ be able to make a step. In the common case where $\mathsf{LN}(C)$ is nonempty, this restriction is the same as the simpler requirement.

Lifting soundness from control programs to systems yields a much simpler theorem. However, note how the strange requirement for β-reduction steps infects the theorem:

Theorem 11 (Global Soundness). *If* $LN(C_1) \subseteq \mathfrak{L} \neq \emptyset$ *and* $[\![C_1]\!]_{\mathfrak{L}} \overset{L}{\Longrightarrow}_S \Pi$ *then there is an* R *and* C_2 *such that (a)* $[\![C_2]\!]_{\mathfrak{L}} \downarrow$, *(b)* $[\![C_2]\!]_{\mathfrak{L}} \leq_{nd} \Pi$, *(c)* $[\![R]\!] = L$, *and (d)* $C_1 \overset{R\ \emptyset}{\Longrightarrow}_c C_2$.

We use these to develop the most important theorem for Pirouette (and choreographies in general): deadlock freedom by design. Interestingly, the proof is a simple interplay of soundness and completeness for our translation along with type soundness.

Theorem 12 (Deadlock Freedom By Design). *If choreography typing enjoys both progress and preservation (e.g., if local typing is sound),* $\cdot;\ \cdot \vdash C_1 : \tau$, *and* $[\![C_1]\!]_{\mathfrak{L}} \overset{Ls^*}{\Longrightarrow}_S \Pi$, *then either every location in* Π *maps to a control-language value, or there are* L *and* Π' *such that* $\Pi \overset{L}{\Longrightarrow}_S \Pi'$.

Sketch of the mechanized proof. By soundness, there are a list of redices Rs and a choreography C_2 such that $C_1 \overset{Rs\ \emptyset}{\Longrightarrow}_c C_2$ and $[\![C_2]\!]_{\mathfrak{L}} \leq_{\mathrm{nd}} \Pi$. By preservation, $\cdot;\ \cdot \vdash C_2 : \tau$, and by progress either C_2 is a choreography value or there are R and C_3 such that $C_2 \overset{R\ \emptyset}{\Longrightarrow}_c C_3$. If C_2 is a value, then every location in Π maps to a control-language value, and we are done. Otherwise, completeness tells us there is a Π'' such that $[\![C_2]\!]_{\mathfrak{L}} \overset{[\![R]\!]}{\Longrightarrow}_S \Pi''$. We can then use Theorem 7 to get a Π' such that $\Pi \overset{[\![R]\!]}{\Longrightarrow}_S \Pi'$, as desired. □

6 NOTES ON THE COQ CODE

In this paper, we have presented all of our work in a standard, named style. However, in our Coq code we use a nameless style with de Bruijn indices. While normally this would not pose any difficulties, we have taken the unusual step of treating the local language generically. Here, we give the nameless version of our requirements on the local language, ensuring that the transition to the named style does not create undue confusion.

First, we require that expressions have decidable *syntactic* equality—usually a trivial requirement, since programs are usually sentences from a context-free grammar. Next, we formalize the fact that variables must be expressions by requiring a mathematical function var from natural numbers to expressions. In all of our examples, this is a terminal in the language, but this is not required. Instead of the ability to compute the set of free variables of an expression, we require a predicate $\text{Closed}_n(e)$, which means that there are no free variables above n in e. We require that $\text{Closed}_n(\text{var}(m))$ if and only if $m < n$, and we write $\text{Closed}(e)$ for $\text{Closed}_0(e)$.

Substitution is changed to allow for infinite parallel substitution. Formally, we require an operation $e[\sigma]$ where σ is a mathematical function from natural numbers to expressions. This must obey the following equations:

- $\text{var}(n)[\sigma] = \sigma(n)$
- $e[n \mapsto \text{var}(n)] = e$
- $(e[\sigma_1])[\sigma_2] = e[n \mapsto (\sigma_1(n)[\sigma_2])]$
- if $\sigma_1(n) = \sigma_2(n)$ for every natural number n, then $e[\sigma_1] = e[\sigma_2]$

The last requirement is important because Coq is an *intensional* type theory, so functions that behave the same on all inputs are not necessarily equal. The other three equations are nameless versions of the equations in Section 2.

