Step in Tine: Forking Processes in Functional Choreographies

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

Traditional concurrent-programming techniques require programmers to painstakingly write programs for each participant in a concurrent system. Choreographic programming, in contrast, allows a programmer to write one centralized program and compile it to the individual programs. This approach simplifies critical properties like deadlock freedom, but it complicates *forking new processes*, a core primitive in concurrent programming. This work addresses that gap with the choreographic fork calculus $\lambda\Y$, the first functional choreographic language with process forking. $\lambda\Y$ provides a deadlock-freedom guarantee while allowing programs to dynamically determine when to spawn new processes, what they will do, and who will communicate with them. In doing so, it supports practical algorithms like parallel divide-and-conquer.

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

As nearly every computer system has come to rely on parallelism for efficiency, the difficulty of writing correct concurrent code has become an increasing concern. Complex interactions between processes can lead to subtle bugs such as deadlocks, where two or more processes are waiting on each other, preventing the system from progressing. *Choreographic programming* [Montesi 2023] has recently emerged as a promising tool to address this challenge. Instead of writing separate programs for each process, the choreographic paradigm allows programmers to specify the behavior and interactions of all processes in a single, top-level program called a choreography. A compiler then produces code for each process using a procedure called *endpoint projection* (EPP). This global specification allows programmers to reason about the system as a whole and leads to *deadlock freedom by design*, eliminating a common source of bugs in concurrent programs.

While early works on choreographic programming built a promising foundation [Carbone and Montesi 2013; Montesi 2013], they lacked the language features necessary for the paradigm to be used in modern software engineering. A flurry of recent papers have been adding these capabilities to choreographic calculi, including higher-order programming [Cruz-Filipe et al. 2022; Giallorenzo et al. 2023; Hirsch and Garg 2022], process polymorphism [Graversen et al. 2024; Samuelson et al. 2025], and multiply-located values [Bates et al. 2025; Samuelson et al. 2025]. While these features have made choreographic programming more expressive, they still lack important features, including the ability to dynamically *fork processes*. This ability is key to many concurrent applications, as was recognized by early choreographic work [see e.g., Carbone and Montesi 2013; Cruz-Filipe and Montesi 2016a,b]. These early calculi, however, focused on simpler languages lacking important functionality needed for modern software engineering.

This work presents $\lambda\Y$ (pronounced "lambda-fork") to bridge this gap by integrating dynamic process forking with the powerful features mentioned above. To see how these capabilities combine,

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.

consider the following recursive divide-and-conquer algorithm to sum a list of integers.

```
recursiveSum : ∀ℓ. list(int)@ℓ → int@ℓ
recursiveSum ℓ XS = if ℓ.(len XS < localMaxLen)
                    then ℓ.sum(XS)
                    else let (ℓ.xs₁, ℓ.xs₂) := ℓ.split(XS)
                             α := ℓ.fork()
                             α.s₁ := recursiveSum α (ℓ.xs₁ ⇝ α)
                             ℓ.s₂ := recursiveSum ℓ ℓ.xs₂
                             ℓ.s₁ := α.s₁ ⇝ ℓ
                         in ℓ.(s₁ + s₂)
```

This choreography finds the sum of the list XS owned by ℓ, indicated by the type $\mathsf{list}(\mathsf{int})@\ell$. The first line checks if XS is long enough to be worth parallelizing; otherwise, ℓ sums the list locally. For longer lengths, ℓ splits XS into two halves, xs_1 and xs_2, recursively summing each half in parallel. To perform this parallelism, ℓ uses the syntax $\ell.\mathsf{fork}()$ to spawn a new thread α to sum xs_1 while ℓ sums xs_2. Once α is spawned, ℓ sends it the first half of XS using the notation $\ell.xs_1 \rightsquigarrow \alpha$, which produces a value located at α. The new thread α then calls recursiveSum using this value, potentially spawning its own children to sum its half of the list.

Though they are separate processes, both ℓ and α can call recursiveSum with their respective lists because it is *process polymorphic*—can execute with any location—as indicated by the $\forall\ell$ preceding its type. After both processes have summed their halves, α sends its sum s_1 to ℓ. At this point, the thread α falls out of scope and (implicitly) dies. Finally, ℓ returns the sum $s_1 + s_2$.

The fork construct—a core contribution of $\lambda\Lsh$—allows ℓ to spawn a new thread, binding a variable to its name. This child thread is treated like any other process: while its name is in scope, it may be asked to execute single-threaded computations, perform control flow to sequence multiple computations, communicate with other processes, and spawn further threads. Moreover, the fork construct eases programmers' administrative burden. First, the programmer does not need to explicitly state what code a thread will run when it is spawned; EPP extracts the code for the new thread automatically. Second, because process names are scoped, new threads implicitly die when their name is no longer in scope.

Retaining core properties like deadlock freedom in $\lambda\Lsh$ requires careful design. Combining fork with closures and (first-class) location polymorphism poses a particular challenge not raised by prior work. A function F that closes over the name α of a spawned process could persist beyond the scope of α. Applying F could then cause a live process to attempt to communicate with α, immediately producing a deadlock. $\lambda\Lsh$ prevents such situations through careful tracking of spawned location names in the type system.

Section 2 reviews background, Section 3 defines the system model underlying $\lambda\Lsh$. Then, the main contributions of this work are presented as follows:

- Section 4 presents $\lambda\Lsh$, the first functional choreographic language to allow forking and killing child threads. $\lambda\Lsh$ also includes first-class process names, enabling parent processes to notify other locations when they have spawned a child.
- Section 4.3.2 formulates a type system for $\lambda\Lsh$ that tracks which processes might participate in a choreography to ensure that no process needs to perform computation after it dies.
- Section 6 defines endpoint projection (EPP), a procedure to compile a choreography into a target-language program (Section 5) for each process, and characterizes EPP's correctness with respect to a top-level operational semantics. This result combines with the soundness of the type system to prove that executing a projected system will never cause a deadlock.

Finally, Section 7 reviews related work, and Section 8 concludes.

2 Background

To better situate the contributions of our work, we first review the design, features, and limitations of prior choreographic programming languages.

2.1 Functional Choreographies

Our language primarily extends λQC [Samuelson et al. 2025], which in turn extends Pirouette [Hirsch and Garg 2022], the first functional choreographic programming language. Like most choreographic languages, λQC prefixes each local operation with the process that performs it. For example, to specify that location A should compute $1+3$ and send the result to B, one would write $\mathsf{A}.(1+3) \rightsquigarrow \mathsf{B}$. To differentiate operations, we write source programs using a sans-serif font, with location constants in red, local operations in green, and choreographic operations in blue. Local programs such as "$1 + 3$" can be specified in nearly any language, so long as it is equipped with a substitution-based operational semantics, a sound type system, and its values can be shared via message-passing. Network-level constructs such as send ($\rightsquigarrow$), on the other hand, are determined by the choreographic language.

While the choreographic and local operations are separate, the choreography can sequence local computations using features such as let-expressions. For example, the output of $\mathsf{A}.(1 + 3) \rightsquigarrow \mathsf{B}$ is an integer located at B, which can then be used in a subsequent local computation at B by binding the result to a (local) variable x as follows: $\mathsf{let}\ \mathsf{B}.x := \mathsf{A}.(1 + 3) \rightsquigarrow \mathsf{B}\ \mathsf{in}\ \mathsf{B}.(x - 2)$.

Choreography-level control flow is supported by the expression $\mathsf{if}\ C\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$, where C evaluates to a boolean value known to some process ℓ. As others outside of ℓ cannot see the output of C, they cannot determine which branch to execute. To allow other locations to participate in this branch, λQC includes a *selection statement* $\ell[d] \rightsquigarrow \rho\ ;\ C$ in which ℓ communicates the chosen branching direction $d \in \{\mathsf{L}, \mathsf{R}\}$ of **L**eft or **R**ight to all locations in the set ρ.

λQC supports *multiply-located values* (MLVs) [Bates et al. 2025; Sweet et al. 2023], which are local values known to multiple locations and ensure all parties agree. For instance, $\{\mathsf{A}, \mathsf{B}\}.(3 > 1)\ {}_{\{\mathsf{A}\}}\!\rightsquigarrow \mathsf{C}$ first instructs A and B to compute the local operation $3 > 1$, and then instructs A to send the result to C. The result is the multiply-located value $\{\mathsf{A}, \mathsf{B}, \mathsf{C}\}.\mathsf{true}$. MLVs can be used as an alternative to synchronization messages for branching. For instance, the following choreography ensures all three locations branch in the same direction: $\mathsf{if}_{\{\mathsf{A},\mathsf{B},\mathsf{C}\}}\ \{\mathsf{A}, \mathsf{B}\}.(3 > 1)\ {}_{\{\mathsf{A}\}}\!\rightsquigarrow \mathsf{C}\ \mathsf{then}\ C_1\ \mathsf{else}\ C_2$. Note that when using MLVs, it often becomes necessary to annotate e.g., who is sending a value or participating in an if expression.

Endpoint Projection. Like most choreographies, λQC defines a compilation procedure called *endpoint projection* (EPP) that translates a choreography into a separate program for each participant. EPP is a syntax-guided translation that extracts the actions that a single location needs to perform from the choreography. The location being projected to is denoted by a subscript to the compilation operator, as in $[\![C]\!]_{\mathsf{A}}$ and $[\![C]\!]_{\mathsf{B}}$. For instance, consider the choreography

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

in which A and B each compute a value and then send it to C. The only actions that A and B need to perform are computing the value and sending it, while C needs to receive both values:

$$[\![C]\!]_{\mathsf{A}} = \texttt{send ret}(2 * 4)\ \texttt{to C} \qquad [\![C]\!]_{\mathsf{B}} = \texttt{send ret}(3 + 2)\ \texttt{to C} \qquad [\![C]\!]_{\mathsf{C}} = \begin{array}{l}\texttt{recv from A ;}\\ \texttt{recv from B}\end{array}$$

The target (network) language is written using an orange teletype font.

λQC also defines a top-level operational semantics directly on choreographies, allowing developers to reason about the behavior of the system as a whole. This choreographic semantics allows for out-of-order execution, so long as the order of operations for each individual location is respected. For instance, note that if we execute the projected programs shown above concurrently, either A or B could perform their local computation first, but C must receive the values in the specified order. This means that A and B can compute 8 and 5, respectively, in any order in choreography C, even though B's computation is after the semicolon. C, conversely, must receive 8 before 5. The top-level choreographic semantics allows any of these orderings.

As EPP and the top-level semantics provide two different interpretations of the same choreography, it is important that their results are equivalent. Samuelson et al. [2025] show that these two semantics are bisimilar for λQC, and so will always produce the same value. Besides allowing developers to soundly reason about the execution of a system using the top-level semantics, this property also guarantees that any concurrent execution of a projected choreography is deadlock-free, meaning that no process will wait indefinitely for a message that will never arrive due to a mismatch between the expected send and receive operations.

2.2 Process Polymorphism

Process polymorphism [Graversen et al. 2024] allows choreographies to abstract over their participants. Analogously to type polymorphism, process polymorphism in λQC is implemented with a process abstraction $\Lambda\ell.\, C$, where variable ℓ represents a generic process name bound in the scope of C, allowing it to be instantiated with different participants. For example, a programmer could write a process function in which ℓ computes the sum of two numbers and sends the result to A as $F = \Lambda\ell.\, \ell.(1 + 3) \rightsquigarrow \mathsf{A}$, and later instantiate ℓ to B with the syntax $F\ \mathsf{B}$.

While process polymorphism allows for choreographies to be instantiated with different participants, it does not—on its own—allow for the names of processes to be treated in a first-class manner. First-class process names [Samuelson et al. 2025; Sweet et al. 2023] solve this problem by allowing local computations to generate, examine, and send process names as values. For example, the expression $\mathsf{A}.(\text{if } e \text{ then } \lceil \mathsf{B} \rfloor \text{ else } \lceil \mathsf{C} \rfloor)$ selects between the process names B and C based on the value of the boolean e known to A. The output of this computation can then be shared with B and C so they are aware of who should perform the subsequent computation, and bound to a variable ℓ using λQC's *type-let expression* as follows:

$$\begin{array}{l} \text{let } \{\mathsf{A}, \mathsf{B}, \mathsf{C}\}.\ell \coloneqq \mathsf{A}.(\text{if } e \text{ then } \lceil \mathsf{B} \rfloor \text{ else } \lceil \mathsf{C} \rfloor) \rightsquigarrow \{\mathsf{B}, \mathsf{C}\} \\ \text{in } \ell.(1 + 3) \rightsquigarrow \mathsf{A} \end{array}$$

2.3 Process Spawning

While $\lambda\Uparrow$ is the first *functional* choreographic language to include the ability to spawn and kill child threads, previous procedural and imperative choreographic languages have included this feature. For example, the language introduced by Cruz-Filipe and Montesi [2016a] allows programmers to write a recursive divide-and-conquer implementation of merge sort, similar to that in Section 1. However, the features of their language are heavily restricted: each process stores a single memory cell holding a value of a fixed type, and the value of the memory cell may only be modified by calling a local procedure or accepting a value from another process. The early choreographic language constructed by Carbone and Montesi [2013] includes process spawning and allows more expressive local computations. However, their language lacks process polymorphism and higher-orderedness—eroding modularity and precluding the recursive divide-and-conquer approach.

