Choreographic Quick Changes: First-Class Location (Set) Polymorphism

ASHLEY SAMUELSON, University of Wisconsin–Madison, USA
ANDREW K. HIRSCH, University at Buffalo, SUNY, USA
ETHAN CECCHETTI, University of Wisconsin–Madison, USA

Choreographic programming is a promising new paradigm for programming concurrent systems where a developer writes a single centralized program that compiles to individual programs for each node. Existing choreographic languages, however, lack critical features integral to modern systems, like the ability of one node to dynamically compute who should perform a computation and send that decision to others. This work addresses this gap with λ_{QC}, the first typed choreographic language with *first class process names* and polymorphism over both types and (sets of) locations. λ_{QC} also improves expressive power over previous work by supporting algebraic and recursive data types as well as multiply-located values. We formalize and mechanically verify our results in Rocq, including the standard choreographic guarantee of deadlock freedom.

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

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

1 Introduction

Concurrent programs are integral to many modern software systems, but programming them correctly is notoriously difficult. Traditionally, each process in the system runs a separate program, but developers must reason about the interactions between these programs and the order in which these events occur. This complex behavior can easily lead to bugs such as deadlocks, where execution stalls due to nodes with mismatched send and receive expectations forever waiting on each other.

Choreographic programming [Montesi 2013, 2023] is an emerging paradigm that promises to simplify development of correct concurrent systems. A choreography is a single top-level program that describes the computation performed by every node, including the interactions between them. A compiler then *projects* programs for individual nodes from this single top-level program. Choreographies centralize code, putting global control flow in one place and leading to *deadlock-freedom by design*, structurally eliminating a notoriously challenging issue in concurrent code.

The theory of choreographies has advanced rapidly in the last several years with the addition of higher-order functions [Cruz-Filipe et al. 2022; Hirsch and Garg 2022], polymorphism over types and processes [Graversen et al. 2024], and multiply-located values [Bates et al. 2025]. However, none of these results allow for *dynamic computation and sending* of process names in a *type-safe manner*. The ability to treat host names as first-class values and share them between nodes is critical to many practical applications, such as dynamic load balancers. The only existing work supporting first-class host names [Sweet et al. 2023] entirely lacks a type system, and consequently lacks critical type-safety properties.

This paper presents the choreographic Quick Change calculus, λ_{QC}, the first choreographic language to support first-class process names and types. To understand the value of these features, and choreographic programming in general, consider a simple cloud computing example where a client C wishes to outsource expensive computation F on input X. If C wishes to run F on a specific (statically known) worker W, they can do so using the following choreography. Here t@C

Authors' Contact Information: Ashley Samuelson, University of Wisconsin–Madison, Madison, Wisconsin, USA, ashley.samuelson@wisc.edu; Andrew K. Hirsch, University at Buffalo, SUNY, Buffalo, New York, USA, akhirsch@buffalo.edu; Ethan Cecchetti, University of Wisconsin–Madison, Madison, Wisconsin, USA, cecchetti@wisc.edu.

indicates a value of type t located at C.

$$\begin{aligned}
&\mathsf{runAtW} : (t \to t')@\mathsf{C} \to t@\mathsf{C} \to t'@\mathsf{C}\\
&\mathsf{runAtW}\ F\ X = \mathsf{let}\ \mathsf{W}.f \coloneqq F \rightsquigarrow \mathsf{W}\\
&\qquad\qquad\qquad\qquad \mathsf{W}.x \coloneqq X \rightsquigarrow \mathsf{W}\\
&\qquad\qquad\qquad \mathsf{in}\ \mathsf{W}.(f\ x) \rightsquigarrow \mathsf{C}
\end{aligned}$$

The notation $F \rightsquigarrow \mathsf{W}$ means that whoever owns F (in this case C) should send it to W. So in this program, C sends the code (F) and the input (X) to W, who locally binds them to variables f and x, respectively. Then W computes $f\ x$ and sends the result back to C.

Real systems, however, typically include a pool of workers and a load balancer to manage task assignment. Clients contact this pool manager, who then (a) selects a worker, (b) notifies the worker of their new client, and (c) sends the client the worker's identity. To implement such a thread pool in a choreography, the client C can ask a pool manager M where to run the task, and M must be able to reply with a dynamically chosen identity. A choreography for such a process might look as follows, where acquireWorker and releaseWorker are local operations by the pool manager M to locally track the state of the thread pool and select and return workers to the pool, respectively.

$$\begin{aligned}
&\mathsf{runWithWorker} : (t \to t')@\mathsf{C} \to t@\mathsf{C} \to t'@\mathsf{C}\\
&\mathsf{runWithWorker}\ F\ X = \mathsf{let}\ W \coloneqq \mathsf{M}.\mathsf{acquireWorker}() \rightsquigarrow \{\mathsf{C}\} \cup \mathsf{pool}\\
&\qquad\qquad\qquad\qquad\qquad W.f \coloneqq F \rightsquigarrow W\\
&\qquad\qquad\qquad\qquad\qquad W.x \coloneqq X \rightsquigarrow W\\
&\qquad\qquad\qquad\qquad\qquad \mathsf{C}.res \coloneqq W.(f\ x) \rightsquigarrow \mathsf{C}\\
&\qquad\qquad\qquad\qquad \mathsf{in}\ W.\text{“done”} \rightsquigarrow \mathsf{M}\ ;\ \mathsf{M}.(\mathsf{releaseWorker}\ w)\ ;\ \mathsf{C}.res
\end{aligned}$$

On the first line of this program, the pool manager M selects an idle worker from the pool and sends this *dynamically computed* value to the client C, where it is bound to the variable W. Additionally, workers in the pool are all notified about which worker was chosen to prevent situations where a worker is not aware that they have been selected.

As in the earlier runAtW function, the client then sends F and X to W, who binds them to f and x, computes $f\ x$ locally, and sends the result to C. Finally, the selected worker notifies the manager that they have finished the job, the manager releases the worker back into the thread pool, and the computation finishes by yielding the result.

The first line of this program has very similar syntax to sending and receiving local data, but is critically different: it sends a location name which is then bound to the variable W known to all parties in $\{\mathsf{C}\} \cup$ pool and allows the programmer to use it as a location name within its scope. This is the core feature of λQC: enabling *first-class location names* which can be dynamically computed (and hence quickly changed) at runtime. While this feature appears simple on its face, it hides several important complexities. First, the value is known to a set of hosts, not just a single host, making it multiply-located. Second, location names in choreographies are generally part of the language of *types*, not values. To support first-class location names without the need for dependent types, we carefully separate locations from other values in the λQC type system, while still allowing interaction between the two. These key insights allow us to intermingle dynamically chosen host names with process polymorphism, such as in the example above.

The main contributions of this work are as follows.

- We generalize the constraints on the local (message) language of *Pirouette* [Hirsch and Garg 2022] to allow for polymorphism, vastly increasing the expressivity of local computations (Section 3).

- We present λ_{QC}, the first typed choreographic programming language with the ability to send and receive first-class location names and types as messages. Our language includes polymorphism over types, processes, and sets of processes, algebraic and recursive data types, and multiply-located values. We formulate a sound, System F-like type system for our language without relying on dependent types or an operational semantics for types (Section 4).
- We define a network language (Section 5) which serves as the compilation target for our endpoint projection procedure. We show that compilation is complete, and is sound when all executed local computations are terminating. This result, along with the soundness of our type system, allows us to prove that projected systems are always deadlock-free (Section 6).
- We formalize and verify all results in the Rocq Prover (formerly Coq). This is the first mechanized formalization both of process (set) polymorphism and of multiply-located values (Section 7).

2 Background

To understand the contributions of this work, it is helpful to understand some background on choreographies with higher-order functions, process polymorphism, and multiply-located values.

2.1 Pirouette

We generalize and build on Pirouette [Hirsch and Garg 2022], a higher-order choreographic programming language. As with other choreographies, Pirouette uses a located syntax inspired by the "Alice and Bob" syntax of cryptographic protocols. To specify that A should (locally) compute $2 + 3$ and send the result to B, one would write $\mathsf{A}.(2 + 3) \rightsquigarrow \mathsf{B}$. For clarity, we use sans-serif for these source-level operations and color choreographic operations in blue, local operations in green, and location constants in red.

One core aspect of Pirouette's design that we inherit is a clear separation between choreographic operations and local operations. The choreographic operations are fixed by Pirouette, and are agnostic to the local language. The local operations can be specified in nearly any language whose values—possibly including functions—can be communicated between nodes via message-passing. The local language need only have an operational semantics, a type system, and a type that can act as a boolean (every value of that type can be used as true or false).

To connect the choreographic and local languages, Pirouette offers a form of let-expression that binds local variables to the output of a choreography. For instance, the result of $\mathsf{A}.(2 + 3) \rightsquigarrow \mathsf{B}$ is a choreographic value located at B, so using it in B's local computation requires binding a *local* variable with the output of a *choreographic* computation. The following program demonstrates how B can perform this binding and use the result locally as the variable x.

$$\mathsf{let}\ \mathsf{B}.x \coloneqq (\mathsf{A}.(2 + 3) \rightsquigarrow \mathsf{B})\ \mathsf{in}\ \mathsf{B}.(4 * x)$$

Pirouette also supports conditionals $\mathsf{if}\ C\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$ where the choreography itself branches based on the result of C, which should yield a boolean at some location ℓ. However, only ℓ can access the branch condition, so others with differing behavior in the $\mathbf{L}$(eft) and $\mathbf{R}$(ight) branches cannot proceed without knowing which to take. The location ℓ can communicate the choice $d \in \{\mathbf{L}, \mathbf{R}\}$ of branch to another location ℓ' using a *selection statement* $\ell[d] \rightsquigarrow \ell'\ ;\ C$.

At the choreographic level, Pirouette is simply typed, containing only base types and function types. The base types are of the form $t@\ell$, and describe a value of local type t—any type from the local language's type system—located at ℓ. This mixing of the local and choreographic type systems and the local variables bound by let-expressions, shown above, forces the type system to separately track both choreographic and local variables.

Pirouette defines two separate semantics: one directly at the choreographic level to allow developers to more easily reason about the behavior of the choreography, and one using a translation called *endpoint projection* (EPP) that defines how to compile a choreography to separate programs for each location. Despite operating directly on a choreography, the first semantics also captures concurrent interleavings by allowing out-of-order execution as long as it does not reorder the operations for any individual location. For example, consider the program shown below.

$$\mathsf{A}.(2 + 3) \rightsquigarrow \mathsf{B} \,;\, \mathsf{C}.(5 * 4) \rightsquigarrow \mathsf{B}$$

The local computations for A and C involve disjoint parties, so they can execute in either order. The sends, however, both involve B and must occur in the order specified: B must receive 5 from A before receiving 20 from C.

EPP defines how to compile a choreography into separate programs for each location that include only the operations that location needs to run. The location being projected to is denoted by a subscript to the projection operator, as in $[\![C]\!]_{\mathsf{A}}$ and $[\![C]\!]_{\mathsf{B}}$. For instance, sending a value from A to B requires A to compute the value and send it, and requires B to receive from A:

$$[\![\mathsf{A}.(2 + 3) \rightsquigarrow \mathsf{B}]\!]_{\mathsf{A}} = \texttt{send ret}(2 + 3)\ \texttt{to B} \qquad [\![\mathsf{A}.(2 + 3) \rightsquigarrow \mathsf{B}]\!]_{\mathsf{B}} = \texttt{recv from A}$$

We write operations of our target (network) language in `orange teletype` font.

Hirsch and Garg [2022] prove that running these projected programs in parallel produces the same result as the first semantics, meaning that developers can safely reason using the top-level semantics and, critically, any projected choreography is deadlock-free, meaning it cannot get stuck with locations waiting to receive messages from each other (or for any other reason).

2.2 Process Polymorphism

While this work is the first to support *first-class* locations in a typed choreography, PolyChorλ [Graversen et al. 2024] introduced the notion of process polymorphism, providing a way to abstract over the participants in a choreography. PolyChorλ introduces a *process abstraction* $\Lambda\ell.\,C$, akin to a classic type abstraction, that binds variable ℓ representing a process name in choreography C. In our syntax, a programmer could write the following process function in which A computes a value and sends it to a yet to be determined recipient ℓ.

$$F = \Lambda\ell.\,\mathsf{A}.(2 + 3) \rightsquigarrow \ell$$

In PolyChorλ, the only way to use a process abstraction is to apply it to a location (e.g., F B or F C) or a location variable created by another process abstraction (e.g., $F\ \ell$). Location names are not first-class values that can be computed locally or sent between parties.

Despite process names being statically resolvable in PolyChorλ, the projection of a process abstraction requires locations to behave differently depending on what the variable ℓ resolves to *at run time*. This is accomplished using an "AmI" construct in the compiled language that produces a local branch and reflects the fact that each process should know its own identity. Specifically, the program `AmI A then` E_1 `else` E_2 will execute E_1 when running at A and E_2 when running elsewhere. EPP can then project a process abstraction to an `AmI` statement where each branch is a different projection of C: `then` replaces ℓ with the current process, while `else` assumes ℓ has resolved to a different name. That is,

$$[\![\Lambda\ell.\,C]\!]_{\mathsf{A}} = \Lambda\ell.\,\texttt{AmI}\ \ell\ \texttt{then}\ [\![C[\ell \mapsto \mathsf{A}]]\!]_{\mathsf{A}}\ \texttt{else}\ [\![C]\!]_{\mathsf{A}}.$$

A feature of many polymorphic languages which is notably missing from PolyChorλ is *recursive types*. The addition of recursive types to this language would be challenging because their type system—based on System F_ω—includes an operational semantics, and as a result their endpoint projection procedure requires types which appear in the choreography to be fully-reduced type values. Implementing this feature would require either that endpoint projection handle choreographies in which nonterminating types may appear, or that recursive types are limited in order to force type computations to converge. Both of these options have significant drawbacks.

2.3 Multiply-Located Values

An alternative to the selection messages Pirouette and similar choreographies use in choreographic conditionals is *multiply-located values* [Bates et al. 2025; Sweet et al. 2023]. These generalize the notion of local values at one location to local values at a set of locations. This feature recognizes that, after sending a value, all parties involved know that value, allowing the choreography to guarantee that the locations agree.

For instance, the choreography $\{\mathsf{A}, \mathsf{B}\}.(2 < 3)\ \{\mathsf{A}\}\!\rightsquigarrow \mathsf{C}$ specifies that A and B should *both* compute $2 < 3$ and A should send the result to C. Notably, the result of this computation is the multiply-located boolean value $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}.\mathsf{true}$, meaning a condition could branch on the result without requiring further synchronization between these three parties.

3 System Model

Before introducing the choreographic constructs in λQC, we specify the assumptions on the setting in which it operates. As is standard in the choreography literature, we assume a fixed set of locations $\mathcal{L}$, each with a unique name. Names are taken to be opaque identifiers associated with an underlying node, process, thread, etc.

Like Pirouette [Hirsch and Garg 2022], λQC abstracts over the language for local computation, requiring only that it satisfy a small set of rules. To support type and location polymorphism and multiply-located values, λQC adds a few assumptions beyond prior work, but it still supports numerous languages.

The local language must have a syntax that specifies a set of values, a small-step operational semantics, and a type system. We write $e_1 \longrightarrow e_2$ to denote that a local term e_1 steps to e_2 in the local language's semantics. The semantics must satisfy two properties:

(1) Values cannot step. That is, if $\mathrm{Val}(v)$, then there is no e such that $v \longrightarrow e$.
(2) Local steps satisfy the *diamond property*. That is, if $e_1 \longrightarrow e_2$ and $e_1 \longrightarrow e_3$, then either $e_2 = e_3$ or there is some e_4 such that $e_2 \longrightarrow e_4$ and $e_3 \longrightarrow e_4$.

Property (1) is taken from Pirouette, while property (2) ensures multiply-located computations all produce the same result. Property (2) may appear restrictive, but many pure functional languages and all deterministic languages enjoy it. We believe it can be weakened to the more general confluence statement that, if $e_1 \longrightarrow^* e_2$ and $e_1 \longrightarrow^* e_3$ then $e_2 \longrightarrow^* e_4$ and $e_3 \longrightarrow^* e_4$ for some e_4, but we leave the challenge of mechanically verifying this conjecture to future work.

3.1 Local Kinding and Type System

As λQC is constructed generically over the local language, λQC's type system—and the guarantees it provides—depend on the local language specifying a type system. In particular, we require the local language to specify both a type system and a kinding system. For simplicity, we show only a single local kind ($*_e$) here, but the results generalize to an arbitrary non-empty set of local kinds, which our Rocq formalization allows. The local type system can be polymorphic, but this is not required—it may instead be a simple type system.

The local kinding and type systems are specified by a type well-formedness judgment $\Gamma \Vdash t$ and an expression typing judgment $\Gamma; \Sigma \Vdash e : t$, respectively. The double-vertical turnstile $\Vdash$ indicates the local kind and type systems. In each judgment Γ is a kinding context, and Σ is a typing context.

$$\frac{\alpha \in \Gamma}{\Gamma \Vdash \alpha} \qquad \frac{\Gamma \Vdash t_1 \qquad \Gamma, \alpha \Vdash t_2}{\Gamma \Vdash t_2[\alpha \mapsto t_1]} \qquad \frac{\Gamma; \Sigma \Vdash e : t}{\Gamma \Vdash \Sigma} \qquad \frac{\Gamma; \Sigma \Vdash e : t}{\Gamma \Vdash t}$$

$$\frac{\Gamma \Vdash \Sigma \qquad x{:}t \in \Sigma}{\Gamma; \Sigma \Vdash x : t} \qquad \frac{\Gamma; \Sigma \Vdash e_1 : t_1 \qquad \Gamma; \Sigma, x{:}t_1 \Vdash e_2 : t_2}{\Gamma; \Sigma \Vdash e_2[x \mapsto e_1] : t_2} \qquad \frac{\Gamma \Vdash t_1 \qquad \Gamma, \alpha; \Sigma \Vdash e_2 : t_2}{\Gamma; \Sigma[\alpha \mapsto t_1] \Vdash e_2[\alpha \mapsto t_1] : t_2[\alpha \mapsto t_1]}$$

Fig. 1. Required-admissible kinding and typing rules for the local language.

We only require that the rules in Figure 1 be admissible—they must be true of the system, but need not be axioms. These rules are standard for polymorphic type systems.

As in Pirouette, the local type system must include a type, which we write as bool, where every value can be interpreted as either true or false. These are required to define the behavior of conditionals, which branch on local values. We also require three more types: loc_ρ, locset_ρ, and tyRep, defining the first-class local *representations* of location names, sets of locations, and local types, respectively. The defining property of representations is that we can convert values of type loc_ρ to location names in $\mathcal{L}$, values of type locset_ρ to *sets* of location names, and values of type tyRep to well-formed types in the local language. For example, one could represent location names using integers, mapping 0 to A, 1 to B, and all other integers to C. We write representations of a type, location name, or set of location names as $\lceil t \rfloor$, $\lceil L \rfloor$, or $\lceil \{L_1, \ldots, L_n\} \rfloor$ respectively, to disambiguate from the resolved type t, location L, or set of locations $\{L_1, \ldots, L_n\}$. In the above example, $\lceil \mathrm{A} \rfloor$ is syntactic sugar for 0, while A refers to the *actual* location $\mathrm{A} \in \mathcal{L}$. A location need not have a unique representative, but each value can only represent one location. There is also no requirement that all possible booleans, location names, etc. be present, or even that the local types be inhabited, but uninhabited types will render choreographic features unavailable.

The subscript ρ in the type loc_ρ is an upper bound on the set of locations to which an expression of that type may resolve. For instance, both $\lceil \mathrm{A} \rfloor$ and if e then $\lceil \mathrm{A} \rfloor$ else $\lceil \mathrm{B} \rfloor$ can have type $\text{loc}_{\{\mathrm{A},\mathrm{B}\}}$, but $\lceil \mathrm{C} \rfloor$ cannot. For locset_ρ, the annotation must be a superset of any value of this type (e.g., $\lceil \{\mathrm{A}, \mathrm{B}\} \rfloor : \text{locset}_{\{\mathrm{A},\mathrm{B},\mathrm{C}\}}$). The precision of ρ does not explicitly affect the operational semantics of our language, but an imprecise annotation may require a choreography to add unnecessary communication to remain well-typed (see Section 4.4). In practice, this static upper bound could be computed in a multitude of ways, such as refinement types [Freeman and Pfenning 1991] or a separate static analysis. In this work, we assume for simplicity that the annotations are given directly by the local type system, and leave computing them precisely to future work.