Finally, we require an additional *renaming* operation, which we write $e\langle\xi\rangle$, where ξ is a mathematical function from natural numbers to natural numbers. This must satisfy the equation $e\langle\xi\rangle = e[n \mapsto \text{var}(\xi(n))]$. While this equation could serve as a definition of renaming, it is often useful to define and reason about it separately. For instance, we can use renaming to define exchange and weakening in typing.

Typing judgments in the nameless setting use contexts which are mathematical functions from natural numbers to types. The requirements in Figure 1 are then equivalent to the following requirements:

$$\textsc{Var}\ \frac{}{\Gamma \vdash \text{var}(n) : \Gamma(n)} \qquad \textsc{Extensionality}\ \frac{\forall n, \Gamma(n) = \Delta(n) \qquad \Gamma \vdash e : t}{\Delta \vdash e : t} \qquad \textsc{Weakening}\ \frac{\Gamma \vdash e : t}{\Gamma \circ \xi \vdash e\langle\xi\rangle : t}$$

$$\textsc{Strengthening}\ \frac{\text{Closed}_n(e) \qquad \forall m < n, \Gamma(m) = \Delta(m) \qquad \Gamma \vdash e : t}{\Delta \vdash e : t} \qquad \textsc{Substitution}\ \frac{\forall n, \Gamma \vdash \sigma(n) : \Delta(n) \qquad \Delta \vdash e : t}{\Gamma \vdash e[\sigma] : t}$$

Note that both Exchange and Weakening are absorbed into the more-general Weakening rule. Moreover, we made Substitution infinitary. The Var and Strengthening rules are straightforward transformations of the named versions in Figure 1.

7 RELATED WORK

7.1 Choreographies

Choreographies originate as a way of writing web services. The W3C released a report on choreographies as a way to write web services [(W3C) 2004], which originated the idea of endpoint projection [Zongyan et al. 2007] (see Cruz-Filipe and Montesi [2017a] for the historical note). Soon, there was work formalizing and understanding choreographies from this point of view.

Choreographic programming as a paradigm originates with Montesi's Ph.D. thesis [Montesi 2013]. There, he develops the first choreographic programming language and begins exploring the formal properties thereof. Later works extend this idea, with Cruz-Filipe and Montesi [2017a] creating a core calculus of choreographies and Carbone and Montesi [2013] combining choreographies

with session types to ensure that endpoint projection never fails. However these works were all lower-order, used equivalence-based semantics, and had no mechanized metatheory.

More closely related with this project, Cruz-Filipe and Montesi [2017b] create a choreographic language with procedures and the ability to make procedure calls. While parts of our endpoint projection definition are inspired by this work, procedural choreographies remain resolutely lower-order. Procedures are global and shared, but cannot be treated as data; moreover, they cannot take other choreographies as inputs nor return them as outputs. However, unlike our functions, these procedures are able to take locations as inputs. This is harder in our setting because locations are part of types; we consider polymorphism (including location polymorphism) to be future work.

Some work has been done on higher-order choreographies. The Choral Project [Giallorenzo et al. 2020] builds choreographies on top of object-oriented programming. However, Choral has never been formalized, and so the theoretical underpinnings of higher-order choreographies were left unexplored.

Significantly, concurrent and completely independent work on functional choreographies has recently been undertaken by Cruz-Filipe et al. [2021a]. This work resembles Pirouette in several ways, but has a significantly different design. Most importantly, they identify the local language with the choreography language. This design choice allows functions to be sent as messages, but they require that messages only mention a single location, preventing functions sent as messages from including communication. It also leads to a simpler system, but makes it much harder for them to identify the core features of the local language which allow for type soundness of the choreographic language. Moreover, their operational semantics allows for functions to be reduced; this allows them to not need as many synchronizations as Pirouette projection requires. However, it violates the common requirement of call-by-value languages that functions be treated as values and makes execution more unpredictable. We preferred to keep the simpler and more-traditional operational semantics. Finally, they do not have any mechanized formalization of their system.