3 System Model

$\lambda\Uparrow$ assumes the underlying system contains a set of potential computational units (threads, processes, etc.), which we refer to as *locations*, and each location has a unique name from a space $\mathcal{L}$. The number of locations executing at any given time is finite, but programs may spawn an unbounded number of new locations and each name must be unique, so $\mathcal{L}$ must be infinite.

As noted in Section 2.1, $\lambda\Uparrow$ follows Pirouette [Hirsch and Garg 2022] and λQC [Samuelson et al. 2025] and allows local programs to be specified in nearly any language. This *local language* must only satisfy a set of rules common to most expression-based languages. Our assumptions on the local language are nearly identical to those in λQC and Pirouette, although we make some generalizations.

3.1 Local Operational Semantics

We require that the local language be presented as a set of expressions coupled with a small-step operational semantics, a distinguished set of values, and a type system. We write $e_1 \longrightarrow e_2$ to denote that the expression e_1 steps to e_2 in the local language's operational semantics. The semantics must satisfy the following two properties.

(1) Values cannot step: if $\mathrm{Val}(v)$, then there is no e such that $v \longrightarrow e$.
(2) The semantics satisfies the *diamond property*: 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 a standard assumption about the set of values in the language, and property (2) ensures multiply-located computations produce the same result at each location they execute at. While property (2) may seem restrictive, it is satisfied by any deterministic language. Thus, large subsets of industrial-strength functional languages can be used for local computations.

3.2 Local Type System

Just as the syntax of a choreography depends on the syntax of the local language, the choreographic type system depends on the local language specifying a type system. The local type system must include both a kinding judgment and a typing judgement. This allows, but does not require, the local type system to be polymorphic. The type system must also be sound with respect to the operational semantics. Specifically, it should satisfy the standard progress and preservation properties.

We denote a local kinding judgment $\Gamma \Vdash t :: *_e$. We assume for simplicity there is a single local kind $*_e$, but our results generalize to languages with multiple kinds. We denote local typing judgements $\Gamma; \Delta \Vdash e : t$ where Γ is again a kinding context and Δ is a typing context. To distinguish these judgments from the choreographic type system, we use a green double-vertical turnstile $\Vdash$.

To support choreographic control-flow branching, we generalize Pirouette and λQC. Instead of requiring a local boolean type, we allow an arbitrary (user-defined) predicate $\mathrm{isSum}(s, t_1, t_2)$ indicating that every value of type s can be interpreted as either a t_1 or a t_2. For instance, the local language can specify $\mathrm{isSum}(\mathsf{bool}, \mathsf{unit}, \mathsf{unit})$ and interpret true as inl () and false as inr (). The only requirement is that there is a deterministic partial function getCase called the *extraction function*. We require that, if $\mathrm{isSum}(s, t_1, t_2)$ and $\Vdash v : s$ for a value v, then either $\mathrm{getCase}(v) = \mathrm{inl}(v_1)$ with $\Vdash v_1 : t_1$ or $\mathrm{getCase}(v) = \mathrm{inr}(v_2)$ with $\Vdash v_2 : t_2$. On other expressions, it may be undefined.

We support first-class process names identically to λQC. Specifically, the local language has two types loc_ρ and locset_ρ defining first-class *representations* of location names and sets of locations, respectively. As with the sum types above, the local language must be able to uniquely reify any well-typed representation into a corresponding kind. For instance, if the set $\mathcal{L}$ of locations consists of all possible strings (e.g., "Alice", "Bob", and "Charlie"), we could directly represent a location by its string. We write representations using the syntax $\lceil \mathsf{A} \rfloor$ (which here is syntactic sugar for "Alice") and $\lceil \{\mathsf{A}, \mathsf{B}\} \rfloor$ (which is {"Alice", "Bob"}) to distinguish them from the actual location

$\mathsf{A} \in \mathcal{L}$ or location set $\{\mathsf{A}, \mathsf{B}\} \subseteq \mathcal{L}$. Since a spawned thread could take any name, we require each location $L \in \mathcal{L}$ to have at least one representation $\lceil L \rfloor$ to ensure our choreographies do not become stuck when spawning a thread. We do not require there to be any representation for a given set of locations, however, as this is not a safety concern.

The subscript ρ to the types loc_ρ and locset_ρ provides an upper-bound on the set of locations to which an expression of that type might resolve to. As an example, we could assign both $\lceil \mathsf{A} \rfloor$ and $\mathsf{if}\ e\ \mathsf{then}\ \lceil \mathsf{A} \rfloor\ \mathsf{else}\ \lceil \mathsf{B} \rfloor$ to the type $\mathsf{loc}_{\{\mathsf{A},\mathsf{B}\}}$, but $\lceil \mathsf{C} \rfloor$ cannot be assigned this type. Different local languages may determine different values for ρ depending on the strength of their type system and static-analysis abilities. The precision of this bound will not affect the operational semantics of any choreography, but it may force the programmer to add additional operations to some choreographies in order to satisfy the type system.

3.3 Example Local Languages

Many λ-calculi satisfy our requirements with minor, but standard, modifications. Below we present two example local langauges—one extending the simply-typed λ-calculus, and the other System F.

Example 1 (Simply-Typed λ-Calculus)**.** Assuming that $\mathcal{L}$ is the set of all strings, we extend the simply-typed, call-by-value λ-calculus to obtain our first example local language. First, we extend it with sum types, and define $\mathrm{isSum}(s, t_1, t_2)$ to hold precisely when $s = t_1 + t_2$, and define the extraction function as

$$\mathrm{getCase}(e) = \begin{cases} \mathrm{inl}(v) & e = \mathsf{inl}\ v \\ \mathrm{inr}(v) & e = \mathsf{inr}\ v \\ \text{undefined} & \text{otherwise} \end{cases}$$

We further include primitive strings and define $\lceil L \rfloor$ as the string-primitive version of L. In addition to the type string of all strings, we include a type $\mathsf{loc}_{\{L\}}$ containing only $\lceil L \rfloor$ for every location L. We also let locset_ρ be empty for every ρ and loc_ρ be empty when $|\rho| \neq 1$.

The representations defined above demonstrate that the local language need only perform a very simple static analysis to determine the bound ρ on the type loc_ρ. While this design simplifies the language significantly, it precludes expressions such as $\mathsf{if}\ e\ \mathsf{then}\ \lceil \mathsf{A} \rfloor\ \mathsf{else}\ \lceil \mathsf{B} \rfloor$ from having interesting types like $\mathsf{loc}_{\{\mathsf{A},\mathsf{B}\}}$, instead forcing them to be typed with string. To address this shortcoming, we make our final extension to STLC: a primitive function $\mathsf{cast}_L : \mathsf{string} \to \mathsf{unit} + \mathsf{loc}_{\{L\}}$ that dynamically checks if a primitive string argument is equal to $\lceil L \rfloor$, casting it to type $\mathsf{loc}_{\{L\}}$ if so and otherwise returning a unit. This allows dynamic analysis to replace the potentially complex static analysis that otherwise might seem unavoidable. In this instance, by attempting to cast the result of the if-expression above to both $\mathsf{loc}_{\{\mathsf{A}\}}$ and $\mathsf{loc}_{\{\mathsf{B}\}}$—and chaining the output of the casts—we can realize the type $\mathsf{unit} + \mathsf{loc}_{\{\mathsf{A}\}} + \mathsf{loc}_{\{\mathsf{B}\}}$, allowing for this dynamically generated representation to be used at the choreographic level by separately handling the cases when it resolves to A, B, or another location—which will never occur.

Example 2 (System F)**.** For an example of a more-expressive local langauge, System F with algebraic and recursive data types, and primitive strings representing locations, satisfies our requirements. Since we do not require that all local expressions terminate, there is no issue with including named recursive functions and unrestricted recursive types. We use lists of strings (defined using recursive data types) to represent location sets. Having multiple list permutations represent the same set of locations is also not a concern, since we do not require representations to be unique. The syntax of

this potential local language is shown below.

$$
\begin{array}{lrcl}
\text{Types} & t & ::= & \alpha \mid \mathsf{string} \mid \mathsf{loc}_\rho \mid \mathsf{locset}_\rho \mid t_1 \to t_2 \mid t_1 + t_2 \mid t_1 \times t_2 \mid \forall \alpha.\, t \mid \mu \alpha.\, t \\
\text{Expressions} & e & ::= & x \mid s \in \mathit{str} \mid \mathsf{fun}\, f(x\!:\!t) := e \mid e_1\ e_2 \mid \Lambda \alpha.\, e \mid e\ t \\
& & \mid & \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid \mathsf{case}\ e\ \mathsf{of}\ (\mathsf{inl}\ x \Rightarrow e_1)\ (\mathsf{inr}\ y \Rightarrow e_2) \\
& & \mid & (e_1, e_2) \mid \mathsf{fst}\ e \mid \mathsf{snd}\ e \mid \mathsf{fold}\ e \mid \mathsf{unfold}\ e
\end{array}
$$

To provide a nontrivial static upper-bound on representations of locations, this local language includes subtyping, and allows the type loc_ρ to be inhabited by any string in the set ρ—and similarly for the locset_ρ type—using the rules shown below.

$$
\frac{\Gamma \Vdash \Delta \qquad s \in \mathit{str}}{\Gamma; \Delta \Vdash s : \mathsf{loc}_{\{s\}}}
\qquad
\frac{\Gamma; \Delta \Vdash e : t_1 \qquad t_1 <: t_2}{\Gamma; \Delta \Vdash e : t_2}
\qquad
\frac{\rho_1 \subseteq \rho_2}{\mathsf{loc}_{\rho_1} <: \mathsf{loc}_{\rho_2}}
\qquad
\frac{\rho_1 \subseteq \rho_2}{\mathsf{locset}_{\rho_1} <: \mathsf{locset}_{\rho_2}}
$$

4 The $\lambda\Uparrow$ Language

We now present $\lambda\Uparrow$, the first functional choreographic programming language that can dynamically spawn threads. As previously mentioned, we inherit the core of our language from λQC [Samuelson et al. 2025]—including features such as algebraic and recursive data types, multiply-located local computations, process polymorphism, and first-class process names—and retain the traditional deadlock-freedom guarantee of choreographic languages.

4.1 $\lambda\Uparrow$ Syntax

Figure 1 presents the full syntax of $\lambda\Uparrow$. As in λQC, 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_e a local type, and κ a kind.

Most constructs in $\lambda\Uparrow$ are standard for a functional language, consisting of operations on data types appropriately generalized to choreographies, but there are some key differences. The expression $\rho.e$ denotes a local program e that is executed by all locations in the (non-empty) set ρ. In cases where $\rho = \{\ell\}$ is a singleton, we use the shorthand $\ell.e$. Local programs like e can use variables bound in the scope of the choreography, which are prepended with the location(s) that bind(s) them. For instance, $\mathsf{A}.x$ denotes variable x in the namespace of location A, which is distinct from a variable $\mathsf{B}.x$ in the namespace of B, and this is reflected in our substitution semantics. If a local variable is bound in the scope of multiple locations, we write $\rho.x$ to mean that x is in the namespace of all locations in ρ. Local variables (resp. type variables) can be bound to the result of a

$$
\begin{array}{lrcl}
\text{Selection Labels} & d & ::= & \mathbf{L} \mid \mathbf{R} \\
\text{Choreographies} & C & ::= & X \mid \rho.e \\
& & \mid & \mathsf{let}\ \rho.x\!:\!t_e := C_1\ \mathsf{in}\ C_2 \mid \mathsf{let}\ \rho.\alpha :: \kappa := C_1\ \mathsf{in}\ C_2 \\
& & \mid & C\ \{\ell\}\!\rightsquigarrow \rho \mid \ell[d] \rightsquigarrow \rho\ ;\ C \\
& & \mid & \mathsf{fun}_\rho\, F(X) := C \mid C_1\ \$_\rho\ C_2 \mid \mathsf{tfun}_\rho\, F(\alpha :: \kappa) := C \mid C\ \$_\rho\ t \\
& & \mid & \mathsf{localCase}_\rho\ C\ \mathsf{of}\ (\mathsf{inl}\ x \Rightarrow C_1)\ (\mathsf{inr}\ y \Rightarrow C_2) \\
& & \mid & \mathsf{inl}_\rho\ C \mid \mathsf{inr}_\rho\ C \mid \mathsf{case}_\rho\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inr}\ Y \Rightarrow C_2) \\
& & \mid & \mathsf{fold}_\rho\ C \mid \mathsf{unfold}_\rho\ C \mid (C_1, C_2)_\rho \mid \mathsf{fst}_\rho\ C \mid \mathsf{snd}_\rho\ C \\
& & \mid & \mathsf{let}\ (\alpha, x) := \ell.\mathsf{fork}()\ \mathsf{in}\ C \mid \mathsf{kill}\ L\ \mathsf{after}\ C
\end{array}
$$

Fig. 1. Syntax of Choreographies in $\lambda\Uparrow$

choreography using a let expression $\mathsf{let}\ \rho.x : t_e \coloneqq C_1\ \mathsf{in}\ C_2$ (resp. $\mathsf{let}\ \rho.\alpha :: \kappa \coloneqq C_1\ \mathsf{in}\ C_2$). In both of these expressions, ρ may be a subset of the locations who know the output of C_1.