Lastly, the local type system must provide standard progress and preservation guarantees:

- *Local Progress:* If $\Vdash e : t$ then either e is a value, or there is some e' such that $e \longrightarrow e'$.
- *Local Preservation:* If $\Gamma; \Sigma \Vdash e : t$ and $e \longrightarrow e'$, then $\Gamma; \Sigma \Vdash e' : t$.

3.1.1 Example Local Languages. Many simply-typed and polymorphic λ-calculi satisfy our requirements with very minor modification. To show that our requirements are both reasonable and satisfiable we present two examples here, with the full details contained in our formalization.

Example 1 (Simply-Typed λ-Calculus)**.** The simply-typed, call-by-value λ-calculus with primitive integers satisfies our requirements. Specifically, integers can represent both booleans (zero is false while all non-zero values are true) and location names. Having many representations of true is not a concern, as each representation remains unambiguous. The space of both location sets and types, locset_ρ and tyRep, respectively, can both be the empty type, as not all representations are required to exist. This choice will render first-class location sets and local types unavailable at the choreographic level, but does not otherwise interfere with the availability of choreographic type-, location-, or location-set–polymorphism.

Example 2 (System F)**.** System F with primitive booleans, integers representing locations, and lists of integers representing location sets satisfies our requirements. Like using integers for true and false, having multiple ordered lists represent the same set of locations is not a concern. We can also include recursive functions, as there is no requirement that local expressions terminate.

To represent a non-trivial set of local types, we can define tyRep to be a primitive data type with constructors $\lceil\mathsf{int}\rfloor$, $\lceil\mathsf{bool}\rfloor$, and $e_1 \lceil\to\rfloor e_2$. Since the representations are expressions not types, tyRep behaves identically to the other primitive types of the language. The syntax of the resulting example language is shown below.

$$\begin{array}{llcl}
\text{Types} & t & ::= & \alpha \mid \mathsf{loc}_\rho \mid \mathsf{locset}_\rho \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{list}(t) \mid \mathsf{tyRep} \mid t_1 \to t_2 \mid \forall\alpha.t \\
\text{Expressions} & e & ::= & x \mid \mathsf{fun}\, f(x:t) \coloneqq e \mid e_1\ e_2 \mid \Lambda\alpha.e \mid e\ t \\
 & & \mid & n \in \mathbb{Z} \mid e_1 + e_2 \mid e_1 = e_2 \mid e_1 < e_2 \\
 & & \mid & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \\
 & & \mid & \mathsf{nil} \mid \mathsf{cons}(e_1, e_2) \mid \mathsf{case}\ e\ \mathsf{of}\ (\mathsf{nil} \Rightarrow e_1)\ (\mathsf{cons}(x,y) \Rightarrow e_2) \\
 & & \mid & \lceil\mathsf{int}\rfloor \mid \lceil\mathsf{bool}\rfloor \mid e_1 \lceil\to\rfloor e_2
\end{array}$$

As an example of how the type system is used to provide a static upper-bound on representations of locations, two of the typing rules for the loc_ρ type are given below.

$$\frac{n \in \rho \subseteq \mathbb{Z} \qquad \Gamma \Vdash \Sigma}{\Gamma;\Sigma \Vdash n : \mathsf{loc}_\rho} \qquad\qquad \frac{\Gamma;\Sigma \Vdash e : \mathsf{bool} \qquad \Gamma;\Sigma \Vdash e_1 : \mathsf{loc}_\rho \qquad \Gamma;\Sigma \Vdash e_2 : \mathsf{loc}_\rho}{\Gamma;\Sigma \Vdash \mathsf{if}\ e\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 : \mathsf{loc}_\rho}$$

4 The λQC Language

We now present the Quick Change Choreographic calculus (λQC), a polymorphic λ-calculus for choreographies that supports communication of dynamically generated (sets of) location names and types, as well as multiply-located values and algebraic and recursive data types, while retaining the traditional guarantee of deadlock freedom.

4.1 λQC Syntax

Figure 2 presents the full syntax of λQC. To visually differentiate classes of variables, we write choreographic program variables in uppercase Roman characters ($X, Y, F, \ldots$), local program variables in lowercase Roman characters ($x, y, f, \ldots$), and type, location, and location set variables in lowercase Greek characters ($\alpha, \beta, \ldots$). The metavariable ℓ denotes a location, ρ a set of locations, τ a choreographic type, t a local type, and κ a kind.

Much of the λQC syntax consists of standard algebraic and recursive datatypes lifted to choreographies, but there are a few forms of note. First, the term $\rho.e$ specifies that the set of locations ρ should run local program e. Note that ρ must be non-empty, as an empty set of locations performing

$$\begin{array}{llcl}
\text{Selection Labels} & d & ::= & \mathbf{L} \mid \mathbf{R} \\
\text{Choreographies} & C & ::= & X \mid \rho.e \mid \mathsf{fun}\, F(X) \coloneqq C \mid C_1\ C_2 \mid \Lambda\alpha::\kappa.\, C \mid C\ t \\
 & & \mid & \mathsf{fold}\ C \mid \mathsf{unfold}\ C \mid (C_1, C_2) \mid \mathsf{fst}\ C \mid \mathsf{snd}\ C \\
 & & \mid & \mathsf{inl}\ C \mid \mathsf{inr}\ C \mid \mathsf{case}\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inl}\ Y \Rightarrow C_2) \\
 & & \mid & C \overset{\{\ell\}}{\rightsquigarrow} \rho \mid \ell[d] \rightsquigarrow \rho\ ;C \mid \mathsf{if}_\rho\ C\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \\
 & & \mid & \mathsf{let}\ \rho.x:t \coloneqq C_1\ \mathsf{in}\ C_2 \mid \mathsf{let}\ \rho.\alpha::\kappa \coloneqq C_1\ \mathsf{in}\ C_2
\end{array}$$

Fig. 2. Syntax of Choreographies

a computation would be meaningless. We use the shorthand $\ell.e$ to mean $\{\ell\}.e$. Local programs can use local variables bound in the scope of either the choreography or the local program itself. For these variables, every location has a separate namespace, so $\mathsf{A}.x$ denotes variable x in the namespace of A. Each location's namespace is separate, so that $\mathsf{A}.x \neq \mathsf{B}.x$, and this is reflected in the substitution and renaming operations. Local variables can be multiply-located, where we write $\rho.x$ to mean that the local variable x is in the namespace of every location in the set ρ.

A core feature of choreographies is message passing, which λ_{QC} denotes as $C \ {}_{\{\ell\}}\!\rightsquigarrow \rho$ to indicate sending the result of evaluating C to all locations in the set ρ. Here C must produce a local value located at ℓ (potentially among others); ℓ then sends that value as a message to everyone in ρ. When the result of C exists only at a single location, we elide the ℓ on the arrow for simplicity. After the message send, the locations that know the result of C include ρ and *anyone who already knew the output of* C. For example, the choreography $\mathsf{A}.3 \rightsquigarrow \{\mathsf{B}, \mathsf{C}\}$ reduces to $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}.3$.

Choreographic conditionals, written $\mathsf{if}_\rho\ C\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, branch the entire choreography on the result of a local computation. Here C must produce a boolean known to ρ. Although the locations in ρ know which branch to take, other participants in the program may not. This problem can be solved in two ways: explicitly share the branch condition with all participants, or include selection statements $\ell[d] \rightsquigarrow \rho'\ ;\ C$ in the branches to inform locations in ρ' of which branch was taken. The former follows the literature on multiply-located values [Bates et al. 2025; Sweet et al. 2023], while the latter follows literature on selection messages [see, e.g., Graversen et al. 2024; Hirsch and Garg 2022; Montesi 2013, 2023]. As the first choreographic language to incorporate both options, λ_{QC} allows added flexibility and supports both of the following equivalent choreographies.

$\mathsf{if}_{\{\mathsf{A},\mathsf{B}\}}\ \mathsf{A}.\mathsf{true} \rightsquigarrow \mathsf{B}$	$\mathsf{if}_{\{\mathsf{A}\}}\ \mathsf{A}.\mathsf{true}$
$\mathsf{then}\ \mathsf{B}.1$	$\mathsf{then}\ \mathsf{A}[\mathbf{L}] \rightsquigarrow \mathsf{B}\ ;\ \mathsf{B}.1$
$\mathsf{else}\ \mathsf{B}.2$	$\mathsf{else}\ \mathsf{A}[\mathbf{R}] \rightsquigarrow \mathsf{B}\ ;\ \mathsf{B}.2$

The type abstraction $\Lambda\alpha :: \kappa.\, C$ and type application $C\ t$ together implement polymorphism. As described in Section 4.3 below, λ_{QC} has four kinds, all of which are valid in type abstractions.

In addition, λ_{QC} includes typical algebraic and recursive data types including pairs, projections, injections, case expressions, and the isorecursive constructor fold and eliminator unfold. These constructs have the standard semantics for a strict functional language [Pierce 2002, Chapters 11 & 20], but note that they represent *global* data. As a result, after evaluating a choreographic sum type, for instance, *all* participants know and agree on whether the result is of the form $\mathsf{inl}\ V_1$ or $\mathsf{inr}\ V_2$, although their knowledge of the contents of V_1 and V_2 may differ.

A major contribution of this work is the presence of *two* let expressions: one for local values and one for types. Local-let expressions act like standard let expressions, binding local variables to the result of choreographic computations. If C_1 produces a value located at ρ, then $\mathsf{let}\ \rho.x : t \coloneqq C_1\ \mathsf{in}\ C_2$ binds the result to variable x in the namespace of ρ, making it available in future local computations. Importantly, ρ may be any subset of the locations who know the output of C_1.

Our new type-let expressions, written $\mathsf{let}\ \rho.\alpha :: \kappa \coloneqq C_1\ \mathsf{in}\ C_2$, convert *representations* of locations, location sets, and types, into *actual* locations, location sets, and types. They are semantically similar to a local-let binding above, except they bind type variables rather than local variables. Specifically, C_1 must produce a representation known to ρ of a location name, location set, or local type, which is then reified to a type-level value and bound to α in the body of C_2. We elide ρ and κ for legibility when they are obvious from context. Combining the type-let expression with the collecting sends described above allows arbitrary local language computations to dynamically select locations and propagate the choice to the choreographic level, as shown in Example 3.

Example 3 (Load Balancer)**.** Recall the example of a distributed thread pool from Section 1. Representations of type loc are a value like any other, so the local language could provide a function

$$\text{Redices} \quad R \quad ::= \quad \rho.(e_1 \to e_2) \mid \mathsf{App} \mid \ell.m \rightsquigarrow \rho \mid \mathsf{let}\ \alpha.\rho \coloneqq t$$

$\boxed{\text{rloc}(R)}$

$$\begin{aligned}
\text{rloc}(\rho.(e_1 \to e_2)) &= \rho \\
\text{rloc}(\mathsf{App}) &= \mathcal{L} \\
\text{rloc}(\ell.m \rightsquigarrow \rho) &= \{\ell\} \cup \rho \\
\text{rloc}(\mathsf{let}\ \rho.\alpha \coloneqq t) &= \rho
\end{aligned}$$

$\boxed{\text{cloc}(C)}$

$$\begin{aligned}
\text{cloc}(X) &= \varnothing \\
\text{cloc}(\rho.e) &= \rho \\
\text{cloc}(\mathsf{fun}\ F(X) \coloneqq C) &= \varnothing \\
\text{cloc}(C_1\ C_2) &= \mathcal{L} \\
\text{cloc}(C\ {}_{\{\ell\}}\!\!\rightsquigarrow \rho) &= \text{cloc}(C) \cup \{\ell\} \cup \rho \\
\text{cloc}(\mathsf{let}\ \rho.\alpha :: \kappa \coloneqq C_1\ \mathsf{in}\ C_2) &= \text{cloc}(C_1) \cup \text{cloc}(C_2) \cup \rho
\end{aligned}$$

Fig. 3. Selected Redices and Location Function Rules. Here m is either a local value v or a selection label d.

tasks : loc $\to$ int that returns the number of tasks assigned to a worker. The (M)anager can then balance the load between workers A and B by allocating computation e as follows:

$$\mathsf{let}\ \alpha \coloneqq \mathsf{M}.\begin{pmatrix}\mathsf{if}\ (\mathsf{tasks}(\lceil \mathsf{A} \rfloor) < \mathsf{tasks}(\lceil \mathsf{B} \rfloor)) \\ \mathsf{then}\ \lceil \mathsf{A} \rfloor\ \mathsf{else}\ \lceil \mathsf{B} \rfloor\end{pmatrix} \rightsquigarrow \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}\ \mathsf{in}\ \alpha.e \rightsquigarrow \mathsf{C}$$

Say A currently has fewer tasks than B, so the local program will evaluate to $\lceil \mathsf{A} \rfloor$, meaning the first expression in the let will evaluate to

$$\mathsf{M}.\lceil \mathsf{A} \rfloor \rightsquigarrow \{\mathsf{A}, \mathsf{B}, \mathsf{C}\} \Longrightarrow_c \{\mathsf{M}, \mathsf{A}, \mathsf{B}, \mathsf{C}\}.\lceil \mathsf{A} \rfloor$$

The type-let then reifies the representation $\lceil \mathsf{A} \rfloor$ to an actual location, producing the step

$$\mathsf{let}\ \alpha \coloneqq \{\mathsf{M}, \mathsf{A}, \mathsf{B}, \mathsf{C}\}.\lceil \mathsf{A} \rfloor\ \mathsf{in}\ \alpha.e \rightsquigarrow \mathsf{C} \Longrightarrow_c \mathsf{A}.e \rightsquigarrow \mathsf{C}$$

4.2 λQC Semantics

The operational semantics of λQC consists of a small-step relation using a labeled transition system of the form $C_1 \overset{R}{\Longrightarrow}_c C_2$. The label R represents a redex that tracks the specific reduction occurring.

Choreographies describe the actions of multiple locations, and so distinct locations should be able to perform (unrelated) actions in any order. To capture this idea, our semantics includes *out-of-order* reductions. Importantly, operations for any one location should always execute in the specified order. To enforce this requirement, the semantics uses the redices in the step relation to determine which locations are involved in the step, and only allows reordering of operations when the computation it is jumping before involves a disjoint set of locations.

More precisely, the *redex locations* function rloc(R) returns the set of locations involved in R and the *choreography locations* function cloc(C) gives those involved in C. For example, if A sends v to B (denoted by the redex $\mathsf{A}.v \rightsquigarrow \mathsf{B}$), then precisely A and B participate, so rloc$(\mathsf{A}.v \rightsquigarrow \mathsf{B}) = \{\mathsf{A}, \mathsf{B}\}$. The cloc$(C)$ function operates on a whole choreography, not just a single step, so even though A must take multiple steps before B gets involved, cloc$(\mathsf{let}\ \mathsf{A}.x \coloneqq \mathsf{A}.1\ \mathsf{in}\ (\mathsf{A}.(1 + x) \rightsquigarrow \mathsf{B})) = \{\mathsf{A}, \mathsf{B}\}$. Figure 3 shows selected redices and definitions of both location functions. Note that all locations participate in choreographic function application, so cloc$(C_1\ C_2) = $ rloc$(\mathsf{App}) = \mathcal{L}$.

It is then safe to move a step R before an entire computation C if the set of participants are disjoint (i.e., cloc$(C) \cap$ rloc$(R) = \varnothing$), even if a normal in-order execution would execute C before step R. The following out-of-order rule for type-let expressions exhibits this structure, also prohibiting the out-of-order step from including locations binding a variable in the let.

$$[\text{C-TyLetI}]\ \frac{C_2 \overset{R}{\Longrightarrow}_c C_2' \qquad \text{cloc}(C_1) \cap \text{rloc}(R) = \varnothing \qquad \rho \cap \text{rloc}(R) = \varnothing}{\mathsf{let}\ \rho.\alpha :: \kappa \coloneqq C_1\ \mathsf{in}\ C_2 \overset{R}{\Longrightarrow}_c \mathsf{let}\ \rho.\alpha :: \kappa \coloneqq C_1\ \mathsf{in}\ C_2'}$$

$$\text{[C-Done]}\ \frac{e_1 \longrightarrow e_2 \qquad \mathrm{fv}(\rho) = \varnothing}{\rho.e_1 \xRightarrow{\rho.(e_1 \to e_2)}_c \rho.e_2} \qquad \text{[C-TyLetV]}\ \frac{\mathrm{Val}(\lceil t \rfloor) \qquad \mathrm{fv}(\rho) = \varnothing}{\mathsf{let}\ \rho.\alpha :: \kappa \coloneqq \rho'.\lceil t \rfloor\ \mathsf{in}\ C \xRightarrow{\mathsf{let}\ \rho.\alpha := t}_c C[\alpha \mapsto t]} \qquad \text{[C-SendV]}\ \frac{\mathrm{Val}(v) \qquad L_1 \in \rho_1 \qquad \mathrm{fv}(\rho_2) = \varnothing}{\rho_1.v\ \{L_1\}\!\rightsquigarrow \rho_2 \xRightarrow{L_1.v \rightsquigarrow \rho_2}_c (\rho_1 \cup \rho_2).v}$$

Fig. 4. Selected λQC Operational Semantics

λQC also allows out-of-order execution in the branches of an if-expression before fully evaluating the branch condition. Such a step is safe only when: (1) the locations involved in the step are disjoint from those computing the branching condition, similarly to C-TyLetI, and (2) this precise step is guaranteed to happen eventually. The latter holds only if both branches take identical steps, which we enforce by requiring identical redices. The result is the following C-IfI rule.

$$\text{[C-IfI]}\ \frac{\begin{array}{cc} C_1 \xRightarrow{R}_c C_1' & C_2 \xRightarrow{R}_c C_2' \\ \mathrm{cloc}(C) \cap \mathrm{rloc}(R) = \varnothing & \rho \cap \mathrm{rloc}(R) = \varnothing \end{array}}{\mathsf{if}_\rho\ C\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \xRightarrow{R}_c \mathsf{if}_\rho\ C\ \mathsf{then}\ C_1'\ \mathsf{else}\ C_2'}$$

As an example, consider the choreography $\mathsf{if}\ \mathsf{A}.e\ \mathsf{then}\ \mathsf{B}.(2+3)\ \mathsf{else}\ \mathsf{B}.(2+3)$. Although the condition is not yet evaluated, B will run the same program on either branch, and B is not involved in computing the branch condition. That is, no matter what $\mathsf{A}.e$ evaluates to, B will run $\mathsf{B}.(2+3)$. It is thus safe to reduce both branches to $\mathsf{B}.5$ and the overall choreography to $\mathsf{if}\ \mathsf{A}.e\ \mathsf{then}\ \mathsf{B}.5\ \mathsf{else}\ \mathsf{B}.5$. However, no reordering is possible in the following choreography.

$$\mathsf{if}_{\mathsf{A}}\ \big(\mathsf{let}\ \mathsf{A}.x \coloneqq (\mathsf{B}.5 \rightsquigarrow \mathsf{A})\ \mathsf{in}\ \mathsf{A}.(x < 4)\big)\ \mathsf{then}\ \mathsf{B}.(2+3)\ \mathsf{else}\ \mathsf{B}.(2+3)$$

Despite B performing identical computation in both branches and only A knowing which way the branch will go, executing the branches before stepping $\mathsf{B}.5 \rightsquigarrow \mathsf{A}$ in the condition would reorder local operations for B, which is not permitted.

Figure 4 contains a selection of additional semantic rules. C-Done shows how local programs execute in a choreography, C-TyLetV shows how type representations are reified to types, and C-SendV formalizes the collecting message send semantics discussed in Section 4.1. Each rule also requires $\mathrm{fv}(\rho) = \varnothing$, which simply demands all location variables to be resolved before taking a step. Otherwise it would be impossible to know who is performing the computation. Due to the out-of-order rules, this condition is nontrivial, even for closed choreographies.

The remaining rules are either very similar to those presented here or are standard for a strict functional language. The complete semantics can be found in Appendix A.7.