Mechanization. Two very recent papers have mechanized the metatheory of core choreographies, but both are lower-order. Cruz-Filipe et al. [2021c] use Coq to formalize the core calculus of choreographies developed by Cruz-Filipe and Montesi [2017a], modified so that the language of local computations is a parameter. The focus is on proving that the language is Turing-complete. In the meantime, Cruz-Filipe et al. [2021b] extend that work to verify (in Coq) endpoint projection and deadlock freedom of the same language.

Interestingly, both these papers formalize out-of-order execution for choreographies using a labeled-transition system, similar to ours. As far as we are aware, they, along with a forthcoming book on choreographies [Montesi 2020] were the first to do so, though our work was formalized before we discovered their concurrent work. Their work takes inspiration from Honda et al. [2016], who use a similar labeled-transition system semantics for multiparty session types. Thus, they do not use any construct similar to our block sets. This means that they cannot allow locations to locally reduce their messages when the receiver of that message is blocked. While for their language this does not matter (since messages do not reduce at all), in our setting this would be problematic.

7.2 Functional Concurrent Programming

There is a long tradition of mixing functional and concurrent programming in principled ways. In practice, this often leads to languages that look a lot like our control language, including the concurrency features in Racket [The Racket Team 2021]. The first academic language with channels and communication in a functional language was Facile [Giacalone et al. 1989]. Even before then,

work on parallelizing compilers for functional programming was popular. Burton [1987] considered adding annotations to functions describing when arguments should be evaluated at what location.

Currently, the most related academic project might be Links [Cooper et al. 2006] and its core language, the RPC calculus [Cooper and Wadler 2009]. Links also provides for a mixture of higher-order typed functional programming and concurrency. However, in Links, communication always happens at function boundaries, unlike Pirouette. Moreover, the RPC calculus only allows for one thread of execution, even when multiple unrelated locations can profitably be taking actions. This is in contradiction to the work on choreographies—and Pirouette in particular—where unrelated locations can compute and even pass messages concurrently.

Another language which mixes functional and concurrent programming is Murphy VII et al.'s ML5 [Licata and Harper 2010; Murphy VII et al. 2007]. ML5 has a type system for communication based on the Kripke semantics of modal logic. Interestingly, our type system also takes inspiration from modal logic: our type system can be seen as (the Curry-Howard analogue of) a degenerate form of the *proof system* for modal logic. Their use of the Kripke semantics makes sense, however, because different worlds can be viewed as different machines. Unfortunately, this led to difficulties in shipping some data that could be treated as code, so ML5 focuses on annotating *mobile code*, which includes static types like strings and integers, but not functions or local resources like arrays. This restriction is necessary due to their use of local state. Since Pirouette has no state, is has no such constraint.

REFERENCES

F.W. Burton. 1987. Functional Programming for Concurrent and Distributed Programming. *Comput. J.* 30, 5 (1987), 437–450. https://doi.org/10.1093/comjnl/30.5.437

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2007. A Calculus of Global Interaction based on Session Types. In *Developments in Computational Model (DCM)*. https://doi.org/10.1016/j.entcs.2006.12.041

Marco Carbone and Fabrizio Montesi. 2013. Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/2429069.2429101

Marco Carbone, Fabrizio Montesi, and Carsten Schürmann. 2014. Choreographies, Logically. In *Concurrency Theory (CONCUR)*. https://doi.org/10.1007/978-3-662-44584-6_5

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2006. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects (FMCO)*. https://doi.org/10.1007/978-3-540-74792-5_12

Ezra E.K. Cooper and Philip Wadler. 2009. The RPC Calculus. In *Principles and Practice of Declarative Programming (PPDP)*. https://doi.org/10.1145/1599410.1599439

Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. 2021a. Choreographies as Functions. https://arxiv.org/abs/2111.03701 In Preperation for ECOOP.