Data can be shared between locations using the operation $C\ \{\ell\}\!\rightsquigarrow\ \rho$, in which the output of choreography C is sent by ℓ to all locations in the set ρ via message passing. The output of C must be a local value known to ℓ, although it may also be known to others. For notational simplicity, we elide the ℓ and write $C \rightsquigarrow \rho$ when C is known only to ℓ. Since the sender, all recipients, and anyone else who knew the value of a message will all agree on it afterward, the semantics of sends are *collecting*—the output is a value located at all relevant locations. For instance, the send $\{\mathsf{A}, \mathsf{B}\}.(4 - 2)\ \{\mathsf{A}\}\!\rightsquigarrow\ \{\mathsf{C}, \mathsf{D}\}$ results in the multiply-located value $\{\mathsf{A}, \mathsf{B}, \mathsf{C}, \mathsf{D}\}.2$. A separate use of message passing is *selection* statements $\ell[d] \rightsquigarrow \rho' ; C$, which can be used to synchronize on the branch $d \in \{\mathbf{L}, \mathbf{R}\}$ taken in a case expression—described below—with the locations in ρ'.

Named recursive functions are written as $\mathsf{fun}_\rho\ F(X) \coloneqq C$, where F is the name of the function, X is its argument, and C is the body of the function. The additional parameter ρ is the set of locations that may be involved in executing the body of the function and is needed to ensure that spawned process names do not persist beyond the lifetime of the process. It also constrains who knows the function definition and is dually reflected in the syntax for function application, written $C_1\ \$_\rho\ C_2$, which contains an annotation to convey that only those locations in ρ need to perform the application. As with message sending syntax, we elide the $\$_\rho$ when ρ is clear from context.

Polymorphism is implemented in $\lambda\Uparrow$ using type functions and type applications. Type functions, written $\mathsf{tfun}_\rho\ F(\alpha :: \kappa)\ \coloneqq C$, are similar to the type abstractions $\Lambda\alpha.\, C$ found in System F and previous chreographies [Graversen et al. 2024; Samuelson et al. 2025], but allow the function F to be recursively defined. All kinds κ of the language may be abstracted over in type functions, and similarly to standard functions, the set of locations ρ tracks which locations know the definition of the type function. Type applications, written $C\ \$_\rho\ t$, mirror function applications explained above.

$\lambda\Uparrow$ includes two separate forms of branching: local case-expressions and choreographic case-expressions. Local case-expressions, written $\mathsf{localCase}_\rho\ C\ \mathsf{of}\ (\mathsf{inl}\ x \Rightarrow C_1)\ (\mathsf{inr}\ y \Rightarrow C_2)$, generalize the if-expressions found in prior work [e.g., Cruz-Filipe et al. 2022; Graversen et al. 2024; Hirsch and Garg 2022; Samuelson et al. 2025], and branch the choreography on the result of a local computation. Here C must produce a value of local type t known to ρ where $\mathrm{isSum}(t, t_1, t_2)$ holds. The local variables x and y are bound for all locations in ρ with types t_1 and t_2, respectively. To inform additional locations ρ' which branch is taken, a programmer has two options. First, they can share the value of C using the send operation $C\ \{\ell\}\!\rightsquigarrow\ \rho'$ and branch on the resulting collected value. Alternatively, they can include selection statements in the branches to inform locations in ρ' which branch was taken. In the second case, the value of C is not available to the additional locations, which may be desirable for security or performance reasons.

Choreographic case-expressions, written $\mathsf{case}_\rho\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inr}\ Y \Rightarrow C_2)$, are conceptually similar to local case-expressions, but instead branch the choreography on a choreographic sum—either of the form $\mathsf{inl}_\rho\ V_1$ or $\mathsf{inr}_\rho\ V_2$. This means that V_1 or V_2 could be a more complex data type, such as a choreographic pair or list containing multiple local values, rather than just a single local value. Selection statements can also be used in the branches of choreographic case-expressions to allow locations outside of ρ to know which branch to take.

Similar to case-expressions, choreographic pairs and recursive data types act like their usual functional-programming counterparts. but with an annotation ρ describing who knows about the data. As with other such annotations, we elide them when they are clear from context.

Fork and Kill Expressions. The key addition of $\lambda\Uparrow$ is the fork expression, which allows dynamic spawning of new locations. Specifically, the expression $\mathsf{let}\ (\alpha, x) \coloneqq \ell.\mathsf{fork}()\ \mathsf{in}\ C$ instructs location ℓ to spawn a child process and binds its name to the type variable α, allowing the new location to

perform computations in the body C. The variable x is bound at locations α and ℓ to a first-class local representation of the name α, allowing ℓ to notify other locations of the new child and facilitating direct communication between α and any other location.

We include two notational shortcuts for fork. First, if x is not free in C, we simplify the binding and write $\text{let } \alpha := \ell.\text{fork}() \text{ in } C$. Second, the notation $\text{let } \alpha := \ell.\text{fork}() \rightsquigarrow \rho \text{ in } C$ is sugar for

$$\text{let } (\_, x) := \ell.\text{fork}() \text{ in } (\text{let } \alpha := x \ {}^{\{\ell\}}\!\!\rightsquigarrow \rho \text{ in } C)$$

which shares the name α of the newly spawned process with everyone in ρ, as well as ℓ and α itself.

The construct kill L after C serves as a dual to the fork expression, and is used to track which threads are currently spawned and differentiate them from other non-ephemeral processes. Informally, this is not intended to be in the surface language used by programmers. Specifically, when a new thread L is spawned by a fork expression with body C, the body will simply be placed within a kill-after expression while executing to denote the fact that L will die once C finishes execution.

Example 3, shown below, demonstrates how the fork expression can be used in tandem with other language features such as case-expressions, process polymorphism, and the type-let expression.

Example 3 (Load Balancer)**.** Consider a cloud computing application where a client C wishes to outsource an expensive computation F with input X. The below function runWithWorker shows how C can run F on a generic worker node W using process polymorphism. Once the worker has computed $F\,X$, they will inform a manager process M, who will execute a callback function onFinish.

$$\begin{aligned}
&\text{runWithWorker} : \forall W.\, (t \to t')@\text{C} \to t@\text{C} \to (\text{unit} \to \text{unit})@\text{M} \to t'@\text{C} \\
&\text{runWithWorker } W\ F\ X\ \text{onFinish} = \text{let } W.f \ := F \rightsquigarrow W \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ W.x \ := X \rightsquigarrow W \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \text{C}.\mathit{res} := W.(f\ x) \rightsquigarrow \text{C} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{in } W.\text{“done”} \rightsquigarrow \text{M} \ ; \ \text{M}.(\text{onFinish } ()) \ ; \ \text{C}.\mathit{res}
\end{aligned}$$

The above function does not actually select a worker node; that job falls to M, which maintains a pool of permanent workers, and selects an available worker dynamically to process each request. However, if all workers are busy, M will spawn an ephemeral worker using fork that terminates after a single job. The handleRequest function below implements this functionality.

$$\begin{aligned}
&\text{handleRequest} : (t \to t')@\text{C} \to t@\text{C} \to t'@\text{C} \\
&\text{handleRequest } F\ X = \text{localCase } (\text{M}.\text{acquireWorker}() \rightsquigarrow \{\text{C}\} \cup \text{pool}) \text{ of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \text{some}(w) \Rightarrow \text{let } W := w \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in runWithWorker } W\ F\ X\ \text{M}.(\lambda_.\ \text{releaseWorker } w) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \text{none} \Rightarrow \text{let } W := \text{M}.\text{fork}() \rightsquigarrow \text{C} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \text{in runWithWorker } W\ F\ X\ \text{M}.(\lambda_.\ ())
\end{aligned}$$

M uses acquireWorker, which searches for a free worker and returns some(w) if it finds a free worker w and none otherwise. After alerting all relevant parties to the result, the choreography branches. If the job is run on a free worker, M releases that worker afterward. If not, there is nothing to do afterward, as the newly-spawned process falls out of scope and automatically terminates.

Note that this example critically relies on the ability, inherited from λQC [Samuelson et al. 2025], to send and receive first-class location name representations and reify them into type-level location names. Both the output of acquireWorker and the spawned location name are sent as messages, and the final worker identity is bound to a type-level location.

$$\text{Redices} \quad R \;::=\; \rho.(e_1 \to e_2) \mid \mathsf{App}_\rho \mid \mathsf{CaseInl}_\rho \mid L.m \rightsquigarrow \rho \mid L_1.\mathsf{fork}(L_2, C)$$

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

$$\begin{aligned}
\mathrm{rloc}(\rho.(e_1 \to e_2)) &\triangleq \rho \\
\mathrm{rloc}(\mathsf{App}_\rho) &\triangleq \rho \\
\mathrm{rloc}(\mathsf{CaseInl}_\rho) &\triangleq \rho \\
\mathrm{rloc}(L.m \rightsquigarrow \rho) &\triangleq \{L\} \cup \rho \\
\mathrm{rloc}(L.\mathsf{fork}(L', C)) &\triangleq \{L\}
\end{aligned}$$

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

$$\begin{aligned}
\mathrm{cloc}(X) &\triangleq \varnothing \\
\mathrm{cloc}(\rho.e) &\triangleq \rho \\
\mathrm{cloc}(\mathsf{fun}_\rho\, F(X) := C) &\triangleq \varnothing \\
\mathrm{cloc}(C_1\ \$_\rho\ C_2) &\triangleq \mathrm{cloc}(C_1) \cup \mathrm{cloc}(C_2) \cup \rho \\
\mathrm{cloc}(\mathsf{let}\ \rho.\alpha :: *_{\mathsf{loc}} := C_1\ \mathsf{in}\ C_2) &\triangleq \mathrm{cloc}(C_1) \cup (\mathrm{cloc}(C_2) \setminus \{\alpha\}) \cup \rho \\
\mathrm{cloc}(\mathsf{let}\ (\alpha, x) := \ell.\mathsf{fork}()\ \mathsf{in}\ C) &\triangleq \{\ell\} \cup (\mathrm{cloc}(C) \setminus \{\alpha\})
\end{aligned}$$

Fig. 2. Selected Redices and Location Function Rules. Here m is either a local value v or a selection label d.

4.2 Operational Semantics

The operational semantics of $\lambda\Uparrow$ consists of a small-step relation using a labeled-transition system of the form $\langle C_1, \Omega_1\rangle \overset{R}{\Longrightarrow}_c \langle C_2, \Omega_2\rangle$. The label R represents a redex that tracks the specific reduction occurring. The parameters Ω_1 and Ω_2 track the locations who are executing the choreography before and after the step, respectively, and are used to track which locations remain alive.

Choreographies describe concurrent computation, so the operational semantics includes *out-of-order* reductions to reflect the ability of locations to execute independently. These steps allow unrelated actions to occur in different orders, so long as the order of operations for each individual location is respected. The redices in the step relation specify the step taken and the locations involved, and a step may only be reordered when any computations it is jumping ahead of involve a disjoint set of locations.

We compute the locations involved in a step using the *redex locations* function $\mathrm{rloc}(R)$, and the locations (possibly) involved in an entire choreography using the *choreography locations* function $\mathrm{cloc}(C)$. For example, the redex $\mathsf{A}.v \rightsquigarrow \mathsf{B}$ denotes that A sends v to B. Since precisely A and B participate in this step, $\mathrm{rloc}(\mathsf{A}.v \rightsquigarrow \mathsf{B}) = \{\mathsf{A}, \mathsf{B}\}$. The function $\mathrm{cloc}(C)$, on the other hand, captures all locations that may eventually participate in a step made by C. Thus $\mathrm{cloc}(\mathsf{let}\ \mathsf{A}.x := \mathsf{A}.2\ \mathsf{in}\ (\mathsf{A}.(2 + x) \rightsquigarrow \mathsf{B})) = \{\mathsf{A}, \mathsf{B}\}$, even though A must take multiple steps before B gets involved. Figure 2 shows selected redices and definitions for both location functions.