4.2.1 Substitution. The separation between choreographic variables, local variables, and type variables results in three corresponding types of variable substitution. Choreographic variable substitution, denoted $C_1[X \mapsto C_2]$, follows standard capture-avoiding substitution rules. Local variable substitution for locations or location sets, denoted $C[\ell|x \mapsto e]$ or $C[\rho|x \mapsto e]$, respectively, is similar, but it operates only over the namespace of the location ℓ (resp. ρ), which may itself be a type variable. Notably, a multiply-located variable must be substituted for its entire namespace simultaneously, so $(\rho.x)[\rho|x \mapsto e] = \rho.e$, but $(\{\ell, \ell'\}.x)[\ell|x \mapsto e]$ is undefined.

Type substitution, denoted $C[\alpha \mapsto t]$, requires additional care when substituting locations or location sets. A naïve implementation can capture variables due to namespace collisions. For example, consider the following choreography with a location variable α and concrete location L.

$$\begin{aligned} C = \mathsf{let}\ & \alpha.x \coloneqq \alpha.2 \\ & \mathsf{L}.x \coloneqq \mathsf{L}.3 \\ & \alpha.y \coloneqq \mathsf{L}.x \rightsquigarrow \alpha \\ \mathsf{in}\ & \alpha.(x + y) \end{aligned}$$

Because α and L are (syntactically) distinct, the local variables $\alpha.x$ and $\mathsf{L}.x$ are also distinct, so this choreography should always produce $\alpha.5$ regardless of the concrete location to which α resolves.

But when α resolves to L, how do we define $C[\alpha \mapsto \mathsf{L}]$? One might think that, since nothing binds α in C, we can simply replace all instances of α with L. However, doing so makes the previously-distinct local variables $\alpha.x$ and $\mathsf{L}.x$ collapse. The variable $\alpha.x$ is incorrectly captured by the definition of $\mathsf{L}.x$ and the choreography would wrongly evaluate to $\mathsf{L}.6$. Note that, when substituting $[\alpha \mapsto \mathsf{L}]$, capture can occur when binding either $\alpha.x$ or $\mathsf{L}.x$. Inside a binding of $\alpha.x$, free instances of $\mathsf{L}.x$ in the body will be captured, and so too will free instances of $\alpha.x$ be captured when substituting under a binding of $\mathsf{L}.x$.

To avoid this namespace capture, substitutions of locations in local-let expressions must rename variables when binding within *either* namespace. The following rules define safe location set substitution for local-let bindings, with single location substitution defined as a special case by replacing the set σ with the singleton $\{\ell\}$ in the side conditions. Here $\mathrm{fv}_\rho(C)$ denotes the local variables free in C for *any* location in ρ.

$$(\mathsf{let}\ \rho.x \coloneqq C_1\ \mathsf{in}\ C_2)[\alpha \mapsto \sigma] = \begin{cases} \mathsf{let}\ (\rho[\alpha \mapsto \sigma]).x \coloneqq C_1[\alpha \mapsto \sigma] & \text{if } \alpha \subseteq \rho \text{ and } x \notin \mathrm{fv}_{\sigma-\rho}(C_2) \\ \ \mathsf{in}\ C_2[\alpha \mapsto \sigma] & \\ \mathsf{let}\ (\rho[\alpha \mapsto \sigma]).y \coloneqq C_1[\alpha \mapsto \sigma] & \text{if } \alpha \subseteq \rho, x \in \mathrm{fv}_{\sigma-\rho}(C_2), \\ \ \mathsf{in}\ C_2[\rho|x \mapsto y][\alpha \mapsto \sigma] & \text{and } y \notin \mathrm{fv}_{\rho\cup\sigma}(C_2) \\ \mathsf{let}\ \rho.y \coloneqq C_1[\alpha \mapsto \sigma] & \text{if } \alpha \nsubseteq \rho, \sigma \cap \rho \neq \varnothing, \\ \ \mathsf{in}\ C_2[\rho|x \mapsto y][\alpha \mapsto \sigma] & x \in \mathrm{fv}_\alpha(C_2), \text{and } y \notin \mathrm{fv}_{\rho\cup\alpha}(C_2) \\ \mathsf{let}\ \rho.x \coloneqq C_1[\alpha \mapsto \sigma] & \text{if } \alpha \nsubseteq \rho \text{ and either} \\ \ \mathsf{in}\ C_2[\alpha \mapsto \sigma] & \sigma \cap \rho = \varnothing \text{ or } x \notin \mathrm{fv}_\alpha(C_2) \end{cases}$$

4.3 λQC Kinding System

We now turn to our static semantics, which has two components: an elementary kinding system, and a typing system. For notational brevity we assume throughout that new variables are always fresh. A kinding judgment takes the form $\Gamma \vdash t :: \kappa$, where Γ is a kinding context, t is a type, and κ is a kind. The kind κ classifies t as either a program type ($*$), a location ($*_\ell$), a set of locations ($*_s$), or a local program type ($*_e$). Figure 5 presents the syntax for the λQC types and kinds.

Kinds	κ	::=	$* \mid *_\ell \mid *_s \mid *_e$
Local Program Types	t_e	::=	$\alpha \mid \mathsf{bool} \mid \mathsf{tyRep} \mid \mathsf{loc}_\rho \mid \mathsf{locset}_\rho \mid \ldots$
Locations	L, A, B, ...	$\in$	$\mathcal{L}$
Choreography Types	τ, ℓ, ρ, t	::=	$\alpha \mid t_e@\rho \mid \tau_1 \to \tau_2 \mid \forall\alpha::\kappa.\tau$
		$\mid$	$\tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \mu\alpha.\tau \mid \mathsf{L} \mid \{\ell\} \mid \rho_1 \cup \rho_2$

Fig. 5. Syntax of Types and Kinds. Here α is a type variable.

$$[\text{T-Done}]\ \frac{\Gamma \vdash \rho :: *_s \qquad \Gamma; \Sigma|_\rho \Vdash e : t_e \qquad \Gamma \vdash \Delta \qquad \Gamma \vdash \Sigma}{\Gamma; \Delta; \Sigma \vdash \rho.e : t_e@\rho} \qquad [\text{T-Send}]\ \frac{\Gamma; \Delta; \Sigma \vdash C : t_e@\rho_1 \qquad \ell \in \rho_1 \qquad \Gamma \vdash \rho_2 :: *_s}{\Gamma; \Delta; \Sigma \vdash C\ \{\ell\}\rightsquigarrow \rho_2 : t_e@(\rho_1 \cup \rho_2)}$$

$$[\text{T-LetLocal}]\ \frac{\Gamma; \Delta; \Sigma \vdash C_1 : t_e@\rho_2 \qquad \rho_1 \subseteq \rho_2 \qquad \Gamma; \Delta; \Sigma, \rho_1.x{:}t_e \vdash C_2 : \tau}{\Gamma; \Delta; \Sigma \vdash \text{let } \rho_1.x{:}t_e \coloneqq C_1 \text{ in } C_2 : \tau} \qquad [\text{T-LetLoc}]\ \frac{\Gamma; \Delta; \Sigma \vdash C_1 : \text{loc}_{\rho_1}@\rho_3 \qquad \rho_1 \subseteq \rho_2 \subseteq \rho_3 \qquad \Gamma \vdash \tau :: * \qquad \Gamma, \alpha{::}*_\ell; \Delta; \Sigma \vdash C_2 : \tau}{\Gamma; \Delta; \Sigma \vdash \text{let } \rho_2.\alpha{::}*_\ell \coloneqq C_1 \text{ in } C_2 : \tau}$$

Fig. 6. Selected Typing Rules

All program types of kind $*$ directly extend their analogues from System F, except for base types. Our base types take the form $t_e@\rho$ and represent a single local program of type t_e running at all locations in the set ρ. The kind $*_\ell$ represents location names, which can refer to either concrete locations $\mathsf{L} \in \mathcal{L}$, or in-context location variables. The kind $*_s$ classifies (non-empty) finite sets of location names, which can be either a type variable, a singleton set ($\{\ell\}$), or a union of sets ($\rho_1 \cup \rho_2$). Finally, types of kind $*_e$ are precisely the types included in the local language under a given type variable context. Notably, local types may use type variables bound in the choreography, even if the local type system does not include a type variable binding mechanism.

The full kinding rules can be found in Appendix B.1.

4.4 λQC Type System

The second component of our static semantics is the type system. Typing judgments for λQC take the form $\Gamma; \Delta; \Sigma \vdash C : \tau$, where Γ is a kinding context, as described above, Δ is a choreographic typing context that handles variables bound by choreographic functions and case expressions, and Σ is a local typing context. These contexts are handled in a standard manner for a polymorphic type system [Pierce 2002, Chapter 23]. The local typing context Σ is a list of ascriptions of the form $\rho.x{:}t_e$ tracking variables bound by local let expressions (see Section 4.1). Figure 6 contains a selection of typing rules for λQC.

The T-Done rule type-checks local computations by appealing to the local type system, but it needs to know which local variables are in scope. In a multiply-located computation $\rho.e$, every location in ρ should compute the same result, so each free variable in e must be bound with the same meaning at every location in ρ. To check this requirement, we define a projection operation $\Sigma|_\rho$ that restricts Σ to only those local variables bound in a namespace ρ' where $\rho \subseteq \rho'$. Formally,

$$\Sigma|_\rho = \begin{cases} \cdot & \text{if } \Sigma = \cdot \\ \Sigma'|_\rho, x{:}t_e & \text{if } \Sigma = \Sigma', \rho'.x{:}t_e \text{ and } \rho \subseteq \rho' \\ \Sigma'|_\rho & \text{if } \Sigma = \Sigma', \rho'.x{:}t_e \text{ and } \rho \not\subseteq \rho' \end{cases}$$

The T-Send rule ensures (1) that only local values can be sent by requiring C to have type $t_e@\rho_1$, and (2) that the sender (ℓ) must know the value being sent. By locating the output type at $\rho_1 \cup \rho_2$, it captures the multiply-located collecting semantics described in Section 4.2.

The other two rules in Figure 6 are the two forms of let binding. The T-LetLocal rule for local-let bindings is very similar to a traditional let binding rule. It just restricts to binding local values—C_1 has type $t_e@\rho_2$—and requires that ρ_1 (the locations binding x) must all have access to the result of the computation C_1—that is, $\rho_1 \subseteq \rho_2$.

The T-LetLoc rule for type-let expressions, core to supporting first-class location names, is much more subtle. Indeed, a naïve definition could introduce type dependency or unsoundness. Type dependency is the simpler concern, and is addressed by the premise $\Gamma \vdash \tau :: *$, which demands that τ be a well-formed type in the context Γ *without the newly-bound variable* α. As a result, while the body C_2 of the let may freely reference α, its *type* τ may not. This restriction prevents the type of the entire let expression from depending on the value of C_1, thereby avoiding value dependency in the type system. As an example, the program below on the left is not well-typed, because the body has type int@α in which α is free. By contrast, the program on the right is well-typed because the body, and the choreography itself, has the type int@B.

$$\nvdash \left(\begin{array}{l} \mathsf{let}\ \mathsf{A}.\alpha \coloneqq \mathsf{A}.\lceil \mathsf{A} \rfloor \\ \ \mathsf{in}\ \alpha.(1+1) \end{array} \right) : \tau \qquad\qquad \vdash \left(\begin{array}{l} \mathsf{let}\ \{\mathsf{A},\mathsf{B}\}.\alpha \coloneqq \{\mathsf{A},\mathsf{B}\}.\lceil \mathsf{A} \rfloor \\ \quad \mathsf{B}.x \coloneqq \alpha.(1+1) \rightsquigarrow \mathsf{B} \\ \ \mathsf{in}\ \mathsf{B}.x \end{array} \right) : \mathsf{int@B}$$

The other unusual premise to T-LetLoc is $\rho_1 \subseteq \rho_2 \subseteq \rho_3$, which is necessary to ensure soundness. To understand why, recall Example 3, in which the thread-pool manager M dynamically assigns a task to either A or B. Consider what happens if M were to inform only B and the client C of their decision, but not A, the selected worker. As A is not aware that she should compute e and send back the result, C will wait forever on a message from A that never arrives, causing a deadlock.

Amazingly, the simple premise $\rho_1 \subseteq \rho_2 \subseteq \rho_3$ prevents all deadlocks resulting from the type-let expression. As described in Section 3.1, ρ_1 is a static upper bound on the set of values to which the dynamically generated location may resolve. The location variable is bound in the namespace of ρ_2, so only those locations may use the result, and all locations in ρ_3 know the value of the dynamically generated location name. The subset relationship therefore ensures that any location to which α may resolve will bind α, and that all locations that bind α know the value. In the unsound hypothetical above, $\rho_1 = \{\mathsf{A}, \mathsf{B}\}$ and $\rho_2 = \{\mathsf{M}, \mathsf{B}, \mathsf{C}\}$, violating this premise.

The full type system, which can be found in Appendix B.2, is sound with respect to the operational semantics, as demonstrated by the following mechanically verified theorems.

Theorem 1 (Type Preservation). *If* $\Gamma; \Delta; \Sigma \vdash C_1 : \tau$ *and* $C_1 \Longrightarrow_c C_2$, *then* $\Gamma; \Delta; \Sigma \vdash C_2 : \tau$.

Theorem 2 (Progress). *If* $\vdash C_1 : \tau$, *then either* C_1 *is a value, or there is some* C_2 *such that* $C_1 \Longrightarrow_c C_2$.

5 Network Language

Choreographies specify the behavior of concurrent systems, and compiling them to a set of programs that can actually run concurrently at different locations requires a language to specify that system. We thus provide a *network language* for λ_{QC}. It specifies programs at individual locations and how to compose them into a parallel system with concurrent operational semantics.

5.1 Network Language Syntax

The network language is a concurrent λ-calculus where messages are values from the same local language as choreographies. The syntax is similar to the choreographic syntax from Section 4,

$$\begin{array}{llcl}
\text{Network Program} & E & ::= & X \mid \mathtt{ret}(e) \mid \mathtt{()} \mid \mathtt{send}\ E\ \mathtt{to}\ \rho \mid \mathtt{recv\ from}\ \ell \\
& & \mid & E_1 \mathbin{;} E_2 \mid \mathtt{fun}\ F(X) \coloneqq E \mid E_1\ E_2 \mid \Lambda\alpha.\ E \mid E\ t \\
& & \mid & \mathtt{let}\ x \coloneqq E_1\ \mathtt{in}\ E_2 \mid \mathtt{let}\ \alpha :: \kappa \coloneqq E_1\ \mathtt{in}\ E_2 \\
& & \mid & \mathtt{allow}\ \ell\ \mathtt{choice}\ (\mathbf{L} \Rightarrow E_1)\ (\mathbf{R} \Rightarrow E_2) \mid \mathtt{choose}\ d\ \mathtt{for}\ \rho \mathbin{;} E \\
& & \mid & \mathtt{if}\ E\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2 \mid \mathtt{AmI}\!\in \rho\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2 \\
\text{Systems} & \Pi & ::= & L_1 \triangleright E_1 \parallel \ldots \parallel L_n \triangleright E_n
\end{array}$$

Fig. 7. Selected Network Program Syntax. Here $L \in \mathcal{L}$ is a concrete location name.

except sending and receiving messages are now split into two separate constructs. Figure 7 contains selected syntactic forms, using the same variable naming conventions as choreographies. The full definition is available in Appendix C.1.

The network language counterpart of the choreographic local computation $\rho.e$ is the return expression $\texttt{ret}(e)$, which similarly executes the local language program e. Since a network program is only for a single location L, it should only run e when $L \in \rho$ and do nothing otherwise. We include a unit value $()$ to represent a network program that does nothing.

Choreographies have a single message sending operation, which represents both sending and receiving the message. As the network language describes only a single location's behavior, it splits these into two constructs: $\texttt{send}\ E\ \texttt{to}\ \rho$ which multicasts the results of E to every location in ρ, and $\texttt{recv from}\ \ell$, which receives a local value from location ℓ. Note that $\texttt{send}$ can only send local values, so $\texttt{send}\ ()\ \texttt{to}\ \rho$, for instance, would be stuck.

Both local-let and type-let expressions are mirrored identically from choreographies to the network language, as are recursive functions, type abstractions, and applications for both. The network language also includes a primitive sequencing operator $E_1\ ;\ E_2$ for a similar reason to why it includes $()$. Specifically, a location L may have computation to perform in the head and body of a choreographic let expression, yet not be included in the locations binding a variable. In this case, L will sequence their actions from the two parts of the let expression.

Finally, the network language contains three branching mechanisms for different purposes: the standard $\texttt{if}\ E\ \texttt{then}\ E_1\ \texttt{else}\ E_2$ construct, $\texttt{allow-choice}$ to implement choreographic selection statements, and $\texttt{AmI}{\in}$, which branches based on the identity of the current location.

Recall from Section 4.1 that a selection statement $\ell[d] \rightsquigarrow \rho\ ;\ C$ has ℓ send d to each location in ρ so they know which branch to take in a choreographic $\texttt{if}$ statement. As with message sends, the network language splits the construct in two: sending and receiving. Each recipient is waiting on an *external choice*. Some other location ℓ must pick either $\mathbf{L}$ or $\mathbf{R}$, represented by the expression $\texttt{allow}\ \ell\ \texttt{choice}\ (\mathbf{L} \Rightarrow E_1)\ (\mathbf{R} \Rightarrow E_2)$, which will execute E_1 after receiving $\mathbf{L}$ or E_2 after receiving $\mathbf{R}$. Meanwhile, ℓ knows the choice d and can send it to all locations in ρ before executing E using the expression $\texttt{choose}\ d\ \texttt{for}\ \rho\ ;\ E$.

Note that an $\texttt{allow-choice}$ with only one branch, such as $\texttt{allow}\ \ell\ \texttt{choice}\ (\mathbf{L} \Rightarrow E)$, is valid. This one-sided branch will execute E if ℓ sends $\mathbf{L}$, but will get stuck if ℓ sends $\mathbf{R}$. We write $E_\perp$ for the program in a branch that might be missing.

The last form of branching is the "AmI-In" expression $\texttt{AmI}{\in}\ \rho\ \texttt{then}\ E_1\ \texttt{else}\ E_2$, which conditions on whether the currently executing location is in ρ. That is, when executing at location L, E_1 will execute if $L \in \rho$, and E_2 will execute otherwise. This construct generalizes the "AmI" expression of PolyChorλ [Graversen et al. 2024], which uses a single location ℓ instead of a set ρ, and branches based on equality with ℓ rather than inclusion. We write $\texttt{AmI}\ \ell\ \texttt{then}\ E_1\ \texttt{else}\ E_2$ as a shorthand for $\texttt{AmI}{\in}\ \{\ell\}\ \texttt{then}\ E_1\ \texttt{else}\ E_2$.

5.2 Network Language Operational Semantics

The network language semantics is a labeled transition system $L \triangleright E_1 \xRightarrow{l} E_2$ where L is the location executing the program and l is the label on the step. Just as network programs specify the operations of a single location, the labels only acknowledge the view of that location. Figure 8 shows selected transition labels and rules, with the full semantics available in Appendix C.3.

The ι label denotes an internal step, where either the network program or the local program reduces without interaction between locations. The N-Ret rule for local steps shows an example use of ι.