Luís Cruz-Filipe and Fabrizio Montesi. 2017a. A Core Model for Choreographic Programming. In *Formal Aspects of Component Software (FACS)*. https://doi.org/10.1007/978-3-319-57666-4_3

Luís Cruz-Filipe and Fabrizio Montesi. 2017b. Procedural Choreographic Programming. In *Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*. https://doi.org/10.1007/978-3-319-60225-7_7

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2019. Choreographies in Coq. In *Types for Proofs and Programs (TYPES)*. http://www.ii.uib.no/~bezem/abstracts/TYPES_2019_paper_27

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021b. Certifying Choreography Compilation. In *International Colloquium on Theoretical Aspects of Computing (ICTAC)*.

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2021c. Formalizing a Turing-Complete Choreographic Language in Coq. In *Interactive Theorem Proving (ITP)*.

Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. 2015. Dynamic Choreographies: Safe Runtime Updates of Distributed Applications. In *Coordination Models and Languages (COORDINATION)*. https://doi.org/10.1007/978-3-319-19282-6_5

Ornela Dardha, Elena Giachino, and Davide Sangiorgi. 2012. Session Types Revisited. In *Principles and Practice of Declarative Programming (PPDP)*. https://doi.org/10.1145/2370776.2370794

Henry DeYoung, Luís Caires, Frank Pfenning, and Bernardo Toninho. 2012. Cut Reduction in Linear Logic as Asynchronous Session-Typed Communication. In *Computer Science Logic (CSL)*. https://doi.org/10.4230/LIPIcs.CSL.2012.228

Alessandro Giacalone, Prateek Mishra, and Sanjiva Prasad. 1989. Facile: A Symmetric Integration of Concurrent and Functional Programming. *International Journal of Parallel Programming* 18 (1989), 121–160. https://doi.org/10.1007/BF01491213

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

Andrew K. Hirsch and Deepak Garg. 2021. *Pirouette: Higher-Order Typed Functional Choreographies (Technical Report)*. Technical Report. MPI-SWS. https://www.mpi-sws.org/tr/2021-004.pdf

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

Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. 2013. Amending Choreographies. In *Workshop on Automated Specification and Verification of Web Systems (WWV)*. https://doi.org/10.4204/EPTCS.123.5

Daniel R. Licata and Robert Harper. 2010. A Monadic Formalization of ML5. In *Logical Frameworks and Meta-Languages: Theory and Practice (LFMTP)*. https://doi.org/10.4204/EPTCS.34.7

Fabrizio Montesi. 2013. *Choreographic Programming*. Ph.D. Dissertation. IT University of Copenhagen. https://www.fabriziomontesi.com/files/choreographic_programming.pdf

Fabrizio Montesi. 2020. *Introduction to Choreographies*. Cambridge University Press. Accepted for Publication.

Tom Murphy VII, Karl Crary, and Robert Harper. 2007. Type-safe Distributed Programming with ML5. In *Trustworthy Global Computer (TGC)*. https://doi.org/10.1007/978-3-540-78663-4_9

Alceste Scalas and Nobuko Yoshida. 2019. Less is more: Multiparty Session Types Revisited. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3291638

The Racket Team. 2021. Racket Documentation: Concurrency and Synchronization. https://docs.racket-lang.org/guide/concurrency.html Accessed July 3, 2021.

Bernardo Toninho, Luis Caires, and Frank Pfenning. 2012. Functions as Session-Typed Processes. In *Foundations of Software Science and Computational Structures (FoSSaCS)*. https://doi.org/10.1007/978-3-642-28729-9_23

The World Wide Web Consortium (W3C). 2004. WS Choreography Model Overview. https://www.w3.org/TR/ws-chor-model/ Accessed January 29,2021.

Philip Wadler. 2012. Propositions as Sessions. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/2364527.2364568

Qiu Zongyan, Zhao Xiangpeng, Cai Chao, and Yang Hongli. 2007. Towards the Theoretical Foundation of Choreography. In *The Web Conference (WWW)*. https://doi.org/10.1145/1242572.1242704