These two functions together determine when it is safe to reorder steps. Specifically, a step R can execute before an entire computation C if the set of participants in the two are disjoint—$\mathrm{cloc}(C) \cap \mathrm{rloc}(R) = \varnothing$—even if a standard in-order semantics would execute C to completion before R. The following out-of-order rule for let-expressions is an example of such a step.

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

This rule also prohibits the out-of-order step R in the body from including locations binding a variable in the let, and ensures that all locations binding the let have been resolved by requiring $\mathrm{fv}(\rho) = \varnothing$. The second requirement prevents a situation where a location variable later resolves to a location appearing in the step, meaning C-LetI would have rearranged their operations.

Out-of-order execution can similarly occur in branches of case- and localCase-expressions before fully evaluating the scrutinee, with some extra requirements on the steps. Specifically, stepping in the branches is safe when both (1) the locations involved in the step are disjoint from those computing the scrutinee (similarly to C-LetI), and (2) the step will occur regardless of the branch

$$[\text{C-Done}]\ \frac{e_1 \longrightarrow e_2 \qquad \rho \subseteq \Omega}{\langle \rho.e_1, \Omega\rangle \xRightarrow{\rho.(e_1 \to e_2)}_c \langle \rho.e_2, \Omega\rangle} \qquad [\text{C-App}]\ \frac{f = \mathsf{fun}_\rho\, F(X) := C \qquad \mathrm{Val}(V) \qquad \rho \subseteq \Omega}{\langle f\ \$_\rho\ V, \Omega\rangle \xRightarrow{\mathsf{App}_\rho}_c \langle C[F \mapsto f, X \mapsto V], \Omega\rangle}$$

$$[\text{C-SendV}]\ \frac{\mathrm{Val}(v) \qquad L_1 \in \rho_1 \qquad L_1 \in \Omega \qquad \rho_2 \subseteq \Omega}{\langle \rho_1.v\ \{L_1\}\rightsquigarrow \rho_2, \Omega\rangle \xRightarrow{L_1.v \rightsquigarrow \rho_2}_c \langle (\rho_1 \cup \rho_2).v, \Omega\rangle}$$

$$[\text{C-Fork}]\ \frac{\begin{array}{c}L' \text{ globally fresh} \qquad \mathrm{fv}(C') = \varnothing \qquad L \in \Omega \\ C' = C\left[\alpha \mapsto L', x \mapsto \lceil L' \rfloor\right]\end{array}}{\begin{array}{l}\langle \mathsf{let}\ (\alpha, x) := L.\mathsf{fork}()\ \mathsf{in}\ C, \Omega\rangle \\ \quad \xRightarrow{L.\mathsf{fork}(L', C')}_c \left\langle \mathsf{kill}\ L'\ \mathsf{after}\ C', \Omega \cup \{L'\}\right\rangle\end{array}} \qquad [\text{C-Kill}]\ \frac{\mathrm{Val}(V) \qquad L \in \Omega}{\langle \mathsf{kill}\ L\ \mathsf{after}\ V, \Omega\rangle \xRightarrow{\mathsf{kill}(L)}_c \langle V, \Omega \setminus \{L\}\rangle}$$

Fig. 3. Selected $\lambda\Uparrow$ Operational Semantics

taken. The latter point is enforced by requiring identical redices and updates to the set of executing locations in the step in both branches. The result is the following C-LocalCaseI rule, with an analogous rule for choreographic case expressions.

$$[\text{C-LocalCaseI}]\ \frac{\begin{array}{c}\langle C_1, \Omega\rangle \overset{R}{\Longrightarrow}_c \langle C_1', \Omega'\rangle \qquad \langle C_2, \Omega\rangle \overset{R}{\Longrightarrow}_c \langle C_2', \Omega'\rangle \\ \mathrm{cloc}(C) \cap \mathrm{rloc}(R) = \varnothing \qquad \rho \cap \mathrm{rloc}(R) = \varnothing \qquad \mathrm{fv}(\rho) = \varnothing\end{array}}{\left\langle \begin{array}{l}\mathsf{localCase}_\rho\ C\ \mathsf{of} \\ \mid \mathsf{inl}\ x \Rightarrow C_1 \\ \mid \mathsf{inr}\ y \Rightarrow C_2\end{array}, \Omega \right\rangle \overset{R}{\Longrightarrow}_c \left\langle \begin{array}{l}\mathsf{localCase}_\rho\ C\ \mathsf{of} \\ \mid \mathsf{inl}\ x \Rightarrow C_1' \\ \mid \mathsf{inr}\ y \Rightarrow C_2'\end{array}, \Omega' \right\rangle}$$

To see this rule in action, consider the following out-of-order step.

$$\begin{array}{l}\mathsf{localCase}_{\{\mathrm{A},\mathrm{B}\}}\ (\mathrm{A}.e \rightsquigarrow \mathrm{B})\ \mathsf{of} \\ \mid \mathsf{inl}\ x \Rightarrow \mathrm{B}.(1 + x) \rightsquigarrow \mathrm{A}\ ;\ \mathrm{C}.(3 + 2) \\ \mid \mathsf{inr}\ y \Rightarrow \mathrm{C}.(3 + 2)\end{array} \Longrightarrow_c \begin{array}{l}\mathsf{localCase}_{\{\mathrm{A},\mathrm{B}\}}\ (\mathrm{A}.e \rightsquigarrow \mathrm{B})\ \mathsf{of} \\ \mid \mathsf{inl}\ x \Rightarrow \mathrm{B}.(1 + x) \rightsquigarrow \mathrm{A}\ ;\ \mathrm{C}.5 \\ \mid \mathsf{inr}\ y \Rightarrow \mathrm{C}.5\end{array}$$

Although the scrutinee $\mathrm{A}.e \rightsquigarrow \mathrm{B}$ is not yet evaluated, C will run the same program $3 + 2$ on either branch, and C is neither involved in computing the scrutinee nor will their control flow branch. It is thus safe to reduce $\mathrm{C}.(3 + 2)$ to $\mathrm{C}.5$ in both branches. However, no out-of-order step is available in the following choreography.

$$\begin{array}{l}\mathsf{localCase}_{\mathrm{A}}\ (\mathsf{let}\ \mathrm{A}.x := (\mathrm{B}.6 \rightsquigarrow \mathrm{A})\ \mathsf{in}\ \mathrm{A}.(\mathsf{inl}\ (x - 3)))\ \mathsf{of} \\ \mid \mathsf{inl}\ \_ \Rightarrow \mathrm{B}.(3 + 2) \\ \mid \mathsf{inr}\ \_ \Rightarrow \mathrm{B}.(3 + 2)\end{array}$$

Although B executes identical expressions in both branches and will not branch, executing the computation in the branches before the send $\mathrm{B}.6 \rightsquigarrow \mathrm{A}$ in the condition would reorder B's local operations, which is disallowed.

Figure 3 contains a selection of additional rules (the rest can be found in Appendix A). C-Done lifts the local-language semantics to choreographies, C-App applies a function to its argument, and C-SendV formalizes the multiply-located semantics of message-passing. These steps require $\rho \subseteq \Omega$ which ensures that all participants are known and running, meaning everyone who needs to perform this action is able to do so now.

The final two rules formalize how $\lambda\Uparrow$ spawns and kills new locations. To spawn a location, C-Fork selects a globally fresh location name L' and binds α to L' and x to its representation $\lceil L' \rfloor$ in the

body of the fork expression. It checks that the substituted body C' is closed to ensure that L'—which has no access to any enclosing scope—does not attempt to execute a program with free variables. Finally, it wraps C' in a kill-after term to denote that L' should be killed after the computation completes and adds L' to the set Ω of executing locations. Once the computation completes, C-Kill kills the spawned location by removing it from Ω and returning the output of the computation. There is also an out-of-order version of C-Kill that allows a spawned thread L to terminate early when its part of the inner computation C is complete; that is, when $L \notin \mathrm{cloc}(C)$.

4.3 Static Semantics

The static semantics of our language is defined by a kinding judgment and a typing judgment.

4.3.1 $\lambda\Uparrow$ Kinding System. To support polymorphism, we define a kinding judgment $\Gamma \vdash t :: \kappa$, where Γ is a kinding context, t is a type, and κ is a kind. The kind κ classifies t as either a location ($*_{\mathsf{loc}}$), a set of locations ($*_{\mathsf{locset}}$), a local program type ($*_e$), or a program type ($*_\rho$). Figure 4 presents the syntax for these types and kinds.

The kind $*_{\mathsf{loc}}$ represents location names, which can refer to either concrete locations $\mathsf{L} \in \mathcal{L}$ or in-context location variables, while the kind $*_{\mathsf{locset}}$ classifies (non-empty) finite sets of location names, which can be either a type variable, a singleton set ($\{\ell\}$), or a union of sets ($\rho_1 \cup \rho_2$). Types of kind $*_e$ are precisely the types included in the local language under a given type variable context.

The kind $*_\rho$ of program types is similar to the standard program type $*$ of System F and λQC, but is parameterized over a set of locations ρ bounding the locations referenced in a type of that kind. For instance, the type $t_e@\rho$ has kind $*_\rho$, as any value of this type is known by all of the locations in ρ. The idea for other types is similar: if τ has kind $*_\rho$, then only locations in ρ may know any part of a value of this type. The K-Prod and K-Sum rules give two examples of this principle.

$$[\text{K-Prod}]\ \frac{\Gamma \vdash \tau_1 :: *_{\rho_1} \qquad \Gamma \vdash \tau_2 :: *_{\rho_2}}{\Gamma \vdash \tau_1 \times \tau_2 :: *_{\rho_1 \cup \rho_2}} \qquad [\text{K-Sum}]\ \frac{\Gamma \vdash \tau_1 :: *_{\rho_1} \qquad \Gamma \vdash \tau_2 :: *_{\rho_2} \qquad \Gamma \vdash \rho :: *_{\mathsf{locset}} \qquad \rho_1 \cup \rho_2 \subseteq \rho}{\Gamma \vdash \tau_1 +_\rho \tau_2 :: *_\rho}$$

In K-Prod, if a location knows (part of) either side of a pair, then they know part of the entire pair.

One may expect K-Sum to follow a similar rule: collect the annotations on each side. However, sums carry information beyond the underlying types; they also convey if the value is an inl or an inr. The ρ on the plus describes *who knows which side the value is on*, which may include more people than know the data on each side. For instance, a value of type $(\mathsf{int}@\mathsf{A} +_{\{\mathsf{A},\mathsf{B},\mathsf{C}\}} \mathsf{int}@\mathsf{B}) :: *_{\{\mathsf{A},\mathsf{B},\mathsf{C}\}}$ could be an int at either A or B, but all of A, B, *and* C know which. Requiring $\rho_1 \cup \rho_2 \subseteq \rho$ ensures that everyone who might hold data knows whether or not they need to hold that data.

For types including location (set) variables, it is impossible to know which location(s) will be involved. We thus introduce a special value $\top$ that may appear as part of ρ denoting as-yet-unresolved locations. For instance, in the rule K-AllLoc for forall types, variable α may appear free in the set of latent participants ρ (described in Section 4.3.2) and the kind $*_{\rho_\tau}$ of the type τ. To

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

Fig. 4. Syntax of Types and Kinds. Here α is a type variable.

assign a kind to the forall type which captures all referenced locations, we collect the locations in ρ and ρ_τ and replace the bound variable α with $\top$. Since α could resolve to any variable, this matches our intuition for $\top$.

$$\text{[K-AllLoc]}\ \frac{\kappa_\ell \in \{*_{\text{loc}}, *_{\text{locset}}\} \qquad \Gamma, \alpha :: \kappa_\ell \vdash \tau :: *_{\rho_\tau} \qquad \Gamma, \alpha :: \kappa_\ell \vdash \rho :: *_{\text{locset}} \qquad \rho' = ((\rho \cup \rho_\tau) \setminus \alpha) \cup \top}{\Gamma \vdash \forall \alpha :: \kappa_\ell [\rho].\, \tau :: *_{\rho'}}$$

This parameterized kind means $\lambda\Uparrow$ supports a form of bounded polymorphism. If $\alpha :: *_\rho$, then α is restricted in what types it may take on to only those whose participants are in ρ. These bounds make it possible to precisely track the locations that may be involved in a computation, even in the presence of polymorphism. As we will see in Section 4.3.2 below, this tracking is critical to ensuring deadlock freedom with $\lambda\Uparrow$'s combination of closures and thread spawning.