The *synchronized* internal label ι_{sync} denotes internal steps where *all* locations must take that step at the same time. It is used for steps like β-reduction (N-App) where the corresponding

$$\text{Transition Labels} \quad l \quad ::= \quad \iota \mid \iota_{\text{sync}} \mid m \rightsquigarrow \rho \mid L.m \rightsquigarrow$$

$$[\text{N-Ret}]\ \frac{e_1 \longrightarrow e_2}{L \triangleright \texttt{ret}(e_1) \overset{\iota}{\Longrightarrow} \texttt{ret}(e_2)} \qquad [\text{N-App}]\ \frac{f = \texttt{fun}\ F(X) \coloneqq E \qquad \text{Val}(V)}{L \triangleright f\ V \overset{\iota_{\text{sync}}}{\Longrightarrow} E[F \mapsto f, X \mapsto V]}$$

$$[\text{N-Send}]\ \frac{\text{Val}(v) \qquad \text{fv}(\rho) = \varnothing}{L \triangleright \texttt{send ret}(v)\ \texttt{to}\ \rho \xRightarrow{v \rightsquigarrow \rho \setminus \{L\}} \texttt{ret}(v)} \qquad [\text{N-Recv}]\ \frac{\text{Val}(v) \qquad L' \neq L}{L \triangleright \texttt{recv from}\ L' \xRightarrow{L'.v \rightsquigarrow} \texttt{ret}(v)}$$

$$[\text{N-Choose}]\ \frac{\text{fv}(\rho) = \varnothing}{L \triangleright \texttt{choose}\ d\ \texttt{for}\ \rho\ \texttt{;}\ E \xRightarrow{d \rightsquigarrow \rho \setminus \{L\}} E} \qquad [\text{N-AllowL}]\ \frac{L' \neq L}{L \triangleright \begin{array}{l}\texttt{allow}\ L'\ \texttt{choice} \\ \mid \mathbf{L} \Rightarrow E_1 \\ \mid \mathbf{R} \Rightarrow E_{2\perp}\end{array} \xRightarrow{L'.\mathbf{L} \rightsquigarrow} E_1}$$

Fig. 8. Selected Network Language Operational Semantics

$$\text{System Label} \quad l_S \quad ::= \quad \iota \mid \iota_{\text{sync}} \mid L_1.m \rightsquigarrow \rho$$

$$[\text{Internal}]\ \frac{L \triangleright \Pi(L) \overset{\iota}{\Longrightarrow} E}{\Pi \overset{\iota}{\Longrightarrow}_S \Pi[L \mapsto E]} \qquad [\text{Sync-Internal}]\ \frac{\forall L \in \text{dom}(\Pi).\ \left(L \triangleright \Pi(L) \xRightarrow{\iota_{\text{sync}}} \Pi'(L)\right)}{\Pi \xRightarrow{\iota_{\text{sync}}}_S \Pi'}$$

$$[\text{Comm}]\ \frac{L_1 \notin \rho \qquad L_1 \triangleright \Pi(L_1) \xRightarrow{m \rightsquigarrow \rho} E_1 \qquad \forall L \in \rho.\ \left(L \triangleright \Pi(L) \xRightarrow{L_1.m \rightsquigarrow} E_L\right)}{\Pi \xRightarrow{L_1.m \rightsquigarrow \rho}_S \Pi[L_1 \mapsto E_1, \rho \mapsto E_L]}$$

Fig. 9. System Semantics and Labels

choreographic step modifies the choreography for all participants. Synchronization in these cases is a technical requirement needed to prove correctness of the system, which we compare to other approaches in Section 8.1.

The labels $m \rightsquigarrow \rho$ and $L.m \rightsquigarrow$ appear with message sends and receives, respectively, including selection messages. As recipients do not know the value in advance, N-Recv is non-deterministic, allowing any value to arrive. The system semantics below ensures senders and receivers agree on the message. The same is true for N-AllowL and a symmetric N-AllowR rule. The sender's perspective is governed by N-Send and N-Choose, which ensure that the transition label matches the message and recipients intended by the program.

5.2.1 Network Systems. A choreography specifies *interactions* between multiple locations, which we represent with a parallel composition of network programs at different locations. Formally, a *system* $\Pi = \|_{L \in \mathfrak{L}} (L \triangleright E_L)$ maps each location L in a finite set $\mathfrak{L} \subseteq \mathcal{L}$ to the network program E_L it is currently executing. The system semantics is itself a larger labeled transition system, with combined labels shown in Figure 9. While network program labels show only one side of a send or receive, system labels reflect both sides and the semantics ensures that senders and recipients agree on the content of a message.

Figure 9 also shows how to lift the operational semantics of the network language to systems with three rules. Internal allows one location to independently take an internal step. Sync-Internal allows all locations to simultaneously perform a synchronized step. Comm parties sending and

$$\left(\begin{array}{l} \texttt{allow}\ \ell\ \texttt{choice} \\ \mid \mathbf{L} \Rightarrow E_1 \end{array}\right) \sqcup \left(\begin{array}{l} \texttt{allow}\ \ell\ \texttt{choice} \\ \mid \mathbf{R} \Rightarrow E_2 \end{array}\right) = \begin{array}{l} \texttt{allow}\ \ell\ \texttt{choice} \\ \mid \mathbf{L} \Rightarrow E_1 \\ \mid \mathbf{R} \Rightarrow E_2 \end{array}$$

$$\left(\begin{array}{l} \texttt{allow}\ \ell\ \texttt{choice} \\ \mid \mathbf{L} \Rightarrow E_1 \end{array}\right) \sqcup \left(\begin{array}{l} \texttt{allow}\ \ell\ \texttt{choice} \\ \mid \mathbf{L} \Rightarrow E_1' \end{array}\right) = \begin{array}{l} \texttt{allow}\ \ell\ \texttt{choice} \\ \mid \mathbf{L} \Rightarrow E_1 \sqcup E_1' \end{array}$$

$$\left(\begin{array}{l} \texttt{allow}\ \ell\ \texttt{choice} \\ \mid \mathbf{L} \Rightarrow E_1 \end{array}\right) \sqcup \left(\begin{array}{l} \texttt{allow}\ \ell\ \texttt{choice} \\ \mid \mathbf{L} \Rightarrow E_1' \\ \mid \mathbf{R} \Rightarrow E_2 \end{array}\right) = \begin{array}{l} \texttt{allow}\ \ell\ \texttt{choice} \\ \mid \mathbf{L} \Rightarrow E_1 \sqcup E_1' \\ \mid \mathbf{R} \Rightarrow E_2 \end{array}$$

Fig. 10. Selected Merge Operator Definitions

receiving messages together, requiring the sender and *all* specified recipients step at the same time with the same message value. Notationally, $\Pi[\rho \mapsto E_L]$ denotes the updated system mapping L to E_L if $L \in \rho$ and $\Pi(L)$ otherwise.

6 Endpoint Projection

With the compilation target fixed, we turn to the endpoint projection procedure that defines how to compile a choreography into a concurrent network system.

6.1 Network Program Merging

It is important to keep the projection definition *compositional* for simplicity and scalability, but choreographic if statements complicate this process and necessitate a *merge operator*. To understand why, consider the following choreography C.

$$\begin{tikzcd}
\text{if } \mathsf{A}.e \text{ then} & (\mathsf{A}[\mathbf{L}] \rightsquigarrow \mathsf{B} ; \mathsf{B}.1) \arrow[d, "{[\![\cdot]\!]_{\mathsf{B}}}"'] & \text{else} & (\mathsf{A}[\mathbf{R}] \rightsquigarrow \mathsf{B} ; \mathsf{B}.2) \arrow[d, "{[\![\cdot]\!]_{\mathsf{B}}}"'] & = & C \arrow[d, "{[\![\cdot]\!]_{\mathsf{B}}}"'] \\
& \begin{array}{l} \texttt{allow A choice} \\ \mid \mathbf{L} \Rightarrow \texttt{ret}(1) \end{array} & \sqcup & \begin{array}{l} \texttt{allow A choice} \\ \mid \mathbf{R} \Rightarrow \texttt{ret}(2) \end{array} & = & \begin{array}{l} \texttt{allow A choice} \\ \mid \mathbf{L} \Rightarrow \texttt{ret}(1) \\ \mid \mathbf{R} \Rightarrow \texttt{ret}(2) \end{array}
\end{tikzcd}$$

In the then branch, A will always send B the selection message L. In the projection of this branch, B should therefore wait to receive L and then return 1. There are no instructions for what to do if B receives R in this branch—which will never happen—so that side of the allow-choice is empty. Similarly, in the else branch, B can only ever receive R, so the L side of the allow-choice is empty.

In the full if statement, however, both branches are possible, so B's projection must include both options. To compositionally combine the allow-choice statements from each branch, we use a *merge* operator $E_1 \sqcup E_2$. Specifically, $\sqcup$ is an idempotent partial binary function defined structurally homomorphically on matching network programs that incorporates allow-choice branches that exist on only one side (and homomorphically merges those that exist in both). Figure 10 shows a few of the rules defining merge, with the full definition available in Appendix D.1.

This merge operator is extremely similar to the one in Pirouette [Hirsch and Garg 2022], but we allow functions to merge if their bodies merge, not just if their bodies are syntactically equal.

6.2 Endpoint Projection Definition

Using this merge function, we can define endpoint projection (EPP). The projection of a choreography C to a location (endpoint) L, denoted $[\![C]\!]_L$, is the network program L runs to implement its

$$\llbracket \rho.e \rrbracket_L = \begin{cases} \mathtt{ret}(e) & \text{if } L \in \rho \\ () & \text{otherwise} \end{cases}$$

$$\llbracket C \ \{\ell\}\!\rightsquigarrow \rho \rrbracket_L = \begin{cases} \mathtt{send}\ \llbracket C \rrbracket_L\ \mathtt{to}\ \rho & \text{if } L = \ell \\ \llbracket C \rrbracket_L \fatsemi \mathtt{recv\ from}\ \ell & \text{if } L \neq \ell \text{ and } L \in \rho \\ \llbracket C \rrbracket_L & \text{otherwise} \end{cases}$$

$$\llbracket \ell[d] \rightsquigarrow \rho\ ;\ C \rrbracket_L = \begin{cases} \mathtt{choose}\ d\ \mathtt{for}\ \rho\ ;\ \llbracket C \rrbracket_L & \text{if } L = \ell \\ \mathtt{allow}\ \ell\ \mathtt{choice}\ (d \Rightarrow \llbracket C \rrbracket_L) & \text{if } L \neq \ell \text{ and } L \in \rho \\ \llbracket C \rrbracket_L & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{if}_\rho\ C\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \rrbracket_L = \begin{cases} \mathtt{if}\ \llbracket C \rrbracket_L\ \mathtt{then}\ \llbracket C_1 \rrbracket_L\ \mathtt{else}\ \llbracket C_2 \rrbracket_L & \text{if } L \in \rho \\ \llbracket C \rrbracket_L \fatsemi (\llbracket C_1 \rrbracket_L \sqcup \llbracket C_2 \rrbracket_L) & \text{otherwise} \end{cases}$$

$$\llbracket \Lambda\alpha :: \kappa.\, C \rrbracket_L = \begin{cases} \Lambda\alpha :: *_\ell.\ \mathtt{AmI}\ \alpha\ \mathtt{then}\ \llbracket C[\alpha \mapsto L] \rrbracket_L\ \mathtt{else}\ \llbracket C \rrbracket_L & \text{if } \kappa = *_\ell \\ \Lambda\alpha :: *_s.\ \mathtt{AmI}{\in}\ \alpha\ \mathtt{then}\ \llbracket C[\alpha \mapsto \{L\} \cup \alpha] \rrbracket_L\ \mathtt{else}\ \llbracket C \rrbracket_L & \text{if } \kappa = *_s \\ \Lambda\alpha :: \kappa.\ \llbracket C \rrbracket_L & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{let}\ \rho.\alpha :: *_\ell \coloneqq C_1\ \mathsf{in}\ C_2 \rrbracket_L = \begin{cases} \mathtt{let}\ \alpha :: *_\ell \coloneqq \llbracket C_1 \rrbracket_L & \text{if } L \in \rho \\ \quad \mathtt{in\ AmI}\ \alpha\ \mathtt{then}\ \llbracket C_2[\alpha \mapsto L] \rrbracket_L\ \mathtt{else}\ \llbracket C_2 \rrbracket_L & \\ \llbracket C_1 \rrbracket_L \fatsemi \llbracket C_2 \rrbracket_L & \text{if } L \notin \rho \text{ and } \alpha \notin \mathrm{fv}(\llbracket C_2 \rrbracket_L) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Fig. 11. Selected EPP Definitions

part of C. EPP is partial (i.e., a choreography may fail to project) both because the merge operator is partial and because it must verify that locations only use type variables which they have bound.

EPP is defined as a structurally recursive function over the syntax of the choreography. Most of the rules simply convert choreographic syntax into its network language counterpart, but in some cases more involved translation is required. Figure 11 shows these cases.

The first four projection definitions capture intuitions described previously. For $\rho.e$, only locations in ρ should compute e while others should do nothing. For $C\ \{\ell\}\!\rightsquigarrow \rho$, everyone should perform their portion of the computation specified by C, then location ℓ should send a message, locations in ρ should receive a message, and that is all. For selection messages, location ℓ should announce the choice, while other locations in ρ wait for it and condition their behavior on the result. As there is only one choice here, the `allow-choice` only has one branch. The other branch can appear when projecting conditionals, where everyone first performs any computation specified by the condition, then locations who know the result of that condition—locations in ρ—simply branch, while EPP combines the branches for other locations using the merge operation defined above. Merging can fail, so EPP also fails for this location if it is undefined.

Sequencing Function. Note that the rules do not directly use the network language's native sequencing primitive $E_1\ ;\ E_2$, but instead use a *collapsing sequencing function* $E_1 \fatsemi E_2$ defined by

$$E_1 \fatsemi E_2 = \begin{cases} E_2 & \text{if } \mathrm{Val}(E_1) \\ E_1\ ;\ E_2 & \text{otherwise.} \end{cases}$$

This can be seen as a peephole optimization to eliminate null sequences whose primary purpose is to enable projected systems to properly simulate out-of-order choreographic steps. For instance, without collapsing sequenced values, B's projection of the choreography $\mathsf{let}\ \mathsf{A}.x \coloneqq \mathsf{A}.e\ \mathsf{in}\ \mathsf{B}.(2+3)$ would be $() \ ; \mathtt{ret}(2+3)$. An out-of-order step at B would reduce to $\mathsf{let}\ \mathsf{A}.x \coloneqq \mathsf{A}.e\ \mathsf{in}\ \mathsf{B}.5$, but to simulate this step, the projection would need to reduce to $() \ ; \mathtt{ret}(5)$, which is impossible because the reduction occurs to the right of a semicolon. The actual EPP defined above, however, projects these choreographies to $\mathtt{ret}(2+3)$ and $\mathtt{ret}(5)$, respectively, eliminating the concern.

Collapsing values also adds flexibility to EPP. To see why, consider the following choreography.

$$C = \mathsf{if}\ \mathsf{A}.e\ \mathsf{then}\ (\mathsf{let}\ \mathsf{A}.x \coloneqq \mathsf{A}.(1+1)\ \mathsf{in}\ \mathsf{B}.3)\ \mathsf{else}\ \mathsf{B}.3$$

Intuitively, B takes identical actions—returning 3—in both branches, so $[\![C]\!]_{\mathsf{B}}$ should simply be $\mathtt{ret}(3)$. However, without the sequencing function, the then branch would project to $() \ ; \mathtt{ret}(3)$, while the else branch would project to $\mathtt{ret}(3)$. These network programs do not merge, so C would not project. Collapsing the then branch to $\mathtt{ret}(3)$ allows the expected projection.

Process Polymorphism. The final two projections in Figure 11 concern location (set) abstraction. Choreographic type abstractions project to network type abstractions. When the type variable is a location ($*_\ell$) or location set ($*_s$), however, the body must behave differently depending on how the location executing relates to the value the type variable resolves to. Graversen et al. [2024] solve this problem with the $\mathtt{AmI}$ construct, executing $[\![C[\alpha \mapsto L]]\!]_L$ when the locations match and $[\![C]\!]_L$ when they do not. When the locations match, this projection is clearly correct. When they do not, the projection is correct because EPP treats location variables as abstract identifiers that are equal only to themselves, meaning $[\![C]\!]_L$ will behave as though $\alpha \neq L$.

Location *set* abstraction generalizes this idea. Replacing $\mathtt{AmI}$ with $\mathtt{AmI}\in$ is straightforward, as is $[\![C]\!]_L$ treating α as an abstract set where $L \notin \alpha$. The other branch, where α resolves to a set containing L, is more subtle. This projection of C must treat α as though it contains L, *without changing the meaning of* α. We accomplish this by explicitly adding L to the set in this branch and using $[\![C[\alpha \mapsto \{L\} \cup \alpha]]\!]_L$. The symbolic containment relation recognizes that $L \in \{L\} \cup \rho$ for any ρ, including the variable α, so the projection will properly treat L as being in the set. Moreover, despite symbolically changing the set, the operational semantics remain correct. This branch only executes when α resolves to some ρ where $L \in \rho$, in which case $\{L\} \cup \rho = \rho$.

Finally, type-let expressions combine a structurally recursive projection, also used in local-let expressions, with the location-based conditionals of type abstraction. For locations binding the type variable, the projection first computes the type, binds it, and then uses $\mathtt{AmI}$ to condition based on its value. For other locations, EPP simply sequences the head of the let with its body. Notably, the type system, to avoid dependency, does not prevent locations outside of ρ from referencing α. Any such invalid reference would leave α unbound in the body of the let, so EPP checks that $\alpha \notin \mathrm{fv}([\![C_2]\!]_L)$. If both α is free and $L \notin \rho$, there is no correct way for L to execute its part of C_2, so the projection is undefined. The projection for type-let expressions where α has kind $*_s$ are nearly identical, with $\mathtt{AmI}\in$ in place of $\mathtt{AmI}$ and the approach described above for type abstractions when $L \in \rho$.

This EPP definition gives the network program corresponding to one location. Combining these programs into a parallel system (see Section 5) gives an executable interpretation of the entire choreography. Specifically, we lift EPP point-wise to a finite set of locations $\mathfrak{L} \subseteq \mathcal{L}$, defining $[\![C]\!]_{\mathfrak{L}} = \|_{L\in\mathfrak{L}}\ (L \triangleright [\![C]\!]_L)$, which also requires $[\![C]\!]_L$ to be defined for all $L \in \mathfrak{L}$.

6.3 Bisimulation Relation

Next, we will examine the relationship between our choreographic operational semantics and the semantics given by EPP, with two main goals in mind. Firstly, to exhibit a bisimulation between these two semantics to justify the correctness of EPP. Secondly, to give a guarantee of deadlock freedom by design for compiled systems.

To construct a bisimulation, we must choose which systems are related to a given choreography. The natural choice is to say that a choreography C is related only to its projection $[\![C]\!]_{\mathfrak{L}}$. However, this relation is too strict. When a choreography branches, the branch which is not taken is discarded, but in the projected system, that branch may be preserved by locations waiting on selection messages. For instance, consider the following example where C_1 reduces to C_2.

$$\begin{tikzcd}
C_1 = \begin{array}{l} \text{if A.true} \\ \text{then A}[\mathbf{L}] \rightsquigarrow \text{B} ; \text{B}.1 \\ \text{else A}[\mathbf{R}] \rightsquigarrow \text{B} ; \text{B}.2 \end{array} \arrow[r, Rightarrow, "\text{if}_{\text{A}}\ \text{true}", "c"'] \arrow[d, mapsto, "{[\![\cdot]\!]_{\text{B}}}"'] & \text{A}[\mathbf{L}] \rightsquigarrow \text{B} ; \text{B}.1 = C_2 \arrow[d, mapsto, "{[\![\cdot]\!]_{\text{B}}}"] \\
{[\![C_1]\!]_{\text{B}}} = \begin{array}{l} \texttt{allow A choice} \\ \mid \mathbf{L} \Rightarrow \texttt{ret}(1) \\ \mid \mathbf{R} \Rightarrow \texttt{ret}(2) \end{array} \arrow[r, dashed, no head, "\neq"] & \begin{array}{l} \texttt{allow A choice} \\ \mid \mathbf{L} \Rightarrow \texttt{ret}(1) \end{array} = [\![C_2]\!]_{\text{B}}
\end{tikzcd}$$

Once C_1 takes the left branch, A can never send the selection message **R** to B. While this makes sense from A's perspective, it does not make sense for B, whose projected program has discarded the **R** branch without receiving any input from A. We expect that choreographic steps will only affect the projection of locations involved in the step.