4.3.2 $\lambda\Uparrow$ Type System. Typing judgments in $\lambda\Uparrow$ take the form $\Theta \vdash C : \tau \triangleright \rho$, where $\Theta = \Gamma; \Delta_e; \Delta$ is a three-part context of type, local, and choreographic variables, C is a choreography, τ is a program type, and ρ is a set of participants who may be involved in computing C.

Participant Tracking. Just as the participant parameter ρ on the kind $*_\rho$ bounds the types of that kind, the participant parameter ρ in the typing judgment bounds the locations actively participating in the choreography C. This information is used to ensure that a thread, once killed, will not be asked to perform further computation. To see the challenge in enforcing this guarantee, consider the following program:

$$\text{let } F := \left(\begin{array}{l}\text{let } \alpha := \text{A.fork}() \\ \text{ in } (\lambda\_.\ \text{let A.}x := (\alpha.(1+2) \rightsquigarrow \text{A}) \text{ in A.}x)\end{array}\right) \text{ in } F\ \text{A.}()$$

Here A spawns a thread α who then immediately dies, as the body of the fork expression—the λ-abstraction—is a value. However, the returned abstraction closes over α and, when applied, asks α—which is now dead—to send a message to A, causing deadlock.

The $\lambda\Uparrow$ type system has two key features to prevent this scenario. First, it uses ρ to track which locations might participate in a choreography. For instance, the body of the λ-abstraction above types as

$$\alpha :: *_{\text{loc}} \vdash (\text{let A.}x := (\alpha.(1+2) \rightsquigarrow \text{A}) \text{ in A.}x) : \text{int@A} \triangleright \{\alpha, \text{A}\}.$$

indicating that α and A might participate in the function body, but nobody else will.

Second, we augment function types to include the set of locations who might participate in the body—the function's *latent participants*. Here, for instance, the full λ-abstraction is typed as

$$\alpha :: *_{\text{loc}} \vdash (\lambda\_.\ \text{let A.}x := (\alpha.(1+2) \rightsquigarrow \text{A}) \text{ in A.}x) : \text{unit@A} \xrightarrow{\{\alpha,\text{A}\}} \text{int@A} \triangleright \varnothing$$

with latent participants $\{\alpha, \text{A}\}$ located above the function arrow. Note that $\rho = \varnothing$ here since an abstraction is a value so no locations are involved in computing it. Because the type variable α is free in the function type, the type system can rule out the enclosing fork expression.

By contrast, the program below is valid, since the type $\text{unit@A} \xrightarrow{\{\text{A},\text{B}\}} \text{int@A}$ of the fork's body does not contain α, indicating that it is safe to use the value after α is killed.

$$\vdash \text{let } F := \left(\begin{array}{l}\text{let } \alpha \quad := \text{A.fork}() \rightsquigarrow \text{B} \\ \quad \text{B.}y := \alpha.(1+2) \rightsquigarrow \text{B} \\ \text{ in } (\lambda\_.\ \text{let A.}x := (\text{B.}y \rightsquigarrow \text{A}) \text{ in A.}x)\end{array}\right) \text{ in } F\ \text{A.}() : \text{int@A} \triangleright \{\text{A}, \text{B}\}$$

Type abstractions binding location (set) variables complicate tracking, leading to the use of $\top$ described above. However, such bindings create no deadlock concerns as the type system can guarantee that, when the abstraction is applied, the resolved location(s) are alive.

Typing Rules. The rules T-Fun and T-App formalize the intuition above.

$$[\text{T-Fun}]\ \frac{\Theta, F : \tau_1 \xrightarrow{\rho} \tau_2, X : \tau_1 \vdash C : \tau_2 \rhd \rho \quad \Theta \vdash \tau_1 :: *_{\rho_a} \quad \Theta \vdash \tau_2 :: *_{\rho_b} \quad \rho' = \rho_a \cup \rho_b \cup \rho}{\Theta \vdash \mathsf{fun}_{\rho'}\ F(X) := C : \tau_1 \xrightarrow{\rho} \tau_2 \rhd \varnothing}$$

$$[\text{T-App}]\ \frac{\Theta \vdash C_1 : \tau_1 \xrightarrow{\rho} \tau_2 \rhd \rho_1 \quad \Theta \vdash C_2 : \tau_1 \rhd \rho_2 \quad \Theta \vdash \tau_1 :: *_{\rho_a} \quad \Theta \vdash \tau_2 :: *_{\rho_b} \quad \rho' = \rho_a \cup \rho_b \cup \rho}{\Theta \vdash C_1\ \$_{\rho'}\ C_2 : \tau_2 \rhd \rho_1 \cup \rho_2 \cup \rho'}$$

To type a function, we check that the body C is well-typed with both the function's name F and argument X in scope, and require the participants in the body of the function be the same as the latent participants in the function type. The other three premises of T-Fun ensure that every location who might know the input or the output, or who might participate in the function body, knows the definition of the function.

The application rule T-App ensures that the function is well-typed, and that its argument has the required input type. Similarly to the T-Fun rule, we must ensure that every location who is involved with its body, input, or output performs the application. Lastly, the locations involved in the entire expression are collected from those who are involved in computing C_1 or C_2, and anyone else who performs the application.

T-Fork types the new fork expression. The first two premises are straightforward: the body of the expression must be well-typed with the new location variable α and its first-class representation x in scope, and the parent location ℓ must be well-kinded as a location.

$$[\text{T-Fork}]\ \frac{\Theta, \alpha :: *_{\mathsf{loc}}, \{\ell, \alpha\}.x : \mathsf{loc}_\alpha \vdash C : \tau \rhd \rho \quad \Theta \vdash \ell :: *_{\mathsf{loc}} \quad \Theta \vdash \tau :: *_{\rho_\tau}}{\Theta \vdash \mathsf{let}\ (\alpha, x) := \ell.\mathsf{fork}()\ \mathsf{in}\ C : \tau \rhd \{\ell\} \cup (\rho \setminus \alpha)}$$

$$[\text{T-Kill}]\ \frac{\Theta \vdash C : \tau \rhd \rho}{\Theta \vdash \mathsf{kill}\ L\ \mathsf{after}\ C : \tau \rhd \rho \cup \{L\}}$$

The third requirement—that the type τ of the body is well-kinded *without α in scope*—serves two purposes. First, it prevents type dependency. As the name of the spawned thread is chosen at runtime, we cannot know a-priori which name α will resolve to, so we cannot assign a coherent type to the overall fork expression if that type may depend on the thread's name. Second, it prevents spawned threads from being asked to perform computation after they are killed. Because our type system tracks latent participants, the kinding judgement ensures that the type does not refer to any out-of-scope locations *even in pending computations inside (type) functions*. The rule T-Kill is comparatively straightforward, and simply adds the spawned thread to the set of participants.

The T-TFunLoc and T-TAppLoc rules show how the type system uses $\top$ in tracking when abstracting over locations.

$$[\text{T-TFunLoc}]\ \frac{\kappa_\ell \in \{*_{\mathsf{loc}}, *_{\mathsf{locset}}\} \quad \rho' = (\rho \setminus \alpha) \cup \top \quad \Theta, F : \forall \alpha :: \kappa_\ell[\rho].\, \tau, \alpha :: \kappa_\ell \vdash C : \tau \rhd \rho}{\Theta \vdash \mathsf{tfun}_{\rho'}\ F(\alpha :: \kappa_\ell) := C : \forall \alpha :: \kappa_\ell[\rho].\, \tau \rhd \varnothing}$$

$$[\text{T-TAppLoc}]\ \frac{\kappa_\ell \in \{*_{\mathsf{loc}}, *_{\mathsf{locset}}\} \quad \Theta \vdash C : \forall \alpha :: \kappa_\ell[\rho].\, \tau \rhd \rho_1 \quad \Theta \vdash t :: \kappa_\ell \quad \Theta \vdash \tau[\alpha \mapsto t] :: *_{\rho_\tau} \quad \rho' = \rho_\tau \cup \rho[\alpha \mapsto t]}{\Theta \vdash C\ \$_{\rho'}\ t : \tau[\alpha \mapsto t] \rhd \rho_1 \cup \rho'}$$

In T-Fun above, the location annotation on the function had to identically match the latent participants in the body. When abstracting over (sets of) locations, however, the latent participants can include α, which is free outside the body of the abstraction. We therefore replace α in the annotation on tfun with $\top$ to indicate an unresolved location (set). The rest of the rule is standard.

The type application rule T-TAppLoc is similar to the standard application rule, but since α may be free in type τ and the latent participants ρ of C, we substitute it for the now-resolved variable.

The remaining typing rules can be found in Appendix B.

Example 4 (Fork Bomb)**.** Althought it will run forever and generate an exponentially large number of spawned threads as it runs, a fork bomb does not produce any deadlocks, so the example shown

below is well-typed in our language.

$$\begin{aligned}\mathsf{forkBomb} \;=\; \mathsf{tfun}_{\top}\, F(\ell :: *_{\mathsf{loc}}) \coloneqq \mathsf{let}\ \alpha &\coloneqq \ell.\mathsf{fork}() \\ \beta &\coloneqq \ell.\mathsf{fork}() \\ \mathsf{in}\ F\ \$_{\alpha}\ \alpha\ ;\ F\ \$_{\beta}\ \beta\end{aligned}$$

Specifically, it types as $\vdash \mathsf{forkBomb} : \forall \ell :: *_{\mathsf{loc}}[\ell].\, \mathsf{unit@}\ell \triangleright \varnothing$. Applying the type function to A starts the fork bomb and results in a choreography with the type $\vdash \mathsf{forkBomb}\,\$_{\mathsf{A}}\,\mathsf{A} : \mathsf{unit@A} \triangleright \{\mathsf{A}\}$. Here A is the only participant because no other location has yet been spawned.

4.3.3 Type Soundness. The type system described above enjoys two important notions of soundness: the parameter ρ in the typing judgment captures all locations who may take a step, and the standard guarantee that a well-typed choreography does not get stuck.

To formalize the first notion, recall from Section 4.2 that each step includes a redex R and the $\mathrm{rloc}(R)$ function computes the set of locations involved in R. We therefore show that $\rho \setminus \top$ contains all locations in R for any step. Removing $\top$ is a technical detail to make the theorem meaningful, as all locations are considered to be in ρ if it includes $\top$.

Theorem 1 (Sound Participant Sets). *If* $\Theta \vdash C : \tau \triangleright \rho$ *and* $\langle C, \Omega\rangle \overset{R}{\Longrightarrow}_c \langle C', \Omega'\rangle$, *then* $\mathrm{rloc}(R) \subseteq \rho \setminus \top$.

Note that the soundness of ρ does not extend to multiple steps; if the step spawns a thread, then the participants in the choreography may increase. Our full type preservation theorem, found in Appendix E.2, ensures that C' can always be typed at some set ρ', meaning our type system will always yield a sound set of participants at any given moment in time.

Standard type soundness requires two simple additional premises. First, all locations mentioned in the initial choreography must be running. Second, since kill-after expressions are not intended to be available at the surface level, we only consider choreographies that start without them. The execution may generate kill-after statements without losing any guarantees.

Theorem 2 (Type Soundness). *If* $\vdash C : \tau \triangleright \rho$, *every location literal in* C *is in* Ω, *and* C *contains no* kill-after *expressions, then whenever* $\langle C, \Omega\rangle \Longrightarrow_c^* \langle C', \Omega'\rangle$, *either* C' *is a value, or* $\langle C', \Omega'\rangle$ *can step.*

Note that in Appendix E.2, we have more-traditional progress and preservation theorems. However, our preservation theorem needs significant technical machinery to account for kill-after expressions, so we present only full type soundness here.

5 Network Language

To compile a choreography into multiple programs that a system can execute concurrently, we need to specify the target language that nodes of this system will run. This *network language* proscribes the actions of each individual location in the system, and gives a concurrent operational semantics to describe the execution of the entire system.

5.1 Network Language Syntax

The network language is a concurrent λ-calculus with messages from the same space as in choreographies—local language values and selection messages. The syntax, given in Figure 5, closely mirrors our choreographic syntax, except we split message sends—including selection messages—into two separate constructs to account for the sender and recipient(s).

The return expression $\mathtt{ret}(e)$ is used to execute and yield the output of a local expression e, mirroring the choreographic MLV $\rho.e$. To account for a location $L \notin \rho$—who should not execute e—and other scenarios where a location is not involved in part of the overall choreography, we include a unit value $()$ which does nothing.

To model message-passing, we need two separate constructs for the sender and recipient(s) of a message. Specifically, the send expression $\texttt{send}\ E\ \texttt{to}\ \rho$ multicasts the result of program E to every location in ρ (and also has the sender yield the output of E), and the dual receive expression $\texttt{recv from}\ \ell$ waits to receive a local value from ℓ. Since only local values can be sent, send may only send a value $\texttt{ret}(v)$, while other forms such as $\texttt{send}\ ()\ \texttt{to}\ \rho$ will be stuck.

(Type) let-expressions, (type) functions, case, localCase, and algebraic and recursive data types are included in the network language identically to in choreographies, less some un-needed location (set) annotations. The sequencing construct $E_1\ ;\ E_2$ is standard.

On top of these relatively familiar expressions, there are a two (groups of) constructions that are standard in (process-polymorphic) choreographies. Recall that a location can participate in a case or localCase expression if it either knows the data being branched on *or synchronization messages are inserted telling it which way to go.* In the second case, we need some way to represent that location branching on the result of waiting for a synchronization message. We do this using allow-choice expressions. Specifically, $\texttt{allow}\ \ell\ \texttt{choice}\ (\mathbf{L} \Rightarrow E_1)\ (\mathbf{R} \Rightarrow E_2)$ is the program that waits for a synchronization message from ℓ. If it is $\mathbf{L}$, then it continues as E_1; otherwise, it continues as E_2. Note that if a synchronization message in a choreography is used outside of a branch, then we statically know what message will be received. In this case, we allow an allow-choice expression to only have one branch. Such a term receiving the wrong synchronization message becomes stuck.

Next, "AmI-In" expressions were introduced by λ_{QC} as a generalization of a similar "AmI" construct found in PolyChorλ [Graversen et al. 2024]. Intuitively, it represents a process's knowledge of its own name, and is used to implement process polymorphism. In particular, $\texttt{AmI}\in\ \rho\ \texttt{then}\ E_1\ \texttt{else}\ E_2$ continues as E_1 if the process running it is in ρ, and as E_2 otherwise.

Finally, we implement process forking using fork and exit expressions. The network-level fork expression $\texttt{let}\ (\alpha, x) \coloneqq \texttt{fork}(E_1)\ \texttt{in}\ E_2$ spawns a new thread with the network code E_1, called the *thread task*. The name of this thread is then bound to α while a local representation of this name is bound to x, similar to fork expressions in choreographies. Note that, unlike in a choreography, the thread task is explicitly specified in the term, rather than being implicit from scoping. Dually, the exit command halts execution, removing the location from the system.

5.2 Network Language Operational Semantics

The labeled transition system $L \triangleright E_1 \overset{l}{\Longrightarrow} E_2$ gives the operational semantics of the network language, where L is the location executing the program and l is the label on the step. Selected transition labels and rules are shown in Figure 6.

$$
\begin{array}{llll}
\text{Network Program} & E & ::= & X \mid \texttt{ret}(e) \mid () \mid \texttt{send}\ E\ \texttt{to}\ \rho \mid \texttt{recv from}\ \ell \\
& & \mid & E_1\ ;\ E_2 \mid \texttt{fun}\ F(X) \coloneqq E \mid E_1\ E_2 \mid \texttt{tfun}\ F(\alpha) \coloneqq E \mid E\ t \\
& & \mid & \texttt{let}\ x \coloneqq E_1\ \texttt{in}\ E_2 \mid \texttt{let}\ \alpha :: \kappa \coloneqq E_1\ \texttt{in}\ E_2 \\
& & \mid & \texttt{fold}\ E \mid \texttt{unfold}\ E \mid (E_1, E_2) \mid \texttt{fst}\ E \mid \texttt{snd}\ E \\
& & \mid & \texttt{inl}\ E \mid \texttt{inr}\ E \mid \texttt{case}\ E\ \texttt{of}\ (\texttt{inl}\ X \Rightarrow E_1)\ (\texttt{inr}\ Y \Rightarrow E_2) \\
& & \mid & \texttt{localCase}\ E\ \texttt{of}\ (\texttt{inl}\ x \Rightarrow E_1)\ (\texttt{inr}\ y \Rightarrow E_2) \\
& & \mid & \texttt{allow}\ \ell\ \texttt{choice}\ (d \Rightarrow E) \mid \texttt{allow}\ \ell\ \texttt{choice}\ (\mathbf{L} \Rightarrow E_1)\ (\mathbf{R} \Rightarrow E_2) \\
& & \mid & \texttt{choose}\ d\ \texttt{for}\ \rho\ ;\ E \mid \texttt{AmI}\in\ \rho\ \texttt{then}\ E_1\ \texttt{else}\ E_2 \\
& & \mid & \texttt{let}\ (\alpha, x) \coloneqq \texttt{fork}(E_1)\ \texttt{in}\ E_2 \mid \texttt{exit} \\
\text{Systems} & \Pi & ::= & L_1 \triangleright E_1 \parallel \ldots \parallel L_n \triangleright E_n
\end{array}
$$

Fig. 5. Selected Network Program Syntax. Here $L \in \mathcal{L}$ is a concrete location name.

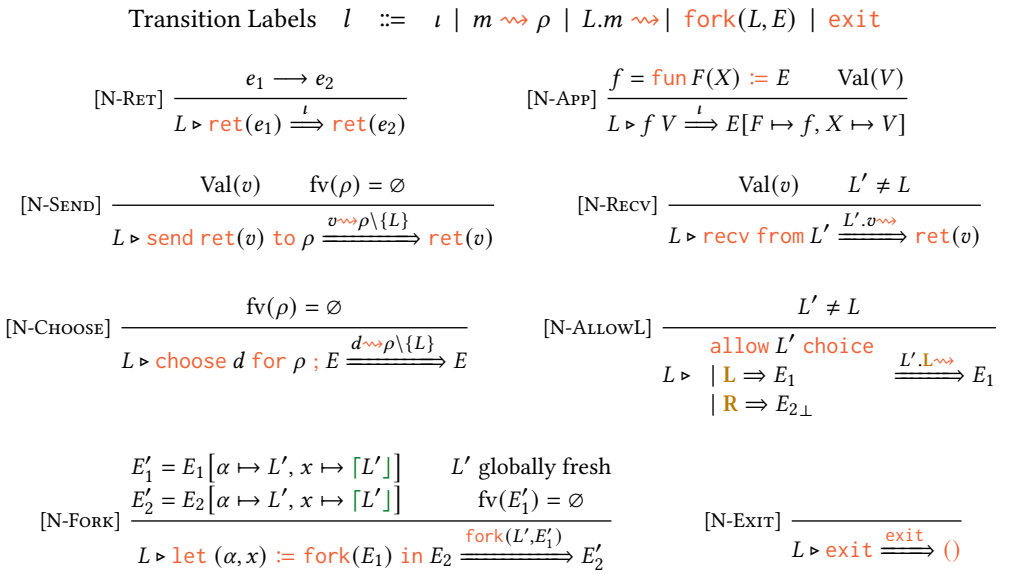


Fig. 6. Selected Network Language Operational Semantics

There are five forms of transition labels, corresponding to five different sorts of steps. The "iota" label ι denotes an internal step, in which the network program or a local program reduces without interaction between other locations.

The send $m \rightsquigarrow \rho$ and receive $L.m \rightsquigarrow$ labels account for message-passing steps, including selection messages. As recipients cannot know the contents of a message in advance, the rules N-Recv and N-AllowL (as well as the omitted N-AllowR rule) are non-deterministic, and allow any value to arrive. The sender follows the N-Send and N-Choose rules, which ensure that the contents of the transition label match the message and recipients specified by the program. The system semantics defined below ensures the sender and recipient agree on the message, resolving the recipient's non-determinism.

The label $\texttt{fork}(L, E)$ indicates the spawning of a new thread and includes the name L of the thread and its thread task E. Note that the N-Fork rule, like the choreographic counterpart C-Fork, ensures that the name L is globally fresh. The dual label `exit` denotes when a thread is killed. While the rule N-Exit has no special effect in this single-location semantics, the corresponding rule in the system semantics below will entirely remove the thread from the system.

5.2.1 Network Systems. Since network programs represent the isolated execution of a single program at a given location, while choreographies represent an entire concurrent system, we need to lift the semantics of our network programs to model an entire system. Formally, we represent a *system* $\Pi = \|_{L \in \Omega}\ (L \triangleright E_L)$ as a map from each location L in a finite set $\Omega \subset \mathcal{L}$ to the network program E_L it is currently executing.

The operational semantics of systems are shown in Figure 7. These rules lift the single-location semantics into a concurrent composition using four rules. The Internal rule allows one location to independently take an internal step. Comm models message passing, and requires the sender and all recipients to simultaneously step with the same message value. Fork spawns a new child thread with a fresh name L', allowing the parent to specify which code the child should run as long as all

$$\text{System Label} \quad l_S \quad ::= \quad \iota_L \mid L_1.m \rightsquigarrow \rho \mid L.\texttt{fork}(L', E) \mid \texttt{kill}(L)$$

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

$$[\textsc{Fork}]\ \frac{L' \text{ globally fresh} \qquad \text{fv}(E_1) = \varnothing \qquad L \triangleright \Pi(L) \xRightarrow{\texttt{fork}(L', E_1)} E_2}{\Pi \xRightarrow{L.\texttt{fork}(L', E_1)}_S \Pi\left[L' \mapsto (E_1\ ;\ \texttt{exit}), L \mapsto E_2\right]} \qquad [\textsc{Kill}]\ \frac{L \triangleright \Pi(L) \xRightarrow{\texttt{exit}} E}{\Pi \xRightarrow{\texttt{kill}(L)}_S \Pi \setminus L}$$

Fig. 7. System Semantics and Labels

variables are resolved. Finally, Kill kills a thread by removing it from the system. Notationally, $\Pi[\rho \mapsto E_L]$ denotes the updated system mapping L to E_L if $L \in \rho$ and $\Pi(L)$ otherwise, and $\Pi \setminus L$ denotes the system which is identical to Π, but removes L from its domain.

6 Endpoint Projection

Having defined our target language, we can now formalize the endpoint projection (EPP) procedure which translates a choreography into a system of concurrently executing locations.

6.1 Network Program Merging

To keep EPP simple and scalable, we would like it to be compositional. However, the case and localCase expressions complicate this desire when non-branching locations participate in the branches. The synchronization messages $\ell[d] \rightsquigarrow \rho$ inform these parties which branch to take, but projecting the branches to a single program requires a *merge operator*. To understand this process, consider the following choreography C.

$$\begin{array}{l} \texttt{localCase A.}e\ \texttt{of}\ \ (\texttt{inl}\ \_ \Rightarrow \texttt{A[L]} \rightsquigarrow \texttt{B}\ ;\ \texttt{B.1})\ \ \texttt{else}\ \ (\texttt{inr}\ \_ \Rightarrow \texttt{A[R]} \rightsquigarrow \texttt{B}\ ;\ \texttt{B.2}) \quad = \quad C \\ \qquad\qquad \llbracket\cdot\rrbracket_{\texttt{B}} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \llbracket\cdot\rrbracket_{\texttt{B}} \\ \texttt{allow A choice} \\ \mid \texttt{L} \Rightarrow \texttt{ret}(1) \end{array} \quad \sqcup \quad \begin{array}{l} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \llbracket\cdot\rrbracket_{\texttt{B}} \\ \texttt{allow A choice} \\ \mid \texttt{R} \Rightarrow \texttt{ret}(2) \end{array} \quad = \quad \begin{array}{l} \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \llbracket\cdot\rrbracket_{\texttt{B}} \\ \texttt{allow A choice} \\ \mid \texttt{L} \Rightarrow \texttt{ret}(1) \\ \mid \texttt{R} \Rightarrow \texttt{ret}(2) \end{array}$$

If the case that the inl branch is executed, A will always send B the selection message L. Therefore in the projection of this branch, B should wait to receive L and then return 1. If instead B receives R in this branch, there are no instructions for what to do (and this can never happen), so the side for R in the allow-choice is missing. Symmetrically if the inr branch is executed, B will only ever receive R, so the R side of this allow-choice returns 2, while the L side is missing.

However, the overall localCase expression contains both branches, so B's projection of this expression must handle both cases. To combine both branches into a single program, we use the merge operator $E_1 \sqcup E_2$: an idempotent binary partial function defined homomorphically on matching network programs. Importantly, this function collects allow-choice branches that exist on only one side, and merges those that exist in both. Our merge operator is identical to the one in

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

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

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

Fig. 8. Selected Merge Operator Definitions

λ_{QC} [Samuelson et al. 2025], accounting for the added `fork` and `exit` constructs. Figure 8 shows the most insight-generating rules, with the full definition available in Appendix D.1.