An additional wrinkle stems from EPP's use of the collapsing sequencing function $E_1 \fatsemi E_2$: local programs (in an arbitrary subexpression) which resolve to a value after a substitution may be removed from the projected program, as in the following example.

$$\begin{tikzcd}
C_1 = \begin{array}{l} \text{let } \{\text{A}, \text{B}\}.x \coloneqq \{\text{A}, \text{B}\}.1 \\ \text{in fun } F(X) \coloneqq \\ \quad \text{let A}.y \coloneqq \{\text{A}, \text{B}\}.x \text{ in A}.2 \end{array} \arrow[r, Rightarrow, "\text{let } \{\text{A}, \text{B}\}.x \coloneqq 1", "c"'] \arrow[d, mapsto, "{[\![\cdot]\!]_{\text{B}}}"'] & \begin{array}{l} \text{fun } F(X) \coloneqq \\ \quad \text{let A}.y \coloneqq \{\text{A}, \text{B}\}.1 \text{ in A}.2 \end{array} = C_2 \arrow[d, mapsto, "{[\![\cdot]\!]_{\text{B}}}"] \\
{[\![C_1]\!]_{\text{B}}} = \begin{array}{l} \texttt{let } x \coloneqq \texttt{ret}(1) \texttt{ in} \\ \texttt{fun } F(X) \coloneqq \texttt{ret}(x) \ ; () \end{array} \arrow[r, Rightarrow, "/" marking] & \texttt{fun } F(X) \coloneqq () = [\![C_2]\!]_{\text{B}}
\end{tikzcd}$$

This reduction substitutes 1 for x in the body of the function. While $[\![C_1]\!]_{\text{B}}$ can analogously substitute x, the resultant program $\texttt{fun } F(X) \coloneqq \texttt{ret}(1) \ ; ()$ differs from $[\![C_2]\!]_{\text{B}}$, as the projection collapses the body of the function to $\texttt{ret}(1) \fatsemi () = ()$. As the mismatch is in the function body, there are no steps B can take to correct it.

We account for both types of mismatches using a relation $E_1 \preceq E_2$ indicating that E_1 may have discarded some unneeded code—either choices or sequenced values—that E_2 retains. Formally, it is the smallest partial order on network programs that is structurally compatible—it admits rules like $\frac{E_1 \preceq E_1' \quad E_2 \preceq E_2'}{E_1 ; E_2 \preceq E_1' ; E_2'}$—and admits the following three rules. The first two handle additional branches, while the third covers collapsing sequences.

$$\frac{E_1 \preceq E_1'}{\begin{array}{l} \texttt{allow } \ell \texttt{ choice} \\ \mid \mathbf{L} \Rightarrow E_1 \end{array} \preceq \begin{array}{l} \texttt{allow } \ell \texttt{ choice} \\ \mid \mathbf{L} \Rightarrow E_1' \\ \mid \mathbf{R} \Rightarrow E_2' \end{array}} \qquad \frac{E_2 \preceq E_2'}{\begin{array}{l} \texttt{allow } \ell \texttt{ choice} \\ \mid \mathbf{R} \Rightarrow E_2 \end{array} \preceq \begin{array}{l} \texttt{allow } \ell \texttt{ choice} \\ \mid \mathbf{L} \Rightarrow E_1' \\ \mid \mathbf{R} \Rightarrow E_2' \end{array}} \qquad \frac{E_1 \preceq E_2 \qquad \text{Val}(V)}{E_1 \preceq V \ ; E_2}$$

The relation also lifts point-wise to systems (with identical domains $\mathfrak{L}$):

$$\Pi_1 \preceq \Pi_2 \;=\; \forall L \in \mathfrak{L}.\, \Pi_1(L) \preceq \Pi_2(L).$$

By relaxing the bisimulation to relate a choreography C not only to $[\![C]\!]_{\mathfrak{L}}$, but to any system Π where $[\![C]\!]_{\mathfrak{L}} \preceq \Pi$, we avoid the problems described above. In the first example, $[\![C_2]\!]_{\mathsf{B}} \preceq [\![C_1]\!]_{\mathsf{B}}$, and in the second, $[\![C_1]\!]_{\mathsf{B}} \Longrightarrow E$ where $[\![C_2]\!]_{\mathsf{B}} \preceq E$, as needed.

6.4 Soundness, Completeness, and Deadlock Freedom

The above correspondence is sufficient to show that our choreographic and projected system semantics are equivalent and yield deadlock freedom of compiled systems as a result. To this end, we prove the choreographic semantics is sound and complete with respect to the projected system.

The theorems require that all *location names in* C, denoted $\mathrm{LN}(C)$, are included in the projected system: $\mathrm{LN}(C) \subseteq \mathfrak{L}$. This requirement formalizes the implicit assumption that all locations specified by the choreography are present in the system, lest, for instance, A be unable to compute $\mathsf{A}.2 \rightsquigarrow \mathsf{B}$ because B is not in the system. We similarly require $\mathfrak{L} \neq \varnothing$ to ensure *someone* is computing.

The completeness theorem is straightforward, and says the projected semantics simulate the choreographic semantics.

Theorem 3 (Completeness). *If* $\Gamma; \Delta; \Sigma \vdash C : \tau$, $C \Longrightarrow_c^n C'$, *and* $\mathrm{LN}(C) \subseteq \mathfrak{L} \neq \varnothing$, *then there is some* Π *and* $k \geq n$ *such that* $[\![C]\!]_{\mathfrak{L}} \Longrightarrow_S^k \Pi$ *and* $[\![C']\!]_{\mathfrak{L}} \preceq \Pi$.

The mechanized proof first shows the system can simulate a single step and then extends it to multiple steps by induction. It also uses the fact that, if $E_1 \preceq E_2$ and E_1 is the projection of a choreography, then E_1 and E_2 can make the same reductions, plus E_2 possibly taking administrative steps to remove extra un-collapsed semicolons.

We would like a soundness theorem that says the choreographic semantics can simulate the projected system, but there are some technical complications. First, after a single reduction in a projected system, the reduced system does not necessarily correspond to any choreography, and may instead be some intermediate state. The following example shows a case where the projected system $[\![C_1]\!]_{\{\mathsf{A},\mathsf{B}\}}$ can reduce to a state Π which no choreography projects to.

$$\begin{tikzcd}
C_1 = \{\mathsf{A},\mathsf{B}\}.(2+3) \arrow[rr, Rightarrow, "c"'] \arrow[dd, mapsto, "{[\![\cdot]\!]_{\{\mathsf{A},\mathsf{B}\}}}"'] & & \{\mathsf{A},\mathsf{B}\}.5 = C_2 \arrow[dd, mapsto, "{[\![\cdot]\!]_{\{\mathsf{A},\mathsf{B}\}}}"] \\
 & \Pi \arrow[d, phantom, "\shortparallel"] & \\
{[\![C_1]\!]_{\{\mathsf{A},\mathsf{B}\}}} = \begin{matrix} \mathsf{A} \triangleright \mathtt{ret}(2+3) \\ \mathsf{B} \triangleright \mathtt{ret}(2+3) \end{matrix} \arrow[r, Rightarrow, "S"'] & \begin{matrix} \mathsf{A} \triangleright \mathtt{ret}(5) \\ \mathsf{B} \triangleright \mathtt{ret}(2+3) \end{matrix} \arrow[r, Rightarrow, "S"'] & \begin{matrix} \mathsf{A} \triangleright \mathtt{ret}(5) \\ \mathsf{B} \triangleright \mathtt{ret}(5) \end{matrix} = {[\![C_2]\!]_{\{\mathsf{A},\mathsf{B}\}}}
\end{tikzcd}$$

To reach the system corresponding to the projection of C_2, Π must continue to progress. As a result, even our single-step soundness theorem must allow the system to continue to reduce until it reaches a point where the original choreography can match its progress.

The second technicality is a fundamental challenge stemming from the combination of nontermination and multiply-located local computations. In the system, each participant in a multiply-located local computation evaluates it independently. If one location performs such a step while another is blocked by an earlier infinite loop, however, the choreography will be unable to match the system. For instance, in $\mathsf{let}\ \mathsf{A}.x \coloneqq \mathsf{A}.\mathsf{loop}\ \mathsf{in}\ \{\mathsf{A},\mathsf{B}\}.(1+1)$ the only possible choreographic step is to run the infinite loop at A, but in the projected system, B may reduce the local expression $1 + 1$ to 2.

To avoid such situations, our soundness theorem only holds for choreographies C where all local computations terminate. Specifically, if $C \Longrightarrow_c^* C'$ and $\rho.e$ is in an evaluation position of C', then e

must terminate. Importantly, this condition does not require the local *language* to be terminating, only the specific local computations that execute in C.

With these two caveats in place, we can now state a single-step version of soundness.

Proposition 1 (Single-Step Soundness). *If* $\vdash C : \tau$, $[\![C]\!]_{\mathfrak{L}} \Longrightarrow_S \Pi$, $\mathrm{LN}(C) \subseteq \mathfrak{L} \neq \emptyset$, *and all local programs that* C *executes terminate, then for some* Π' *and* C', $\Pi \Longrightarrow_S^* \Pi'$, $C \Longrightarrow_c^+ C'$, *and* $[\![C']\!]_{\mathfrak{L}} \preceq \Pi'$.

Extending this result to simulate multiple system steps with a naïve induction runs into another problem: the steps that we wish to simulate and the extra—possibly different—steps needed to ensure the system is not in an intermediate state are independent reduction sequences. To reduce the endpoints of these diverging paths to a common system, we employ a confluence theorem, which critically relies on confluence of the local language.

Proposition 2 (System Confluence). *If* $\Pi_1 \Longrightarrow_S^* \Pi_2$ *and* $\Pi_1 \Longrightarrow_S^* \Pi_3$, *then there is some* Π_4 *such that* $\Pi_2 \Longrightarrow_S^* \Pi_4$ *and* $\Pi_3 \Longrightarrow_S^* \Pi_4$.

Combining confluence with Proposition 1 is enough to prove soundness beginning with any number of system steps.

Theorem 4 (Soundness). *If* $\vdash C : \tau$, $[\![C]\!]_{\mathfrak{L}} \Longrightarrow_S^* \Pi$, $\mathrm{LN}(C) \subseteq \mathfrak{L} \neq \emptyset$, *and all local programs that* C *executes terminate, then for some* Π' *and* C', $\Pi \Longrightarrow_S^* \Pi'$, $C \Longrightarrow_c^* C'$, *and* $[\![C']\!]_{\mathfrak{L}} \preceq \Pi'$.

Having demonstrated the correspondence between the choreographic and system semantics, we can now prove deadlock-freedom. Although bisimulation only holds when all executed local programs terminate, deadlock-freedom holds for all projected systems, even in the presence of non-terminating local computations. By combining the soundness of the type system (Theorems 1 and 2) and the completeness of EPP (Theorem 3), we prove that the system either enters an infinite loop for some location or terminates in a value for all locations, which is sufficient.

Theorem 5 (Deadlock Freedom by Design). *If* $\vdash C : \tau$, $[\![C]\!]_{\mathfrak{L}} \Longrightarrow_S^* \Pi$, *and* $\mathrm{LN}(C) \subseteq \mathfrak{L}$, *then either every location in* Π *maps to a value, or there is some* Π' *such that* $\Pi \Longrightarrow_S \Pi'$.

7 Rocq Development

We now discuss some details of our Rocq formalization and how they differ from the presentation above. For clarity, we presented λQC in a standard style with named variables in the paper, but for ease of development, the Rocq code uses a nameless style with de Bruijn indices for both program and type variables. As in Pirouette [Hirsch and Garg 2022], the local language must also use de Bruijn indices, and provide common guarantees about substitution.

This change also forces a different handling of location- and location-set–variable substitution in the Rocq code. As described in Section 4.2.1, care must be taken when performing location substitution to avoid variable capture between the namespaces of different locations. The Rocq encoding, however, avoids this issue by not separating namespaces by location. Instead, it treats all local variables uniformly as part of a single global namespace of de Bruijn indices. With this formulation, no capture is possible upon substituting location variables when using the standard capture-avoiding definition of location substitution.

Finally, our formalization proves deadlock freedom in two pieces. The first uses bisimulation (Theorems 3 and 4 together) to prove deadlock-freedom when all local programs that execute terminate. The second assumes the presence of a non-terminating local computation and proves deadlock freedom directly. The proof of Theorem 5, which does not condition on local (non-)termination is immediate by combining these, but assumes the law of the excluded middle (LEM) to differentiate between the cases. We make the LEM a premise to this theorem to explicitly indicate the use of a non-constructive assumption.

8 Related Work

As mentioned in Section 1, choreographic programming has seen substantial recent development. This paper fits into the emerging paradigm of *functional* choreographic programming with process polymorphism, originally proposed by Graversen et al. [2024]. Choreographic programming as a whole arises from concurrency theory and the study of π-calculi. We discuss each of these in turn.

8.1 Functional Choreographic Programming

Choreographic programming was crystallized as an independent paradigm in 2013 by the work of Carbone and Montesi [2013], especially Montesi's Ph.D. thesis [Montesi 2013]. For the next 10 years, it advanced, but remained tied to a lower-order model of computing [see, e.g., Carbone et al. 2014; Cruz-Filipe and Montesi 2017a,b; Cruz-Filipe et al. 2018; Lanese et al. 2013].

That bind was broken in 2022 by two independent developments: Pirouette [Hirsch and Garg 2022] and Chorλ [Cruz-Filipe et al. 2022]. These works combine the primitives of choreographic programming with λ calculi, allowing for (sequential) composition of choreographies and program reuse. However, neither support any type of polymorphism or multiply-located values.

λ_{QC} is based on Pirouette, inheriting many of its features including a separate language of messages (the local language) and out-of-order semantics that require all participants to synchronize on choreographic function applications. Chorλ, by contrast, combines the languages of choreographies and messages. It originally had a strictly sequential semantics [Cruz-Filipe et al. 2022], but recent work showed how to give it an out-of-order semantics *without* global synchronization [Cruz-Filipe et al. 2023]. These semantics use commuting conversions, a set of semantics-preserving rewrite rules such as $(\lambda X.\, C_1)\ C_2\ C_3 \Rrightarrow (\lambda X.\, C_1\ C_3)\ C_2$, and allow reductions inside function bodies. However, commuting conversions are fragile. For instance, they fail to preserve types in the presence of named recursive functions, like those in λ_{QC}. That is, the hypothetical rewrite $(\mathsf{fun}\, F(X) \coloneqq C_1)\ C_2\ C_3 \Rrightarrow (\mathsf{fun}\, F(X) \coloneqq C_1\ C_3)\ C_2$ changes the type of F, which is bound in C_1. Polymorphism and recursive types produce similar difficulties.

We thus adopt the more restrictive semantics of Pirouette, and leave finding an appropriate semantics without global synchronization as future work. Importantly, nothing in this paper relies on this global synchronization except for the *statement* of EPP correctness (Section 6.4). We believe an appropriate statement of EPP correctness could support a semantics without this synchronization, but identifying such a statement requires significant research in its own right.

8.2 Process Polymorphism

More recently, Graversen et al. [2024] extended Chorλ with process polymorphism, allowing choreographies to abstract over (the identity of) their participant processes. This extension, PolyChorλ, introduced the process abstraction and shows how to project it by adding the "AmI" construct to the target language. PolyChorλ is based on System F_ω, and includes quantification over processes, types, and types of higher-order kinds, but does not include quantification over *sets* of processes. Their work also does not attempt to define an out-of-order semantics, and instead settles for a sequential, call-by-value semantics for its choreographies.

Bates et al. [2025] separately introduced the idea of *census polymorphism*, which allows choreographies to abstract over the quantity and identity of their participants. Their work presents extensive real-world uses cases for multiply-located values and abstraction over sets of processes and highlights several practical implementations (e.g., in MultiChor, ChoRus, and ChoreoTS). However, the formal model is limited and lacks deadlock-freedom guarantees in the presence of polymorphism.

Our work not only supports first-class location set polymorphism and multiply-located values (missing from PolyChorλ) and a deadlock freedom guarantee (missing from the work of Bates et al.), but we mechanically verify all results in Rocq. To our knowledge, the only prior mechanizations of choreographic results concern Pirouette [Hirsch and Garg 2022] and simpler lower-order systems [Cruz-Filipe et al. 2021a,b], meaning this work contains the first formalization of choreographic process polymorphism or of multiply-located values, let alone λQC's full suite of features.

8.3 Higher-Order Communication

Another important aspect of concurrent systems is *higher-order communication*: communicating a channel (or channel name) over a channel. This feature is critical to model systems with a dynamic communication topology, and can be expressed natively in many untyped process calculi such as the π-calculus and its higher-order variant the HOπ-calculus [Sangiorgi 1993]. Previous work proving deadlock-freedom for concurrent languages with higher-order communication has relied on advanced typing regimes, like higher-order session types [see, e.g., Costa et al. 2022; Mostrous and Yoshida 2007; Poças et al. 2023], that are highly complex and impose significant restrictions on communication patterns, like requiring them to be from a regular or context-free grammar.

PolyChorλ supports a restricted form of higher-order communication using *delegation*, where one process can request that another perform an interaction on their behalf. In particular, they allow processes to communicate entire choreographies, which may themselves contain communication. However, the type system prevents delegated computation from containing unresolved location variables, limiting the expressive power.

Computing and sending first-class location names is a distinct form of higher-order communication from both π-calculus channels and PolyChorλ's delegation. Channels represent a limited capacity to perform a specific action, like sending one value, while choreographic process names are identifiers of participants who may be asked, by name, to perform many actions. Meanwhile, providing first-class data and location names to choreographic functions and type abstractions replicate much of the functionality of delegation, but must be specified manually.

9 Conclusion

This work presented the choreographic Quick Change calculus, λQC, a choreographic calculus supporting process (set) and type polymorphism, algebraic and recursive data types, multiply-located values, and first-class location names. While prior choreographic languages implement some of the first three features, none support all three, and λQC is the first to support first-class location names in a typed choreography.

We showed how to integrate polymorphism over types, processes, and sets of processes using a System F-like type system. The λQC type system simultaneously allows first-class treatment of (sets of) location names and considers locations type-level values. Yet it still avoids dependency by restricting the types of computations which use these locations to not depend on values. To prevent deadlocks, λQC statically ensures that each location knows if it has been selected by any dynamic location computation.

We also showed how to project the constructs in λQC to a network language that faithfully models the concurrent execution of a multi-party system, and we proved the classic choreographic result: all systems projected from well-typed choreographies are deadlock-free. We mechanically verified proofs of this result as well as others in the paper, providing the first mechanized formalization for any choreography supporting any form of process polymorphism or multiply-located values.

While there is still a wealth of future research in expanding and refining the features of our calculus, even now, it is able to facilitate practical and safe programming of concurrent systems. Our paradigmatic example of a distributed thread pool with a load balancer can be concisely written in λQC, yet is far beyond the capabilities of prior typed choreographic languages.

Acknowledgments

We would like to thank Rahul Krishnan for assistance in proofreading.

References

Mako Bates, Shun Kashiwa, Syed Jafri, Gan Shen, Lindsey Kuper, and Joseph P. Near. 2025. Efficient, Portable, Census-Polymorphic Choreographic Programming. In *Programming Languages Design and Implementation (PLDI)*. To Appear.

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

Diana Costa, Andreia Mordido, Diogo Poças, and Vasco T. Vasconcelos. 2022. Higher-order Context-free Session Types in System F. *Electronic Proceedings in Theoretical Computer Science* 356 (March 2022), 24–35. https://doi.org/10.4204/eptcs.356.3

Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. 2022. Functional Choreographic Programming. In *Theoretical Aspects of Computing – ICTAC 2022: 19th International Colloquium, Tbilisi, Georgia, September 27–29, 2022, Proceedings* (Tbilisi, Georgia). Springer-Verlag, Berlin, Heidelberg, 212–237. https://doi.org/10.1007/978-3-031-17715-6_15

Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. 2023. Modular Compilation for Higher-Order Functional Choreographies. In *European Conference on Object-Oriented Programming (ECOOP)*. https://doi.org/10.4230/LIPIcs.ECOOP.2023.7

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 Peresotti. 2018. Communications in Choreographies, Revisited. In *Symposium on Applied Computing (SAC)*. https://doi.org/10.1145/3167132.3167267

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

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

Tim Freeman and Frank Pfenning. 1991. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada) *(PLDI '91)*. Association for Computing Machinery, New York, NY, USA, 268–277. https://doi.org/10.1145/113445.113468

Eva Graversen, Andrew K. Hirsch, and Fabrizio Montesi. 2024. Alice or Bob?: Process Polymorphism in Choreographies. *Journal of Functional Programming (JFP)* 34 (2024), e1. https://doi.org/10.1017/S0956796823000114

Andrew K. Hirsch and Deepak Garg. 2022. Pirouette: Higher-Order Typed Functional Choreographies. In *Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3498684

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

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

Fabrizio Montesi. 2023. *Introduction to Choreographies*. Cambridge University Press. https://doi.org/10.1017/9781108981491

Dimitris Mostrous and Nobuko Yoshida. 2007. Two Session Typing Systems for Higher-Order Mobile Processes. In *Typed Lambda Calculi and Applications*, Simona Ronchi Della Rocca (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 321–335.