6.2 Endpoint Projection Definition

With the merge operator in-hand, we can now define EPP. The projection of a choreography C for a location L, denoted $[\![C]\!]_L$, is the network program that executes the actions involving L in C. EPP is partial, both because the merge operator is partial, and because EPP ensures that variables are only used in locations that have bound them. We denote the cases where a choreography fails to project with the notation "undefined," leaving failures due to the merge operator implicit.

EPP is defined in a structurally-recursive manner over the syntax of choreographies. The rules are very similar to those of λ_{QC}, with the exception of functions and applications, where we must account for the annotations which allow a subset of locations to participate in a function body. The majority of rules simply convert choreographic syntax into the network language equivalent, but in some cases—such as those shown in Figure 9—there is more complexity.

For the MLV $\rho.e$, only locations in ρ should compute e while others do nothing. For functions whose bodies may involve locations from ρ, only those locations in ρ project to a function, while others can simply project to a unit value `()`. The projection of function applications is similar, where locations involved in the function body should apply the function, while other locations can simply sequence the function and its argument, afterwards returning a unit value.

Type functions project to network type functions, but those that abstract over locations (and location sets) must behave differently depending on whether or not the variable α resolves to L. Following Graversen et al. [2024] and Samuelson et al. [2025], we use `AmI` to branch on the identity of the current process. In the `then` branch, when the locations match, we substitute α with ℓ in the body C before projecting C. In the `else` branch, we project the body directly. Because location variables are equal only to themselves, this projection correctly treats $\alpha \neq L$.

For the send $C\ \{\ell\}\!\rightsquigarrow \rho$, all locations first execute their projection of C, then location ℓ multicasts the output of C to the locations in ρ, who receive it. For the selection statement $\ell[d] \rightsquigarrow \rho\ ;\ C$, location ℓ sends the choice d to all in ρ, who condition on this choice using an `allow-choice`. As explained in Section 6.1, because the choice is guaranteed, `allow-choice` only has that branch.

For `localCase` expressions, first all locations execute the program in the guard, then locations in ρ branch, while the merge operator combines the branches for others. Since non-branching locations will not know the scrutinee, we also need to ensure that local variables x and y are not free in the projection of the branches. The choreographic `case` expressions has identical rules.

For the new `fork` expression, the parent location ℓ projects to a network `fork` expression with two pieces of code. The body is the projection of the `fork`'s body to ℓ, and the thread task is the

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

$$[\![C_1 \ \$_\rho \ C_2]\!]_L \triangleq \begin{cases} [\![C_1]\!]_L \ [\![C_2]\!]_L & \text{if } L \in \rho \\ [\![C_1]\!]_L \fatsemi [\![C_2]\!]_L \fatsemi () & \text{otherwise} \end{cases}$$

$$[\![\text{fun}_\rho \ F(X) \coloneqq C]\!]_L \triangleq \begin{cases} \texttt{fun}\ F(X) \coloneqq [\![C]\!]_L & \text{if } L \in \rho \\ () & \text{if } L \notin \rho \text{ and } [\![C]\!]_L \neq \text{undefined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![\text{tfun}_\rho \ F(\alpha :: *_{\text{loc}}) \coloneqq C]\!]_L \triangleq \begin{cases} \texttt{tfun}\ F(\alpha) \coloneqq \texttt{AmI}\in \{\alpha\}\ \texttt{then}\ [\![C[\alpha \mapsto L]]\!]_L\ \texttt{else}\ [\![C]\!]_L & \text{if } L \in \rho \\ () & \text{if } L \notin \rho \text{ and } [\![C]\!]_L, [\![C[\alpha \mapsto L]]\!]_L \neq \text{undefined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

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

$$\left[\!\!\left[\begin{array}{l} \text{localCase}_\rho\ C\ \text{of} \\ \mid \text{inl}\ x \Rightarrow C_1 \\ \mid \text{inr}\ y \Rightarrow C_2 \end{array} \right]\!\!\right]_L \triangleq \begin{cases} \texttt{localCase}\ [\![C]\!]_L\ \texttt{of}\ (\text{inl}\ x \Rightarrow [\![C_1]\!]_L)\ (\text{inr}\ y \Rightarrow [\![C_2]\!]_L) & \text{if } L \in \rho \\ [\![C]\!]_L \fatsemi ([\![C_1]\!]_L \sqcup [\![C_2]\!]_L) & \text{if } L \notin \rho \text{ and } x \notin \text{fv}([\![C_1]\!]_L) \text{ and } y \notin \text{fv}([\![C_2]\!]_L) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![\text{let}\ (\alpha, x) \coloneqq \ell.\text{fork}()\ \text{in}\ C]\!]_L \triangleq \begin{cases} \texttt{let}\ (\alpha, x) \coloneqq \texttt{fork}([\![C]\!]_\alpha)\ \texttt{in}\ [\![C]\!]_L & \text{if } L = \ell \\ [\![C]\!]_L & \text{if } L \neq \ell \text{ and } \alpha, x \notin \text{fv}([\![C]\!]_L) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![\text{kill}\ L' \ \text{after}\ C]\!]_L \triangleq \begin{cases} [\![C]\!]_L \fatsemi \texttt{exit} & \text{if } L = L' \\ [\![C]\!]_L & \text{otherwise} \end{cases}$$

Fig. 9. Selected EPP Definitions

projection of the body to the child thread α. That is, the parent projects the body twice: once for its own role, and a second time for the role of the spawned thread. Locations not equal to the parent can simply project to the body of the fork expression, ensuring that neither of the two variables bound in the body are free. The projection of the kill-after expression is simple: everyone performs their role to execute the body, and then the thread associated with the kill-after expression must `exit`, while others continue on.

Instead of directly using the sequencing primitive $E_1 \ ; \ E_2$, note that EPP must use the *collapsing sequencing function* $E_1 \fatsemi E_2$ introduced by Samuelson et al. [2025], which is defined as

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

It may seem that this function is only an optimization, but it is actually required to ensure projected programs can simulate the out-of-order choreographic steps mentioned in Section 4.2. For instance, these steps allow the program $C = \text{let A}.x := \text{A}.e \text{ in B}.(1 + 2)$ to reduce B's local computation to B.3. If EPP used the primitive ; rather than $\fatsemi$, then C would project to $() \text{ ; } \texttt{ret}(1 + 2)$ for B, preventing this step. In reality we project C to $\texttt{ret}(1 + 2)$ so the step can immediately occur.

Combining this collapsing sequencing operator with the involved location tracking necessary to maintain deadlock freedom also allows us to project entire choreographies to $()$ for uninvolved parties. We therefore, essentially for free, achieve most of the goals of modular endpoint projection [Cruz-Filipe et al. 2023]—that $[\![C]\!]_{\text{A}}$ should not depend on parts of C that do not involve A.

Projecting Functions. There are two important notes to be made about the projection of functions and type functions. First, if the latent-participants annotation ρ includes $\top$ then an unresolved location needs to participate. Since it could resolve to any running location, *everyone* must perform the computation. We thus consider $L \in \top$ for all L.

Second, if $L \notin \rho$, the function projects to $()$ because L will not participate in the body, but we still require $[\![C]\!]_L$ to be *defined*. This requirement may seem unnecessary; if L does not participate in the body, one might hope that the body would always project, preferably to $()$. Unfortunately this is not so, which can cause otherwise-projectable choreographies to step to non-projectable ones. To see why, consider the type function

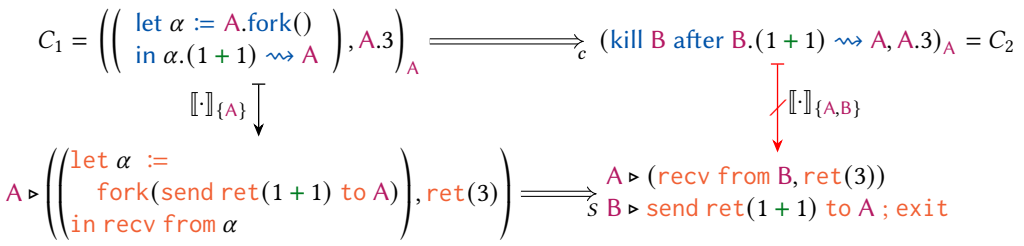
$$C \;=\; \text{tfun}_{\{\text{A}\}\cup\top}\, G(\ell :: *_{\text{loc}}) := \text{if}_{\text{A}}\, X \text{ then } \ell.4 \text{ else } \ell.(3 * 2),$$

which has type $\forall \ell :: *_{\text{loc}}[\text{A}, \ell].\, \text{int@}\ell$ in context $X : \text{bool@A}$. While C projects for A, it does not project for any other location—for instance, B. The then branch of the resulting AmI would need to merge $\texttt{ret}(4) \sqcup \texttt{ret}(3 * 2)$, which is undefined. Wrapping C in a function and applying it gives the well-typed choreography $(\text{fun}_{\text{A}}\, F(X) := C\ \$_{\text{A}}\ \text{A})\ \$_{\text{A}}\ \text{A}.\text{true}$ involving only A. If we did not check that B could project the body, it would project for everyone, but after a β-reduction, it no longer projects for B.

$$\begin{array}{ccc} (\text{fun}_{\text{A}}\, F(X) := C\ \$_{\text{A}}\ \text{A})\ \$_{\text{A}}\ \text{A}.\text{true} & \xrightarrow{\quad c \quad} & C[X \mapsto \text{A}.\text{true}]\ \$_{\text{A}}\ \text{A} \\ \Big\downarrow [\![\cdot]\!]_{\text{B}} & & \Big\downarrow [\![\cdot]\!]_{\text{B}} \\ () \fatsemi () \fatsemi () = () & & \text{undefined} \end{array}$$

B's projection of the type application is $[\![C[X \mapsto \text{A}.\text{true}]]\!]_{\text{B}} \fatsemi ()$, as, in general, B may participate in C even without participating in the final application. However, $\top$ appears in the participant annotation of the tfun, forcing everyone, including B, to project its body, which is not possible. By (unsuccessfully) checking that $[\![C]\!]_{\text{B}}$ is defined initially, we reject the program before the step.

Projecting Systems and Active Threads. While the above EPP definition produces a network program for a single location, choreographies specify the behavior of many participants. Matching the definition in Section 5 of a system of network programs running concurrently, we aim to lift EPP point-wise to a finite set of locations $\Omega \subset \mathcal{L}$. For locations with access to the full choreography, this approach works well. For spawned locations, however, it fails to recognize that they only have code for their thread task, not the whole choreography. Consider the following scenario.



When A spawns thread B in the system semantics on the bottom, B is only aware of the thread task $\texttt{send ret}(1+1)\ \texttt{to A}$—the projection of the fork expression's body to α. However, after the corresponding choreographic step on the top, projecting C_2 to B must sequence the left and right sides of the pair followed by a unit value, producing $[\![C_2]\!]_\text{B} = (\texttt{send ret}(1+1)\ \texttt{to A ; exit})\ \texttt{;}\ ()\ \texttt{;}\ ()$. This discrepancy breaks the correspondence between the choreography and the system.

To account for this, we modify the EPP function to only extract the body of (necessarily unique) kill-after expressions when projecting the location that will be killed. The modified definition, denoted $[\![C]\!]^{\Uparrow}_L$ is defined as follows.

$$[\![C]\!]^{\Uparrow}_L = \begin{cases} [\![C']\!]_L \ \texttt{;}\ \texttt{exit} & \text{if } C \text{ contains a unique subterm } \texttt{kill } L \texttt{ after } C' \\ [\![C]\!]_L & \text{otherwise.} \end{cases}$$

In the example above, this modified EPP will maintain the connection between the choreography and its projection by correctly projecting $[\![C_2]\!]^{\Uparrow}_\text{B} = \texttt{send ret}(1+1)\ \texttt{to A ; exit}$.

To project a full system, we can now lift projection point-wise to each running location using this modified EPP. That is, $[\![C]\!]^{\Uparrow}_\Omega = \|_{L \in \Omega}\ (L \triangleright [\![C]\!]^{\Uparrow}_L)$ for a finite $\Omega \subset \mathcal{L}$. Note that $[\![C]\!]^{\Uparrow}_L$ must be defined for all $L \in \Omega$ for $[\![C]\!]^{\Uparrow}_\Omega$ to be defined.