Benjamin C Pierce. 2002. *Types and Programming Languages*. Springer.

Diogo Poças, Diana Costa, Andreia Mordido, and Vasco T. Vasconcelos. 2023. System F^{μ}_{ω} with Context-free Session Types. In *Programming Languages and Systems: 32nd European Symposium on Programming, ESOP 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2023, Paris, France, April 22–27, 2023, Proceedings* (Paris, France). Springer-Verlag, Berlin, Heidelberg, 392–420. https://doi.org/10.1007/978-3-031-30044-8_15

Davide Sangiorgi. 1993. *Expressing mobility in process algebras: first-order and higher-order paradigms*. Ph. D. Dissertation. http://hdl.handle.net/1842/6569

Ian Sweet, David Darais, David Heath, Ryan Estes, William Harris, and Michael Hicks. 2023. Symphony: Expressive Secure Multiparty Computation with Coordination. In *The Art, Science, and Engineering of Programming (⟨Programming⟩)*.

Appendices

A Choreography Operational Semantics

A.1 Choreography Values

$$\begin{array}{lrcl} \text{Choreography Values} & V & ::= & \rho.v \mid \mathsf{fun}\ F(X) \coloneqq C \mid \Lambda\alpha::\kappa.\,C \\ & & \mid & (V_1, V_2) \mid \mathsf{inl}\ V \mid \mathsf{inr}\ V \mid \mathsf{fold}\ V \end{array}$$

A.2 Redices and Evaluation Contexts

$$\begin{array}{lrcl} \text{Messages} & m & ::= & v \mid d \\ \text{Redices} & R & ::= & \rho.(e_1 \longrightarrow e_2) \mid \mathsf{Fun}(R) \mid \mathsf{Arg}(R) \mid \mathsf{App} \mid \mathsf{TApp} \mid \mathsf{UnfoldFold} \\ & & \mid & \mathsf{PairL}(R) \mid \mathsf{PairR}(R) \mid \mathsf{FstPair} \mid \mathsf{SndPair} \mid \mathsf{CaseInl} \mid \mathsf{CaseInr} \\ & & \mid & \mathsf{let}\ \rho \coloneqq v \mid \mathsf{let}\ \rho \coloneqq t \mid \ell.m \rightsquigarrow \rho \mid \mathsf{if}_\rho\ \mathsf{true} \mid \mathsf{if}_\rho\ \mathsf{false} \\ \\ \text{Evaluation Contexts} & \eta & ::= & [\cdot]\ C \mid V\ [\cdot] \mid [\cdot]\ t \mid \mathsf{fold}\ [\cdot] \mid \mathsf{unfold}\ [\cdot] \\ & & \mid & ([\cdot], C) \mid (V, [\cdot]) \mid \mathsf{fst}\ [\cdot] \mid \mathsf{snd}\ [\cdot] \\ & & \mid & \mathsf{inl}\ [\cdot] \mid \mathsf{inr}\ [\cdot] \mid \mathsf{case}\ [\cdot]\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inl}\ Y \Rightarrow C_2) \\ & & \mid & \mathsf{let}\ \rho.x : t_e \coloneqq [\cdot]\ \mathsf{in}\ C_2 \mid \mathsf{let}\ \rho.\alpha :: \kappa \coloneqq [\cdot]\ \mathsf{in}\ C_2 \\ & & \mid & [\cdot]\ {\scriptstyle\{\ell\}}\!\rightsquigarrow \rho \mid \mathsf{if}_\rho\ [\cdot]\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \end{array}$$

A.3 Redex Blocked Locations

$$\mathrm{rloc}(\rho.(e_1 \longrightarrow e_2)) = \rho \qquad \mathrm{rloc}(\mathsf{Fun}(R)) = \mathrm{rloc}(R) \qquad \mathrm{rloc}(\mathsf{Arg}(R)) = \mathrm{rloc}(R) \qquad \mathrm{rloc}(\mathsf{App}) = \mathcal{L}$$

$$\mathrm{rloc}(\mathsf{TApp}) = \mathcal{L} \qquad \mathrm{rloc}(\mathsf{UnfoldFold}) = \mathcal{L} \qquad \mathrm{rloc}(\mathsf{PairL}(R)) = \mathrm{rloc}(R)$$

$$\mathrm{rloc}(\mathsf{PairR}(R)) = \mathrm{rloc}(R) \qquad \mathrm{rloc}(\mathsf{FstPair}) = \mathcal{L} \qquad \mathrm{rloc}(\mathsf{SndPair}) = \mathcal{L} \qquad \mathrm{rloc}(\mathsf{CaseInl}) = \mathcal{L}$$

$$\mathrm{rloc}(\mathsf{CaseInr}) = \mathcal{L} \qquad \mathrm{rloc}(\mathsf{let}\ \rho \coloneqq v) = \rho \qquad \mathrm{rloc}(\mathsf{let}\ \rho \coloneqq t) = \rho \qquad \mathrm{rloc}(\ell.m \rightsquigarrow \rho) = \{\ell\} \cup \rho$$

$$\mathrm{rloc}(\mathsf{if}_\rho\ \mathsf{true}) = \rho \qquad \mathrm{rloc}(\mathsf{if}_\rho\ \mathsf{false}) = \rho$$

A.4 Choreography Blocked Locations

$$\mathrm{cloc}(X) = \varnothing \qquad \mathrm{cloc}(\rho.e) = \rho \qquad \mathrm{cloc}(\mathsf{fun}\ F(X:\tau) \coloneqq C) = \varnothing \qquad \mathrm{cloc}(C_1\ C_2) = \mathcal{L}$$

$$\mathrm{cloc}(\Lambda\alpha::\kappa.\,C) = \varnothing \qquad \mathrm{cloc}(C\ t) = \mathcal{L} \qquad \mathrm{cloc}(\mathsf{fold}\ C) = \mathrm{cloc}(C) \qquad \mathrm{cloc}(\mathsf{unfold}\ C) = \mathcal{L}$$

$$\mathrm{cloc}((C_1, C_2)) = \mathrm{cloc}(C_1) \cup \mathrm{cloc}(C_2) \qquad \mathrm{cloc}(\mathsf{fst}\ C) = \mathcal{L} \qquad \mathrm{cloc}(\mathsf{snd}\ C) = \mathcal{L}$$

$$\mathrm{cloc}(\mathsf{inl}\ C) = \mathrm{cloc}(C) \qquad \mathrm{cloc}(\mathsf{inr}\ C) = \mathrm{cloc}(C)$$

$$\mathrm{cloc}(\mathsf{case}\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inl}\ Y \Rightarrow C_2)) = \mathcal{L}$$

$$\mathrm{cloc}(\mathsf{let}\ \rho.x : t_e \coloneqq C_1\ \mathsf{in}\ C_2) = \rho \cup \mathrm{cloc}(C_1) \cup \mathrm{cloc}(C_2)$$

$$\mathrm{cloc}(\mathsf{let}\ \rho.\alpha :: \kappa \coloneqq C_1\ \mathsf{in}\ C_2) = \rho \cup \mathrm{cloc}(C_1) \cup \mathrm{cloc}(C_2) \qquad \mathrm{cloc}(C\ {\scriptstyle\{\ell\}}\!\rightsquigarrow \rho) = \{\ell\} \cup \rho \cup \mathrm{cloc}(C)$$

$$\mathrm{cloc}(\ell[d] \rightsquigarrow \rho\ ;\ C) = \{\ell\} \cup \rho \cup \mathrm{cloc}(C)$$

$$\mathrm{cloc}(\mathsf{if}_\rho\ C\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) = \rho \cup \mathrm{cloc}(C) \cup \mathrm{cloc}(C_1) \cup \mathrm{cloc}(C_2)$$

A.5 Redex for an Evaluation Context

If η is an evaluation context and R is a redex, we define $\eta[R]$ to be the redex which corresponds to making the reduction given by R in the context η.

$$([\cdot]\ C)[R] = \mathsf{Fun}(R) \qquad (V\ [\cdot])[R] = \mathsf{Arg}(R) \qquad ([\cdot]\ t)[R] = R \qquad (\mathsf{fold}\ [\cdot])[R] = R$$

$$(\mathsf{unfold}\ [\cdot])[R] = R \qquad ([\cdot], C)[R] = \mathsf{PairL}(R) \qquad (V, [\cdot])[R] = \mathsf{PairR}(R) \qquad (\mathsf{fst}\ [\cdot])[R] = R$$

$$(\mathsf{snd}\ [\cdot])[R] = R \qquad (\mathsf{inl}\ [\cdot])[R] = R \qquad (\mathsf{inr}\ [\cdot])[R] = R$$

$$(\mathsf{case}\ [\cdot]\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inl}\ Y \Rightarrow C_2))[R] = R \qquad ([\cdot]\ \overset{\{\ell\}}{\rightsquigarrow}\ \rho)[R] = R$$

$$(\mathsf{let}\ \rho.x : t_e \coloneqq [\cdot]\ \mathsf{in}\ C)[R] = R \qquad (\mathsf{let}\ \alpha :: \kappa \coloneqq [\cdot]\ \mathsf{in}\ C)[R] = R \qquad (\mathsf{if}_\rho\ [\cdot]\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2)[R] = R$$

A.6 Location Set Relations

Here we define precisely the containment $\ell \in \rho$, disjointness $\rho_1 \cap \rho_2 = \varnothing$, and subset $\rho_1 \subseteq \rho_2$ relations, with special care given to how they are defined when the types in question are non-ground. In particular, we define two versions of containment: *necessary* containment $\Box(\ell \in \rho)$, and *possible* containment $\Diamond(\ell \in \rho)$. The un-annotated containment relation is treated as necessary containment ($\ell \in \rho = \Box(\ell \in \rho)$), and the disjointness relation (to be interpreted as necessary disjointness) is defined as follows:

$$(\rho_1 \cap \rho_2 = \varnothing) = \forall \ell.\, \neg(\Diamond(\ell \in \rho_1) \land \Diamond(\ell \in \rho_2)).$$

The difference between the two versions of containment is that location sets which are variables (or a singleton of a variable) can possibly contain any location, while variable location sets do not necessarily contain any location. Note here that the metavariable ℓ stands for either a type variable α or a concrete location $L \in \mathcal{L}$, and the metavariable ρ stands for any location set, including possibly a type variable.

$$\frac{}{\Box(\ell \in \{\ell\})} \qquad \frac{\Box(\ell \in \rho_1)}{\Box(\ell \in \rho_1 \cup \rho_2)} \qquad \frac{\Box(\ell \in \rho_2)}{\Box(\ell \in \rho_1 \cup \rho_2)}$$

$$\frac{}{\Diamond(\ell \in \alpha)} \qquad \frac{}{\Diamond(\ell \in \{\alpha\})} \qquad \frac{}{\Diamond(L \in \{L\})} \qquad \frac{\Diamond(\ell \in \rho_1)}{\Diamond(\ell \in \rho_1 \cup \rho_2)} \qquad \frac{\Diamond(\ell \in \rho_2)}{\Diamond(\ell \in \rho_1 \cup \rho_2)}$$

To define the (necessarily a) subset relation, we first note that we cannot use the naïve definition in terms of the necessary containment relation. That is, $\forall \ell.\, \Box(\ell \in \rho_1) \Rightarrow \Box(\ell \in \rho_2)$ would not serve as a correct definition for $\rho_1 \subseteq \rho_2$ in the presence of type variables. This is because $\neg\Box(\ell \in \alpha)$ for every location ℓ, so with this definition we would have that $\alpha \subseteq \rho$ for every set ρ. The subset relation (and relations of $\Box$ modality in general) should be preserved under substitution, but our example shows that this is not the case when using the naïve definition. Instead, the subset relation must be defined inductively as follows.

$$\frac{}{\varnothing \subseteq \rho} \qquad \frac{}{\rho \subseteq \rho} \qquad \frac{\rho \subseteq \rho_1}{\rho \subseteq \rho_1 \cup \rho_2} \qquad \frac{\rho \subseteq \rho_2}{\rho \subseteq \rho_1 \cup \rho_2} \qquad \frac{\Box(\ell \in \rho)}{\{\ell\} \subseteq \rho} \qquad \frac{\rho_1 \subseteq \rho \qquad \rho_2 \subseteq \rho}{\rho_1 \cup \rho_2 \subseteq \rho}$$

A.7 Choreography Operational Semantics

[C-Ctx]
$$\frac{C_1 \overset{R}{\Longrightarrow}_c C_2}{\eta[C_1] \overset{\eta[R]}{\Longrightarrow}_c \eta[C_2]}$$

[C-Done]
$$\frac{e_1 \longrightarrow e_2 \qquad \mathrm{fv}(\rho) = \varnothing}{\rho.e_1 \xRightarrow{\rho.(e_1 \to e_2)}_c \rho.e_2}$$

[C-App]
$$\frac{f = \mathsf{fun}\, F(X) \coloneqq C \qquad \mathrm{Val}(V)}{f\ V \xRightarrow{\mathsf{App}}_c C[F \mapsto f, X \mapsto V]}$$

[C-TApp]
$$\frac{}{(\Lambda\alpha :: \kappa.\, C)\ t \xRightarrow{\mathsf{TApp}}_c C[\alpha \mapsto t]}$$

[C-UnfoldFold]
$$\frac{\mathrm{Val}(V)}{\mathsf{unfold}\ (\mathsf{fold}\ V) \xRightarrow{\mathsf{UnfoldFold}}_c V}$$

[C-FstPair]
$$\frac{\mathrm{Val}(V_1) \qquad \mathrm{Val}(V_2)}{\mathsf{fst}\ (V_1, V_2) \xRightarrow{\mathsf{FstPair}}_c V_1}$$

[C-SndPair]
$$\frac{\mathrm{Val}(V_1) \qquad \mathrm{Val}(V_2)}{\mathsf{snd}\ (V_1, V_2) \xRightarrow{\mathsf{SndPair}}_c V_2}$$

[C-CaseInl]
$$\frac{\mathrm{Val}(V)}{\mathsf{case}\ (\mathsf{inl}\ V)\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inl}\ Y \Rightarrow C_2) \xRightarrow{\mathsf{CaseInl}}_c C_1[X \mapsto V]}$$

[C-CaseInr]
$$\frac{\mathrm{Val}(V)}{\mathsf{case}\ (\mathsf{inr}\ V)\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inl}\ Y \Rightarrow C_2) \xRightarrow{\mathsf{CaseInr}}_c C_2[X \mapsto V]}$$

[C-LetV]
$$\frac{\mathrm{Val}(v) \qquad \mathrm{fv}(\rho) = \varnothing}{\mathsf{let}\ \rho.x : t_e \coloneqq \rho'.v\ \mathsf{in}\ C \xRightarrow{\mathsf{let}\ \rho \coloneqq v}_c C[\rho | x \mapsto v]}$$

[C-LetI]
$$\frac{C_2 \overset{R}{\Longrightarrow}_c C_2' \qquad \mathrm{cloc}(C_1) \cap \mathrm{rloc}(R) = \varnothing \qquad \rho \cap \mathrm{rloc}(R) = \varnothing}{\mathsf{let}\ \rho.x : t_e \coloneqq C_1\ \mathsf{in}\ C_2 \overset{R}{\Longrightarrow}_c \mathsf{let}\ \rho.x : t_e \coloneqq C_1\ \mathsf{in}\ C_2'}$$

[C-TyLetV]
$$\frac{\mathrm{Val}(\lceil t \rfloor) \qquad \mathrm{fv}(\rho) = \varnothing}{\mathsf{let}\ \rho.\alpha :: \kappa \coloneqq \rho'.\lceil t \rfloor\ \mathsf{in}\ C \xRightarrow{\mathsf{let}\ \rho.\alpha \coloneqq t}_c C[\alpha \mapsto t]}$$

[C-TyLetI]
$$\frac{C_2 \overset{R}{\Longrightarrow}_c C_2' \qquad \mathrm{cloc}(C_1) \cap \mathrm{rloc}(R) = \varnothing \qquad \rho \cap \mathrm{rloc}(R) = \varnothing}{\mathsf{let}\ \rho.\alpha :: \kappa \coloneqq C_1\ \mathsf{in}\ C_2 \overset{R}{\Longrightarrow}_c \mathsf{let}\ \rho.\alpha :: \kappa \coloneqq C_1\ \mathsf{in}\ C_2'}$$

[C-SendV]
$$\frac{\mathrm{Val}(v) \qquad L_1 \in \rho_1 \qquad \mathrm{fv}(\rho_2) = \varnothing}{\rho_1.v\ \{L_1\}\!\rightsquigarrow \rho_2 \xRightarrow{L_1.v \rightsquigarrow \rho_2}_c (\rho_1 \cup \rho_2).v}$$

[C-Sync]
$$\frac{\mathrm{fv}(\rho) = \varnothing}{L[d] \rightsquigarrow \rho\ ;\ C \xRightarrow{L.d \rightsquigarrow \rho}_c C}$$

[C-SyncI]
$$\frac{C \overset{R}{\Longrightarrow}_c C' \qquad \ell \notin \mathrm{rloc}(R) \qquad \rho \cap \mathrm{rloc}(R) = \varnothing}{\ell[d] \leadsto \rho \,;\, C \overset{R}{\Longrightarrow}_c \ell[d] \leadsto \rho \,;\, C'}$$

[C-IfT]
$$\frac{\mathrm{fv}(\rho) = \varnothing}{\text{if}_\rho\ \rho.\text{true then } C_1 \text{ else } C_2 \xRightarrow{\text{if}_\rho\ \text{true}}_c C_1}$$

[C-IfF]
$$\frac{\mathrm{fv}(\rho) = \varnothing}{\text{if}_\rho\ \rho.\text{false then } C_1 \text{ else } C_2 \xRightarrow{\text{if}_\rho\ \text{false}}_c C_2}$$

[C-IfI]
$$\frac{\begin{array}{cc} C_1 \overset{R}{\Longrightarrow}_c C_1' & C_2 \overset{R}{\Longrightarrow}_c C_2' \\ \mathrm{cloc}(C) \cap \mathrm{rloc}(R) = \varnothing & \rho \cap \mathrm{rloc}(R) = \varnothing \end{array}}{\text{if}_\rho\ C \text{ then } C_1 \text{ else } C_2 \overset{R}{\Longrightarrow}_c \text{if}_\rho\ C \text{ then } C_1' \text{ else } C_2'}$$

[C-AppI]
$$\frac{C_2 \overset{R}{\Longrightarrow}_c C_2' \qquad \mathrm{cloc}(C_1) \cap \mathrm{rloc}(R) = \varnothing}{C_1\ C_2 \overset{R}{\Longrightarrow}_c C_1\ C_2'}$$

[C-PairI]
$$\frac{C_2 \overset{R}{\Longrightarrow}_c C_2' \qquad \mathrm{cloc}(C_1) \cap \mathrm{rloc}(R) = \varnothing}{(C_1, C_2) \overset{R}{\Longrightarrow}_c (C_1, C_2')}$$

B Static Semantics

B.1 λqc Kinding System

[K-Var]
$$\frac{\alpha :: \kappa \in \Gamma}{\Gamma \vdash \alpha :: \kappa}$$

[K-Local]
$$\frac{\Gamma \Vdash t_e}{\Gamma \vdash t_e :: *_e}$$

[K-At]
$$\frac{\Gamma \vdash t_e :: *_e \qquad \Gamma \vdash \rho :: *_s}{\Gamma \vdash t_e @ \rho :: *}$$

[K-Arrow]
$$\frac{\Gamma \vdash \tau_1 :: * \qquad \Gamma \vdash \tau_2 :: *}{\Gamma \vdash \tau_1 \to \tau_2 :: *}$$

[K-All]
$$\frac{\Gamma, \alpha :: \kappa \vdash \tau :: *}{\Gamma \vdash \forall \alpha :: \kappa . \tau :: *}$$

[K-Prod]
$$\frac{\Gamma \vdash \tau_1 :: * \qquad \Gamma \vdash \tau_2 :: *}{\Gamma \vdash \tau_1 \times \tau_2 :: *}$$

[K-Sum]
$$\frac{\Gamma \vdash \tau_1 :: * \qquad \Gamma \vdash \tau_2 :: *}{\Gamma \vdash \tau_1 + \tau_2 :: *}$$

[K-Rec]
$$\frac{\Gamma, \alpha :: * \vdash \tau :: *}{\Gamma \vdash \mu \alpha . \tau :: *}$$

[K-Loc]
$$\frac{L \in \mathcal{L}}{\Gamma \vdash L :: *_\ell}$$

[K-Sng]
$$\frac{\Gamma \vdash \ell :: *_\ell}{\Gamma \vdash \{\ell\} :: *_s}$$

[K-Union]
$$\frac{\Gamma \vdash \rho_1 :: *_s \qquad \Gamma \vdash \rho_2 :: *_s}{\Gamma \vdash \rho_1 \cup \rho_2 :: *_s}$$

[WF-EmpCtx]
$$\frac{}{\Gamma \vdash \cdot}$$

[WF-AddCtx]
$$\frac{\begin{array}{cc} X \notin \Delta & \Gamma \vdash \Delta \\ \multicolumn{2}{c}{\Gamma \vdash \tau :: *} \end{array}}{\Gamma \vdash \Delta, X : \tau}$$

[WF-AddLocalCtx]
$$\frac{\begin{array}{cc} x \notin \Sigma & \Gamma \vdash \Sigma \\ \Gamma \vdash t_e :: *_e & \Gamma \vdash \rho :: *_s \end{array}}{\Gamma \vdash \Sigma, \rho.x : t_e}$$

B.2 λqc Type System

[T-Var]
$$\frac{\begin{array}{cc} \multicolumn{2}{c}{X : \tau \in \Delta} \\ \Gamma \vdash \Delta & \Gamma \vdash \Sigma \end{array}}{\Gamma; \Delta; \Sigma \vdash X : \tau}$$

[T-Done]
$$\frac{\begin{array}{cc} \Gamma \vdash \rho :: *_s & \Gamma; \Sigma|_\rho \Vdash e : t_e \\ \Gamma \vdash \Delta & \Gamma \vdash \Sigma \end{array}}{\Gamma; \Delta; \Sigma \vdash \rho.e : t_e @ \rho}$$

[T-Fun]
$$\frac{\Gamma; \Delta, F : \tau_1 \to \tau_2, X : \tau_1; \Sigma \vdash C : \tau_2}{\Gamma; \Delta; \Sigma \vdash \text{fun } F(X) \coloneqq C : \tau_1 \to \tau_2}$$

[T-App]
$$\frac{\Gamma; \Delta; \Sigma \vdash C_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash C_2 : \tau_1}{\Gamma \vdash C_1\ C_2 : \tau_2}$$

[T-Abs]
$$\frac{\Gamma, \alpha :: \kappa; \Delta; \Sigma \vdash C : \tau}{\Gamma; \Delta; \Sigma \vdash \Lambda \alpha :: \kappa . C : \forall \alpha :: \kappa . \tau}$$

[T-TApp]
$$\frac{\Gamma; \Delta; \Sigma \vdash C : \forall \alpha :: \kappa . \tau \qquad \Gamma \vdash t :: \kappa}{\Gamma; \Delta; \Sigma \vdash C\ t : \tau[\alpha \mapsto t]}$$

[T-Fold]
$$\frac{\Gamma; \Delta; \Sigma \vdash C : \tau[\alpha \mapsto \mu \alpha . \tau]}{\Gamma; \Delta; \Sigma \vdash \text{fold } C : \mu \alpha . \tau}$$

[T-Unfold]
$$\frac{\Gamma; \Delta; \Sigma \vdash C : \mu \alpha . \tau}{\Gamma; \Delta; \Sigma \vdash \text{unfold } C : \tau[\alpha \mapsto \mu \alpha . \tau]}$$

[T-Pair]
$$\frac{\Gamma;\Delta;\Sigma \vdash C_1 : \tau_1 \qquad \Gamma;\Delta;\Sigma \vdash C_2 : \tau_2}{\Gamma;\Delta;\Sigma \vdash (C_1, C_2) : \tau_1 \times \tau_2}$$

[T-Fst]
$$\frac{\Gamma;\Delta;\Sigma \vdash C : \tau_1 \times \tau_2}{\Gamma;\Delta;\Sigma \vdash \mathsf{fst}\ C : \tau_1}$$

[T-Snd]
$$\frac{\Gamma;\Delta;\Sigma \vdash C : \tau_1 \times \tau_2}{\Gamma;\Delta;\Sigma \vdash \mathsf{snd}\ C : \tau_2}$$

[T-Inl]
$$\frac{\Gamma;\Delta;\Sigma \vdash C : \tau_1 \qquad \Gamma \vdash \tau_2 :: *}{\Gamma;\Delta;\Sigma \vdash \mathsf{inl}\ C : \tau_1 + \tau_2}$$

[T-Inr]
$$\frac{\Gamma;\Delta;\Sigma \vdash C : \tau_2 \qquad \Gamma \vdash \tau_1 :: *}{\Gamma;\Delta;\Sigma \vdash \mathsf{inr}\ C : \tau_1 + \tau_2}$$

[T-Case]
$$\frac{\begin{array}{c}\Gamma;\Delta;\Sigma \vdash C : \tau_1 + \tau_2 \\ \Gamma;\Delta, X{:}\tau_1;\Sigma \vdash C_1 : \tau \qquad \Gamma;\Delta, Y{:}\tau_2;\Sigma \vdash C_2 : \tau\end{array}}{\Gamma;\Delta;\Sigma \vdash \mathsf{case}\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inl}\ Y \Rightarrow C_2) : \tau}$$

[T-LetLocal]
$$\frac{\begin{array}{c}\Gamma;\Delta;\Sigma \vdash C_1 : t_e@\rho_2 \qquad \rho_1 \subseteq \rho_2 \\ \Gamma;\Delta;\Sigma, \rho_1.x{:}t_e \vdash C_2 : \tau\end{array}}{\Gamma;\Delta;\Sigma \vdash \mathsf{let}\ \rho_1.x{:}t_e \coloneqq C_1\ \mathsf{in}\ C_2 : \tau}$$

[T-LetLoc]
$$\frac{\begin{array}{c}\Gamma;\Delta;\Sigma \vdash C_1 : \mathsf{loc}_{\rho_1}@\rho_3 \qquad \rho_1 \subseteq \rho_2 \subseteq \rho_3 \\ \Gamma \vdash \tau :: * \qquad \Gamma, \alpha :: *_\ell;\Delta;\Sigma \vdash C_2 : \tau\end{array}}{\Gamma;\Delta;\Sigma \vdash \mathsf{let}\ \rho_2.\alpha :: *_\ell \coloneqq C_1\ \mathsf{in}\ C_2 : \tau}$$

[T-LetLocSet]
$$\frac{\begin{array}{c}\Gamma;\Delta;\Sigma \vdash C_1 : \mathsf{locset}_{\rho_1}@\rho_3 \qquad \rho_1 \subseteq \rho_2 \subseteq \rho_3 \\ \Gamma \vdash \tau :: * \qquad \Gamma, \alpha :: *_s;\Delta;\Sigma \vdash C_2 : \tau\end{array}}{\Gamma;\Delta;\Sigma \vdash \mathsf{let}\ \rho_2.\alpha :: *_s \coloneqq C_1\ \mathsf{in}\ C_2 : \tau}$$

[T-LetLocalTy]
$$\frac{\begin{array}{c}\Gamma;\Delta;\Sigma \vdash C_1 : \mathsf{tyRep}@\rho_2 \qquad \rho_1 \subseteq \rho_2 \\ \Gamma \vdash \tau :: * \qquad \Gamma, \alpha :: *_e;\Delta;\Sigma \vdash C_2 : \tau\end{array}}{\Gamma;\Delta;\Sigma \vdash \mathsf{let}\ \rho_1.\alpha :: *_e \coloneqq C_1\ \mathsf{in}\ C_2 : \tau}$$

[T-Send]
$$\frac{\begin{array}{c}\Gamma;\Delta;\Sigma \vdash C : t_e@\rho_1 \\ \ell \in \rho_1 \qquad \Gamma \vdash \rho_2 :: *_s\end{array}}{\Gamma;\Delta;\Sigma \vdash C\ \{\ell\}\!\rightsquigarrow \rho_2 : t_e@(\rho_1 \cup \rho_2)}$$

[T-Sync]
$$\frac{\begin{array}{c}\Gamma \vdash \ell :: *_\ell \qquad \Gamma \vdash \rho :: *_s \\ \Gamma;\Delta;\Sigma \vdash C : \tau\end{array}}{\Gamma;\Delta;\Sigma \vdash \ell[d] \rightsquigarrow \rho\ ; C : \tau}$$

[T-If]
$$\frac{\begin{array}{c}\Gamma;\Delta;\Sigma \vdash C : \mathsf{bool}@\rho \\ \Gamma;\Delta;\Sigma \vdash C_1 : \tau \qquad \Gamma;\Delta;\Sigma \vdash C_2 : \tau\end{array}}{\Gamma;\Delta;\Sigma \vdash \mathsf{if}_\rho\ C\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 : \tau}$$

C Network Language

C.1 Network Language Expressions

$$\begin{array}{llcl}
\text{Network Program} & E & ::= & X \mid () \mid \mathtt{ret}(e) \mid E_1\ \mathtt{;}\ E_2 \\
 & & \mid & \mathtt{fun}\ F(X) \coloneqq E \mid E_1\ E_2 \mid \Lambda\alpha.\,E \mid E\ t \\
 & & \mid & (E_1, E_2) \mid \mathtt{fst}\ E \mid \mathtt{snd}\ E \\
 & & \mid & \mathtt{inl}\ E \mid \mathtt{inr}\ E \mid \mathtt{case}\ E\ \mathtt{of}\ (\mathtt{inl}\ X \Rightarrow E_1)\ (\mathtt{inr}\ Y \Rightarrow E_2) \\
 & & \mid & \mathtt{fold}\ E \mid \mathtt{unfold}\ E \\
 & & \mid & \mathtt{send}\ E\ \mathtt{to}\ \rho \mid \mathtt{recv\ from}\ \ell \\
 & & \mid & \mathtt{let}\ x \coloneqq E_1\ \mathtt{in}\ E_2 \mid \mathtt{let}\ \alpha :: \kappa \coloneqq E_1\ \mathtt{in}\ E_2 \\
 & & \mid & \mathtt{if}\ E\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2 \\
 & & \mid & \mathtt{choose}\ d\ \mathtt{for}\ \ell\ \mathtt{;}\ E \\
 & & \mid & \mathtt{allow}\ \ell\ \mathtt{choice}\ (\mathbf{L} \Rightarrow E_{1\perp})\ (\mathbf{R} \Rightarrow E_{2\perp}) \\
 & & \mid & \mathtt{AmI}{\in}\ \rho\ \mathtt{then}\ E_1\ \mathtt{else}\ E_2 \\
\text{Network Values} & V & ::= & () \mid \mathtt{ret}(v) \mid \mathtt{fun}\ F(X) \coloneqq E \mid \Lambda\alpha.\,E \\
 & & \mid & (V_1, V_2) \mid \mathtt{inl}\ V \mid \mathtt{inr}\ V \mid \mathtt{fold}\ V
\end{array}$$

C.2 Transition Labels and Evaluation Contexts

$$
\begin{array}{llll}
\text{Transition Labels} & l & ::= & \iota \mid \iota_{\text{sync}} \mid m \rightsquigarrow \rho \mid L.m \rightsquigarrow \\[1em]
\text{Evaluation Contexts} & \eta & ::= & [\cdot] \; ; E \mid [\cdot]\; E \mid V\; [\cdot] \mid [\cdot]\; t \mid \texttt{fold}\; [\cdot] \mid \texttt{unfold}\; [\cdot] \\
 & & \mid & ([\cdot], E) \mid (V, [\cdot]) \mid \texttt{fst}\; [\cdot] \mid \texttt{snd}\; [\cdot] \mid \texttt{inl}\; [\cdot] \mid \texttt{inr}\; [\cdot] \\
 & & \mid & \texttt{case}\; [\cdot]\; \texttt{of}\; (\texttt{inl}\; X \Rightarrow E_1)\; (\texttt{inr}\; Y \Rightarrow E_2) \\
 & & \mid & \texttt{send}\; [\cdot]\; \texttt{to}\; \rho \mid \texttt{let}\; x \coloneqq [\cdot]\; \texttt{in}\; E \mid \texttt{let}\; \alpha :: \kappa \coloneqq [\cdot]\; \texttt{in}\; E \\
 & & \mid & \texttt{if}\; [\cdot]\; \texttt{then}\; E_1\; \texttt{else}\; E_2
\end{array}
$$

C.3 Network Language Operational Semantics

$$\text{[N-Ctx]}\quad \frac{L \triangleright E_1 \overset{l}{\Longrightarrow} E_2}{L \triangleright \eta[E_1] \overset{l}{\Longrightarrow} \eta[E_2]} \qquad \text{[N-Ret]}\quad \frac{e_1 \longrightarrow e_2}{L \triangleright \texttt{ret}(e_1) \overset{\iota}{\Longrightarrow} \texttt{ret}(e_2)} \qquad \text{[N-Seq]}\quad \frac{\mathrm{Val}(V)}{L \triangleright V \; ; E \overset{\iota}{\Longrightarrow} E}$$

$$\text{[N-App]}\quad \frac{f = \texttt{fun}\; F(X) \coloneqq E \qquad \mathrm{Val}(V)}{L \triangleright f\; V \overset{\iota_{\text{sync}}}{\Longrightarrow} E[F \mapsto f, X \mapsto V]} \qquad \text{[N-TApp]}\quad \frac{}{L \triangleright (\Lambda \alpha :: \kappa.\, E)\; t \overset{\iota_{\text{sync}}}{\Longrightarrow} E[\alpha \mapsto t]} \qquad \text{[N-UnfoldFold]}\quad \frac{\mathrm{Val}(V)}{L \triangleright \texttt{unfold}\; (\texttt{fold}\; V) \overset{\iota_{\text{sync}}}{\Longrightarrow} V}$$

$$\text{[N-FstPair]}\quad \frac{\mathrm{Val}(V_1) \qquad \mathrm{Val}(V_2)}{L \triangleright \texttt{fst}\; (V_1, V_2) \overset{\iota_{\text{sync}}}{\Longrightarrow} V_1} \qquad \text{[N-SndPair]}\quad \frac{\mathrm{Val}(V_1) \qquad \mathrm{Val}(V_2)}{L \triangleright \texttt{snd}\; (V_1, V_2) \overset{\iota_{\text{sync}}}{\Longrightarrow} V_2}$$

$$\text{[N-CaseInl]}\quad \frac{\mathrm{Val}(V)}{L \triangleright \texttt{case}\; (\texttt{inl}\; V)\; \texttt{of}\; (\texttt{inl}\; X \Rightarrow E_1)\; (\texttt{inr}\; Y \Rightarrow E_2) \overset{\iota_{\text{sync}}}{\Longrightarrow} E_1[X \mapsto V]}$$

$$\text{[N-CaseInr]}\quad \frac{\mathrm{Val}(V)}{L \triangleright \texttt{case}\; (\texttt{inr}\; V)\; \texttt{of}\; (\texttt{inl}\; X \Rightarrow E_1)\; (\texttt{inr}\; Y \Rightarrow E_2) \overset{\iota_{\text{sync}}}{\Longrightarrow} E_2[X \mapsto V]}$$

$$\text{[N-Let]}\quad \frac{\mathrm{Val}(v)}{L \triangleright \texttt{let}\; x \coloneqq \texttt{ret}(v)\; \texttt{in}\; C \overset{\iota}{\Longrightarrow} C[x \mapsto v]} \qquad \text{[N-TyLet]}\quad \frac{\mathrm{Val}(\lceil t \rfloor)}{L \triangleright \texttt{let}\; \alpha :: \kappa \coloneqq \texttt{ret}(\lceil t \rfloor)\; \texttt{in}\; E \overset{\iota}{\Longrightarrow} E[\alpha \mapsto t]}$$

$$\text{[N-Send]}\quad \frac{\mathrm{Val}(v) \qquad \mathrm{fv}(\rho) = \varnothing}{L \triangleright \texttt{send}\; \texttt{ret}(v)\; \texttt{to}\; \rho \overset{v \rightsquigarrow \rho \setminus \{L\}}{\Longrightarrow} \texttt{ret}(v)} \qquad \text{[N-Recv]}\quad \frac{\mathrm{Val}(v) \qquad L' \neq L}{L \triangleright \texttt{recv from}\; L' \overset{L'.v \rightsquigarrow}{\Longrightarrow} \texttt{ret}(v)}$$

$$\text{[N-Choose]}\quad \frac{\mathrm{fv}(\rho) = \varnothing}{L \triangleright \texttt{choose}\; d\; \texttt{for}\; \rho \; ; E \overset{d \rightsquigarrow \rho \setminus \{L\}}{\Longrightarrow} E} \qquad \text{[N-AllowL]}\quad \frac{L' \neq L}{L \triangleright \textbf{allow}\; L'\; \textbf{choice}\; (\mathbf{L} \Rightarrow E_1)\; (\mathbf{R} \Rightarrow E_{2\perp}) \overset{L'.\mathbf{L} \rightsquigarrow}{\Longrightarrow} E_1}$$

$$\text{[N-AllowR]}\quad \frac{L' \neq L}{L \triangleright \textbf{allow}\ L'\ \textbf{choice}\ (\mathbf{L} \Rightarrow E_{1\perp})\ (\mathbf{R} \Rightarrow E_2) \xRightarrow{L'.\mathbf{R}\rightsquigarrow} E_2}$$

$$\text{[N-IAmIn]}\quad \frac{L \in \rho}{L \triangleright \texttt{AmI}{\in}\ \rho\ \texttt{then}\ E_1\ \texttt{else}\ E_2 \xRightarrow{\iota} E_1}$$

$$\text{[N-IAmNotIn]}\quad \frac{L \notin \rho}{L \triangleright \texttt{AmI}{\in}\ \rho\ \texttt{then}\ E_1\ \texttt{else}\ E_2 \xRightarrow{\iota} E_2}$$

D Compilation

D.1 Network Program Merging

We show the patterns for which $E_1 \sqcup E_2$ is defined; if there is no matching pattern, then $E_1 \sqcup E_2$ is undefined.

$$\begin{aligned}
\text{undefined} \sqcup \text{undefined} &= \text{undefined}\\
\text{undefined} \sqcup E_2 &= E_2\\
E_1 \sqcup \text{undefined} &= E_1\\
X \sqcup X &= X\\
() \sqcup () &= ()\\
\texttt{ret}(e) \sqcup \texttt{ret}(e) &= \texttt{ret}(e)\\
(E_{1,1}\ ;\ E_{1,2}) \sqcup (E_{2,1}\ ;\ E_{2,2}) &= E_{1,1} \sqcup E_{2,1}\ ;\ E_{1,2} \sqcup E_{2,2}\\
(\texttt{fun}\ F(X) \coloneqq E_1) \sqcup (\texttt{fun}\ F(X) \coloneqq E_2) &= \texttt{fun}\ F(X) \coloneqq (E_1 \sqcup E_2)\\
(E_{1,1}\ E_{1,2}) \sqcup (E_{2,1}\ E_{2,2}) &= (E_{1,1} \sqcup E_{2,1})\ (E_{1,2} \sqcup E_{2,2})\\
(\Lambda\alpha :: \kappa.\ E_1) \sqcup (\Lambda\alpha :: \kappa.\ E_2) &= \Lambda\alpha :: \kappa.\ (E_1 \sqcup E_2)\\
(E_1\ t) \sqcup (E_2\ t) &= (E_1 \sqcup E_2)\ t\\
(\texttt{fold}\ E_1) \sqcup (\texttt{fold}\ E_2) &= \texttt{fold}\ (E_1 \sqcup E_2)\\
(\texttt{unfold}\ E_1) \sqcup (\texttt{unfold}\ E_2) &= \texttt{unfold}\ (E_1 \sqcup E_2)\\
(E_{1,1}, E_{1,2}) \sqcup (E_{2,1}, E_{2,2}) &= ((E_{1,1} \sqcup E_{2,1}), (E_{1,2} \sqcup E_{2,2}))\\
(\texttt{fst}\ E_1) \sqcup (\texttt{fst}\ E_2) &= \texttt{fst}\ (E_1 \sqcup E_2)\\
(\texttt{snd}\ E_1) \sqcup (\texttt{snd}\ E_2) &= \texttt{snd}\ (E_1 \sqcup E_2)\\
(\texttt{inl}\ E_1) \sqcup (\texttt{inl}\ E_2) &= \texttt{inl}\ (E_1 \sqcup E_2)\\
(\texttt{inr}\ E_1) \sqcup (\texttt{inr}\ E_2) &= \texttt{inr}\ (E_1 \sqcup E_2)\\
\begin{pmatrix} \texttt{case}\ E_{1,1}\ \texttt{of}\\ \mid \texttt{inl}\ X \Rightarrow E_{1,2}\\ \mid \texttt{inr}\ Y \Rightarrow E_{1,3} \end{pmatrix} \sqcup \begin{pmatrix} \texttt{case}\ E_{2,1}\ \texttt{of}\\ \mid \texttt{inl}\ X \Rightarrow E_{2,2}\\ \mid \texttt{inr}\ Y \Rightarrow E_{2,3} \end{pmatrix} &= \begin{matrix} \texttt{case}\ (E_{1,1} \sqcup E_{2,1})\ \texttt{of}\\ \mid \texttt{inl}\ X \Rightarrow E_{1,2} \sqcup E_{2,2}\\ \mid \texttt{inr}\ Y \Rightarrow E_{1,3} \sqcup E_{2,3} \end{matrix}\\
(\texttt{let}\ x \coloneqq E_{1,1}\ \texttt{in}\ E_{1,2}) \sqcup (\texttt{let}\ x \coloneqq E_{2,1}\ \texttt{in}\ E_{2,2}) &= \texttt{let}\ x \coloneqq (E_{1,1} \sqcup E_{2,1})\ \texttt{in}\ (E_{1,2} \sqcup E_{2,2})\\
(\texttt{let}\ \alpha :: \kappa \coloneqq E_{1,1}\ \texttt{in}\ E_{1,2}) \sqcup (\texttt{let}\ \alpha :: \kappa \coloneqq E_{2,1}\ \texttt{in}\ E_{2,2}) &= \texttt{let}\ \alpha :: \kappa \coloneqq (E_{1,1} \sqcup E_{2,1})\ \texttt{in}\ (E_{1,2} \sqcup E_{2,2})\\
(\texttt{send}\ E_1\ \texttt{to}\ \rho) \sqcup (\texttt{send}\ E_2\ \texttt{to}\ \rho) &= \texttt{send}\ (E_1 \sqcup E_2)\ \texttt{to}\ \rho
\end{aligned}$$

$$(\texttt{recv from } \ell) \sqcup (\texttt{recv from } \ell) = \texttt{recv from } \ell$$

$$(\texttt{choose } d \texttt{ for } \ell \texttt{ ; } E_1) \sqcup (\texttt{choose } d \texttt{ for } \ell \texttt{ ; } E_2) = \texttt{choose } d \texttt{ for } \ell \texttt{ ; } (E_1 \sqcup E_2)$$

$$\begin{pmatrix} \texttt{allow } \ell \texttt{ choice} \\ \mid \mathbf{L} \Rightarrow E_{1,1} \\ \mid \mathbf{R} \Rightarrow E_{1,2} \end{pmatrix} \sqcup \begin{pmatrix} \texttt{allow } \ell \texttt{ choice} \\ \mid \mathbf{L} \Rightarrow E_{2,1} \\ \mid \mathbf{R} \Rightarrow E_{2,2} \end{pmatrix} = \begin{matrix} \texttt{allow } \ell \texttt{ choice} \\ \mid \mathbf{L} \Rightarrow E_{1,1} \sqcup E_{1,2} \\ \mid \mathbf{R} \Rightarrow E_{2,1} \sqcup E_{2,2} \end{matrix}$$

$$\begin{pmatrix} \texttt{if } E_{1,1} \\ \texttt{then } E_{1,2} \\ \texttt{else } E_{1,3} \end{pmatrix} \sqcup \begin{pmatrix} \texttt{if } E_{2,1} \\ \texttt{then } E_{2,2} \\ \texttt{else } E_{2,3} \end{pmatrix} = \begin{matrix} \texttt{if } (E_{1,1} \sqcup E_{2,1}) \\ \texttt{then } (E_{1,2} \sqcup E_{2,2}) \\ \texttt{else } (E_{1,3} \sqcup E_{2,3}) \end{matrix}$$

$$\begin{pmatrix} \texttt{AmI}\!\in \rho \texttt{ then } E_{1,1} \\ \texttt{else } E_{1,2} \end{pmatrix} \sqcup \begin{pmatrix} \texttt{AmI}\!\in \rho \texttt{ then } E_{2,1} \\ \texttt{else } E_{2,2} \end{pmatrix} = \begin{matrix} \texttt{AmI}\!\in \rho \texttt{ then } (E_{1,1} \sqcup E_{2,1}) \\ \texttt{else } (E_{1,2} \sqcup E_{2,2}) \end{matrix}$$

D.2 Endpoint Projection

Note that $\texttt{AmI } \ell \texttt{ then } E_1 \texttt{ else } E_2$ is shorthand for $\texttt{AmI}\!\in \{\ell\} \texttt{ then } E_1 \texttt{ else } E_2$.

$$[\![X]\!]_L = X$$

$$[\![\rho.e]\!]_L = \begin{cases} \texttt{ret}(e) & \text{if } L \in \rho \\ () & \text{otherwise} \end{cases}$$

$$[\![\mathsf{fun}\, F(X : \tau) \coloneqq C]\!]_L = \texttt{fun}\, F(X) \coloneqq [\![C]\!]_L$$

$$[\![C_1\ C_2]\!]_L = [\![C_1]\!]_L\ [\![C_2]\!]_L$$

$$[\![\Lambda\alpha :: \kappa.\, C]\!]_L = \begin{cases} \Lambda\alpha :: *_\ell.\, \texttt{AmI } \alpha \texttt{ then } [\![C[\alpha \mapsto L]]\!]_L \texttt{ else } [\![C]\!]_L & \text{if } \kappa = *_\ell \\ \Lambda\alpha :: *_s.\, \texttt{AmI}\!\in \alpha \texttt{ then } [\![C[\alpha \mapsto \{L\} \cup \alpha]]\!]_L \texttt{ else } [\![C]\!]_L & \text{if } \kappa = *_s \\ \Lambda\alpha :: \kappa.\, [\![C]\!]_L & \text{otherwise} \end{cases}$$

$$[\![C\ t]\!]_L = [\![C]\!]_L\ t$$

$$[\![\mathsf{fold}\ C]\!]_L = \texttt{fold}\ [\![C]\!]_L$$

$$[\![\mathsf{unfold}\ C]\!]_L = \texttt{unfold}\ [\![C]\!]_L$$

$$[\![(C_1, C_2)]\!]_L = ([\![C_1]\!]_L, [\![C_2]\!]_L)$$

$$[\![\mathsf{fst}\ C]\!]_L = \texttt{fst}\ [\![C]\!]_L$$

$$[\![\mathsf{snd}\ C]\!]_L = \texttt{snd}\ [\![C]\!]_L$$

$$[\![\mathsf{inl}\ C]\!]_L = \texttt{inl}\ [\![C]\!]_L$$

$$[\![\mathsf{inr}\ C]\!]_L = \texttt{inr}\ [\![C]\!]_L$$

$$\left[\!\!\left[\begin{matrix} \mathsf{case}\ C\ \mathsf{of} \\ \mid \mathsf{inl}\ X \Rightarrow C_1 \\ \mid \mathsf{inl}\ Y \Rightarrow C_2 \end{matrix}\right]\!\!\right]_L = \begin{matrix} \texttt{case } [\![C]\!]_L \texttt{ of} \\ \mid \texttt{inl } X \Rightarrow [\![C_1]\!]_L \\ \mid \texttt{inr } Y \Rightarrow [\![C_2]\!]_L \end{matrix}$$

$$[\![\mathsf{let}\ \rho.x : t_e \coloneqq C_1\ \mathsf{in}\ C_2]\!]_L = \begin{cases} \texttt{let } x \coloneqq [\![C_1]\!]_L \texttt{ in } [\![C_2]\!]_L & \text{if } L \in \rho \\ [\![C_1]\!]_L \fatsemi [\![C_2]\!]_L & \text{if } L \notin \rho \text{ and } x \notin \text{fv}([\![C_2]\!]_L) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{let}\ \rho.\alpha :: *_e \coloneqq C_1\ \mathsf{in}\ C_2 \rrbracket_L = \begin{cases} \texttt{let}\ \alpha :: *_e \coloneqq \llbracket C_1 \rrbracket_L\ \texttt{in}\ \llbracket C_2 \rrbracket_L & \text{if } L \in \rho \\ \llbracket C_1 \rrbracket_L \fatsemi \llbracket C_2 \rrbracket_L & \text{if } L \notin \rho \text{ and } \alpha \notin \mathrm{fv}(\llbracket C_2 \rrbracket_L) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{let}\ \rho.\alpha :: *_\ell \coloneqq C_1\ \mathsf{in}\ C_2 \rrbracket_L = \begin{cases} \texttt{let}\ \alpha :: *_\ell \coloneqq \llbracket C_1 \rrbracket_L & \text{if } L \in \rho \\ \quad \texttt{in AmI}\ \alpha\ \texttt{then}\ \llbracket C_2[\alpha \mapsto L] \rrbracket_L\ \texttt{else}\ \llbracket C_2 \rrbracket_L & \\ \llbracket C_1 \rrbracket_L \fatsemi \llbracket C_2 \rrbracket_L & \text{if } L \notin \rho \text{ and } \alpha \notin \mathrm{fv}(\llbracket C_2 \rrbracket_L) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{let}\ \rho.\alpha :: *_s \coloneqq C_1\ \mathsf{in}\ C_2 \rrbracket_L = \begin{cases} \texttt{let}\ \alpha :: *_s \coloneqq \llbracket C_1 \rrbracket_L & \text{if } L \in \rho \\ \quad \texttt{in AmI}{\in}\ \alpha\ \texttt{then}\ \llbracket C_2[\alpha \mapsto \{L\} \cup \alpha] \rrbracket_L & \\ \qquad\qquad \texttt{else}\ \llbracket C_2 \rrbracket_L & \\ \llbracket C_1 \rrbracket_L \fatsemi \llbracket C_2 \rrbracket_L & \text{if } L \notin \rho \text{ and } \alpha \notin \mathrm{fv}(\llbracket C_2 \rrbracket_L) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\llbracket C\ \{\ell\}\!\rightsquigarrow \rho \rrbracket_L = \begin{cases} \texttt{send}\ \llbracket C \rrbracket_L\ \texttt{to}\ \rho & \text{if } L = \ell \\ \llbracket C \rrbracket_L \fatsemi \texttt{recv from}\ \ell & \text{if } L \neq \ell \text{ and } L \in \rho \\ \llbracket C \rrbracket_L & \text{otherwise} \end{cases}$$

$$\llbracket \ell[d] \rightsquigarrow \rho\ ;\ C \rrbracket_L = \begin{cases} \texttt{choose}\ d\ \texttt{for}\ \rho\ ;\ \llbracket C \rrbracket_L & \text{if } L = \ell \\ \texttt{allow}\ \ell\ \texttt{choice}\ (\mathbf{L} \Rightarrow \llbracket C \rrbracket_L) & \text{if } L \neq \ell \text{ and } L \in \rho \text{ and } d = \mathbf{L} \\ \texttt{allow}\ \ell\ \texttt{choice}\ (\mathbf{R} \Rightarrow \llbracket C \rrbracket_L) & \text{if } L \neq \ell \text{ and } L \in \rho \text{ and } d = \mathbf{L} \\ \llbracket C \rrbracket_L & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{if}_\rho\ C\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2 \rrbracket_L = \begin{cases} \texttt{if}\ \llbracket C \rrbracket_L\ \texttt{then}\ \llbracket C_1 \rrbracket_L\ \texttt{else}\ \llbracket C_2 \rrbracket_L & \text{if } L \in \rho \\ \llbracket C \rrbracket_L \fatsemi \llbracket C_1 \rrbracket_L \sqcup \llbracket C_2 \rrbracket_L & \text{otherwise} \end{cases}$$

D.3 Locations Named by a Type or Choreography

$$\begin{aligned} \mathrm{LN}(\alpha) &= \varnothing \\ \mathrm{LN}(L) &= \{L\} \\ \mathrm{LN}(\{\ell\}) &= \mathrm{LN}(\ell) \\ \mathrm{LN}(\rho_1 \cup \rho_2) &= \mathrm{LN}(\rho_1) \cup \mathrm{LN}(\rho_2) \\ \mathrm{LN}(X) &= \varnothing \\ \mathrm{LN}(\rho.e) &= \mathrm{LN}(\rho) \\ \mathrm{LN}(\mathsf{fun}\ F(X : \tau) \coloneqq C) &= \mathrm{LN}(C) \\ \mathrm{LN}(C_1\ C_2) &= \mathrm{LN}(C_1) \cup \mathrm{LN}(C_2) \\ \mathrm{LN}(\Lambda\alpha :: \kappa.\ C) &= \mathrm{LN}(C) \\ \mathrm{LN}(C\ t) &= \mathrm{LN}(C) \cup \mathrm{LN}(t) \\ \mathrm{LN}(\mathsf{fold}\ C) &= \mathrm{LN}(C) \\ \mathrm{LN}(\mathsf{unfold}\ C) &= \mathrm{LN}(C) \\ \mathrm{LN}((C_1, C_2)) &= \mathrm{LN}(C_1) \cup \mathrm{LN}(C_2) \end{aligned}$$

$$\begin{aligned}
\mathrm{LN}(\mathsf{fst}\ C) &= \mathrm{LN}(C)\\
\mathrm{LN}(\mathsf{snd}\ C) &= \mathrm{LN}(C)\\
\mathrm{LN}(\mathsf{inl}\ C) &= \mathrm{LN}(C)\\
\mathrm{LN}(\mathsf{inr}\ C) &= \mathrm{LN}(C)\\
\mathrm{LN}\left(\begin{array}{l}\mathsf{case}\ C\ \mathsf{of}\\ \mid \mathsf{inl}\ X \Rightarrow C_1\\ \mid \mathsf{inl}\ Y \Rightarrow C_2\end{array}\right) &= \mathrm{LN}(C) \cup \mathrm{LN}(C_1) \cup \mathrm{LN}(C_2)\\
\mathrm{LN}(\mathsf{let}\ \rho.x : t_e \coloneqq C_1\ \mathsf{in}\ C_2) &= \mathrm{LN}(\rho) \cup \mathrm{LN}(C_1) \cup \mathrm{LN}(C_2)\\
\mathrm{LN}(\mathsf{let}\ \rho.\alpha :: \kappa \coloneqq C_1\ \mathsf{in}\ C_2) &= \mathrm{LN}(\rho) \cup \mathrm{LN}(C_1) \cup \mathrm{LN}(C_2)\\
\mathrm{LN}(C\ \{\ell\}\!\rightsquigarrow \rho) &= \mathrm{LN}(\ell) \cup \mathrm{LN}(\rho) \cup \mathrm{LN}(C)\\
\mathrm{LN}(\ell[d] \rightsquigarrow \rho\ ;\ C) &= \mathrm{LN}(\ell) \cup \mathrm{LN}(\rho) \cup \mathrm{LN}(C)\\
\mathrm{LN}(\mathsf{if}_\rho\ C\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2) &= \mathrm{LN}(\rho) \cup \mathrm{LN}(C) \cup \mathrm{LN}(C_1) \cup \mathrm{LN}(C_2)
\end{aligned}$$

D.4 The Less-Than Relation

$$\frac{}{\text{undefined} \preceq E} \qquad \frac{E_1 \preceq E_2 \qquad \mathrm{Val}(V)}{E_1 \preceq V\ ;\ E_2} \qquad \frac{}{X \preceq X} \qquad \frac{}{() \preceq ()} \qquad \frac{}{\mathtt{ret}(e) \preceq \mathtt{ret}(e)}$$

$$\frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{E_{1,1}\ ;\ E_{1,2} \preceq E_{2,1}\ ;\ E_{2,2}} \qquad \frac{E_1 \preceq E_2}{\mathtt{fun}\ F(X) \coloneqq E_1 \preceq \mathtt{fun}\ F(X) \coloneqq E_2} \qquad \frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{E_{1,1}\ E_{1,2} \preceq E_{2,1}\ E_{2,2}}$$

$$\frac{E_1 \preceq E_2}{\Lambda\alpha.\, E_1 \preceq \Lambda\alpha.\, E_2} \qquad \frac{E_1 \preceq E_2}{E_1\ t \preceq E_2\ t} \qquad \frac{E_1 \preceq E_2}{\mathtt{fold}\ E_1 \preceq \mathtt{fold}\ E_2} \qquad \frac{E_1 \preceq E_2}{\mathtt{unfold}\ E_1 \preceq \mathtt{unfold}\ E_2}$$

$$\frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{(E_{1,1}, E_{1,2}) \preceq (E_{2,1}, E_{2,2})} \qquad \frac{E_1 \preceq E_2}{\mathtt{fst}\ E_1 \preceq \mathtt{fst}\ E_2} \qquad \frac{E_1 \preceq E_2}{\mathtt{snd}\ E_1 \preceq \mathtt{snd}\ E_2} \qquad \frac{E_1 \preceq E_2}{\mathtt{inl}\ E_1 \preceq \mathtt{inl}\ E_2}$$

$$\frac{E_1 \preceq E_2}{\mathtt{inr}\ E_1 \preceq \mathtt{inr}\ E_2} \qquad \frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2} \qquad E_{1,3} \preceq E_{2,3}}{\begin{array}{l}\mathtt{case}\ E_{1,1}\ \mathtt{of}\\ \mid \mathtt{inl}\ X \Rightarrow E_{1,2}\\ \mid \mathtt{inr}\ Y \Rightarrow E_{1,3}\end{array} \preceq \begin{array}{l}\mathtt{case}\ E_{2,1}\ \mathtt{of}\\ \mid \mathtt{inl}\ X \Rightarrow E_{2,2}\\ \mid \mathtt{inr}\ Y \Rightarrow E_{2,3}\end{array}}$$

$$\frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{\mathtt{let}\ x \coloneqq E_{1,1}\ \mathtt{in}\ E_{1,2} \preceq \mathtt{let}\ x \coloneqq E_{2,1}\ \mathtt{in}\ E_{2,2}} \qquad \frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{\mathtt{let}\ \alpha :: \kappa \coloneqq E_{1,1}\ \mathtt{in}\ E_{1,2} \preceq \mathtt{let}\ \alpha :: \kappa \coloneqq E_{2,1}\ \mathtt{in}\ E_{2,2}}$$

$$\frac{E_1 \preceq E_2}{\mathtt{send}\ E_1\ \mathtt{to}\ \rho \preceq \mathtt{send}\ E_2\ \mathtt{to}\ \rho} \qquad \frac{}{\mathtt{recv\ from}\ \ell \preceq \mathtt{recv\ from}\ \ell}$$

$$\frac{E_1 \preceq E_2}{\texttt{choose}\ d\ \texttt{for}\ \ell\ \texttt{;}\ E_1 \preceq \texttt{choose}\ d\ \texttt{for}\ \ell\ \texttt{;}\ E_2}$$

$$\frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{\begin{array}{l}\texttt{allow}\ \ell\ \texttt{choice}\\ \mid \mathbf{L} \Rightarrow E_{1,1}\\ \mid \mathbf{R} \Rightarrow E_{1,2}\end{array} \preceq \begin{array}{l}\texttt{allow}\ \ell\ \texttt{choice}\\ \mid \mathbf{L} \Rightarrow E_{2,1}\\ \mid \mathbf{R} \Rightarrow E_{2,2}\end{array}}$$

$$\frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2} \qquad E_{1,3} \preceq E_{2,3}}{\begin{array}{l}\texttt{if}\ E_{1,1}\\ \texttt{then}\ E_{1,2}\\ \texttt{else}\ E_{1,3}\end{array} \preceq \begin{array}{l}\texttt{if}\ E_{2,1}\\ \texttt{then}\ E_{2,2}\\ \texttt{else}\ E_{2,3}\end{array}}$$

$$\frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{\begin{array}{r}\texttt{AmI}\in \rho\ \texttt{then}\ E_{1,1}\\ \texttt{else}\ E_{1,2}\end{array} \preceq \begin{array}{r}\texttt{AmI}\in \rho\ \texttt{then}\ E_{2,1}\\ \texttt{else}\ E_{2,2}\end{array}}$$