Example 5 (Fork Bomb Projection)**.** Recall the fork bomb program from in Example 4. The forkBomb type function projects to A (or any other location) with the network program

$$\begin{array}{rl}[\![\text{forkBomb}]\!]_\text{A} = & \texttt{tfun } F(\ell) := \texttt{AmI} \in \{\ell\}\ \texttt{then let } \alpha := \texttt{fork}(F\ \alpha) \\ & \qquad\qquad\qquad\qquad\quad \beta := \texttt{fork}(F\ \beta)\ \texttt{in ()} \\ & \qquad\qquad\qquad \texttt{else ()}\end{array}$$

It is okay that $F\ \alpha$ and $F\ \beta$ are the thread tasks for α and β, respectively, even though F is free in those expressions; before the threads are spawned, F will be replaced with its definition once the originating location applies the outer type function. Specifically, $[\![\text{forkBomb } \$_\text{A}\ \text{A}]\!]_\text{A}$ reduces to the below program, where it is now obvious that α and β will be given the appropriate code to run.

$$\begin{array}{l}\texttt{let } \alpha := \texttt{fork}((\texttt{tfun } F(\ell) := \texttt{AmI} \in \{\ell\}\ \ldots)\ \alpha) \\ \quad \beta := \texttt{fork}((\texttt{tfun } F(\ell) := \texttt{AmI} \in \{\ell\}\ \ldots)\ \beta)\ \texttt{in ()}\end{array}$$

6.3 Soundness, Completeness, and Deadlock Freedom

Note that we provide two separate semantics for $\lambda{\Uparrow}$: the top-level choreographic semantics, and the semantics given by EPP. We now examine the relationship between these semantics and use that relationship to provide a deadlock-freedom-by-design guarantee for compiled systems.

Simulation Relation. To relate the two semantics, we must decide which systems are related to a given choreography. While one may think a choreography C should only relate to its projection $[\![C]\!]^{\Uparrow}_\Omega$, this property is not preserved by reductions. Specifically during branching steps, the branch not taken is discarded in the choreography, but is retained in projected programs waiting on a selection

message. Additionally, EPP's use of the collapsing sequencing function $E_1 \fatsemi E_2$ means programs that resolve to a value after a substitution may be removed from the projected program.

To account for these mismatches we follow traditional choreographic style—specifically, the style of Samuelson et al. [2025]—and define a relation $E_1 \preceq E_2$ which relates two network programs if E_1 may have discarded unneeded code—choices or sequenced values—that remain in E_2. Formally, it is the smallest structurally compatible partial order on network programs that admits the following three rules.

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

To extend this relation to entire systems we can simply lift it point-wise:

$$\Pi_1 \preceq \Pi_2 \triangleq \forall L \in \Omega.\, \Pi_1(L) \preceq \Pi_2(L).$$

This relaxed correspondence is sufficient to yield deadlock freedom of compiled systems. To this end, we prove that the projected semantics simulate the choreographic semantics.

Theorem 3 (Completeness). *If $\Theta \vdash C : \tau \triangleright \rho$, every location literal in C is in Ω, and C contains no* kill-after *expressions, then whenever $\langle C, \Omega\rangle \Longrightarrow_c^* \langle C', \Omega'\rangle$, there is some Π' such that $[\![C]\!]_\Omega^{\Uparrow} \Longrightarrow_S^* \Pi'$ and $[\![C']\!]_{\Omega'}^{\Uparrow} \preceq \Pi'$.*

Note that we have labeled this simulation theorem "completeness." One might expect a traditional accompanying soundness theorem, stating that the choreographic semantics also simulate the projected program. However, this is not true for nonterminating choreographies. While our out-of-order steps mimic the concurrent execution of a projected system, they fail to fully capture all possible execution paths in the presence of non-terminating computations. For example, consider the choreography $(\lambda X.\, \lambda Y.\, Y\ {}_{\{\mathrm{B}\}}\!\rightsquigarrow\ \mathrm{C})\ \$_{\{\mathrm{A,B,C}\}}\ \mathrm{A.loop}\ \$_{\{\mathrm{B,C}\}}\ \mathrm{B}.5$. The only possible choreographic step is to run the infinite loop at A. In the projected system, however, B will eventually send 5 to C. λ_{QC} solves this problem by (1) requiring all locations to synchronize at every function boundary—forcing B and C to wait for A to finish the infinite loop before proceeding—and (2) limiting its soundness result to apply only to choreographies that do not contain infinite loops in *local* programs. Such an approach does not work here, since $\lambda\Uparrow$ allows a selected subset of locations to participate in a β-reduction, meaning that even a *choreographic* function can loop for A while allowing B and C to proceed. This problem with endpoint projection has existed since the advent of functional choreographic programming [Cruz-Filipe et al. 2023; Hirsch and Garg 2022]. See Section 7 for more discussion.

Thus, our operational semantics faces a fork in the road: either require every relevant location to synchronize whenever an infinite loop might occur—negating many benefits of parallelism and possibly requiring locations who do not even know each other exist to synchronize—or give up soundness for non-terminating programs. We choose the latter option, and prove our soundness result, presented below, only for terminating programs.

We say a system Π is *final* if every location in Π maps to a value.

Theorem 4 (Soundness). *If $\vdash C : \tau \triangleright \rho$, every location literal in C is in Ω, C contains no* kill-after *expressions, and $[\![C]\!]_\Omega^{\Uparrow} \Longrightarrow_S^* \Pi$ where Π is final, then $\langle C, \Omega\rangle \Longrightarrow_c^* \langle V, \Omega'\rangle$ where $[\![V]\!]_{\Omega'}^{\Uparrow} \preceq \Pi$.*

Although the relationship between our choreographic semantics and projected semantics is weaker than in other systems, they are powerful enough to prove deadlock freedom. Note that deadlock freedom holds *even for nonterminating choreographies*. Specifically, by combining type

soundness (Theorem 2) and (a strengthening of) EPP completeness (Theorem 3), we prove that either the system terminates in a value for all locations or it can execute forever.

Theorem 5 (Deadlock Freedom). *If* $\vdash C : \tau \triangleright \rho$*, every location literal in* C *is in* Ω*, and* C *contains no* kill-after *expressions, then whenever* $[\![C]\!]^{\Uparrow}_{\Omega} \Longrightarrow_c^* \Pi$*, either* Π *is final or it can step.*

7 Related Work

While $\lambda\Uparrow$ is the first functional choreographic language with process forking, it builds on a rich literature which we review here. First, we discuss the development of functional choreographic programming, including process polymorphism. We then compare the approach for process spawning in $\lambda\Uparrow$ to previous (lower-order) choreographic languages. Finally, we look at process spawning in multiparty session types, the main alternative to choreographic programming.

7.1 Functional Choreographic Programming

Since its inception [Carbone and Montesi 2013; Montesi 2013], the choreographic-programming paradigm has advanced considerably. Early work expanded on core features such as local computations, message passing, and recursion [see e.g., Carbone et al. 2014; Cruz-Filipe and Montesi 2017a,b; Cruz-Filipe et al. 2018; Lanese et al. 2013], but only allowed imperative and procedural computation.

Pirouette [Hirsch and Garg 2022] and Chorλ [Cruz-Filipe et al. 2022]—developed independently—were the first functional choreographic languages. While Chorλ unified the language of choreographies and local computations, Pirouette (like $\lambda\Uparrow$) allowed any language to be used for local computations and messages. Bates et al. [2025] then extended Chorλ with multiply-located values, providing an alternative to synchronization messages.

Process polymorphism was originally developed by Graversen et al. [2024] in an extension to Chorλ called PolyChorλ. This additionally enabled delegation, but required integrating the type-level programming features of System F_ω and seemed to preclude recursive types. Later, Samuelson et al. [2025] developed λQC—the language that $\lambda\Uparrow$ primarily extends—which showed that this complication is the result of Chorλ's choice to combine the language of choreographies and local computations. Moreover, it provided process-*set* polymorphism and first-class location names for the first time. This last feature was critical for the development of $\lambda\Uparrow$.

Defining a top-level semantics for functional choreographies remains a significant challenge. While the original Chorλ work simply punted on correctness of projection, Cruz-Filipe et al. [2023] later provided an out-of-order semantics for Chorλ where EPP was both sound and complete. However, this semantics relied on rewriting rules similar to commuting conversions, which are quite fragile, failing in the presence of language features as simple as named recursive functions [Samuelson et al. 2025]. To provide soundness and completeness guarantees for projection, Pirouette required global synchronization at every function boundary, which λQC extended to its addition of choreographic data types, including sums, pairs, and type abstractions.

Constant global synchronization is unsatisfying at the best of times, but when process forking is available, it is anathema. Thus, $\lambda\Uparrow$ does not require global synchronization. This flexibility comes at a cost: not every evaluation path available to the projected program is available at the choreographic level. However, completeness and confluence of the network semantics allow for the deadlock-freedom guarantee we provide. Moreover, we do get a form of a soundness guarantee for *terminating* programs. This is similar to λQC, which only provides a soundness guarantee when every *local* program terminates.

7.2 Process Spawning in Choreographies

Two papers of which we are aware have previously considered process spawning in lower-order choreographies. The first [Carbone and Montesi 2013] was an imperative language, and therefore lacked most features offered by λ⑂. Importantly, the lack of functions and process polymorphism restricts spawned processes to only be used in a local scope, and prevented even simple examples such as the list-summation example of Section 1.

Cruz-Filipe and Montesi [2016a] provided a highly tailored calculus to implement parallel divide-and-conquer algorithms. It was able to provide processes like parallel merge sort, which the earlier work of Carbone and Montesi [2013] could not, but the language lacks a majority of the features found in λ⑂. For instance, only top-level functions may be process polymorphic, and the language entirely lacks higher-orderedness, avoiding many of the challenges addressed by λ⑂. In addition, processes may only store a single value and must update it through a predefined set of local procedures, such as splitting and merging lists, that must be specially designed for each task.

7.3 Process Spawning in (Multiparty) Session Types

Concurrent programming has always had process spawning as a major feature [Milner 1980; Milner et al. 1992], and thus it has always been prevalent in session types [Caires and Pfenning 2010; Gay and Vasconcelos 2010; Honda 1993; Honda et al. 1998; Wadler 2012]. Traditionally, session types either do not guarantee deadlock freedom [Honda 1993; Honda et al. 1998] or require that processes only communicate in an acyclic topology [Caires and Pfenning 2010; Wadler 2012]. *Multiparty* session types address this deficit, but only for a particular set of communicating processes. In order to allow for process spawning, they must create a new session which includes the new process, and then reason about communication orders between sessions. While this leads to complicated reasoning principles, it can be done [Bettini et al. 2008; Coppo et al. 2013, 2016; Jacobs et al. 2022]. Recently, Le Brun et al. [2025] considered multiparty session types with replication, which allows new processes to spawn in the same session. However, the processes that can be spawned via replication are limited; replication is intended to represent client-server communication, rather than allowing more general techniques like parallel divide-and-conquer.

8 Conclusion

This work introduced λ⑂, the first functional choreographic language to support process forking. λ⑂ retains support for key features of λQC—its predecessor—including higher-order programming, process polymorphism, multiply-located computations, and first-class process names. While this combination of features is powerful, it can introduce complex bugs, such as killing a thread and then attempting to execute a function that closes over its name. λ⑂ prevents these bugs by integrating participant tracking into its type system.

λ⑂ can model complex multi-party computations where arbitrarily many threads are spawned and killed, including a fork bomb. Despite this, we retain the classic choreographic result that the projection of every well-typed choreography is deadlock-free.

As parallel programming becomes more prevalent, the need for languages that can express complex concurrent computations is growing. λ⑂ addresses this need by supporting process forking in tandem with functional-programming features such as higher-order functions and polymorphism. It additionally provides the advantages of choreographic languages, such deadlock freedom by design. Our language thus provides a powerful tool for developers to write parallel programs—such as parallel divide-and-conquer—that are both expressive and safe, allowing them to focus on the logic of their applications rather than the intricacies of concurrency.

Acknowledgments

We would like to thank Andrey Yao and Rahul Krishnan for help editing. Support for this research was provided by the University of Wisconsin–Madison Office of the Vice Chancellor for Research with funding from the Wisconsin Alumni Research Foundation.

References

Mako Bates, Shun Kashiwa, Syed Jafri, Gan Shen, Lindsey Kuper, and Joseph P. Near. 2025. Efficient, Portable, Census-Polymorphic Choreographic Programming. *Proc. ACM Program. Lang.* 9, PLDI, Article 193 (June 2025), 24 pages. https://doi.org/10.1145/3729296

Lorenzo Bettini, Mario Coppo, Loris D'Antoni, Marco De Luca, Mariangiola Dezani-Ciancaglini, and Nobuko Yoshida. 2008. Global Progress in Dynamically Interleaved Multiparty Sessions. In *Concurrency Theory (CONCUR)*. https://doi.org/10.1007/978-3-540-85361-9_33

Luís Caires and Frank Pfenning. 2010. Session Types as Intuitionistic Linear Propositions. In *Concurrency Theory (CONCUR)*. https://doi.org/10.1007/978-3-642-15375-4_16

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

Mario Coppo, Mariangiola Dezani-Ciancaglini, Luca Padovani, and Nobuko Yoshida. 2013. Inference of Global Progress Properties for Dynamically Interleaved Multiparty Sessions. In *Coordination Models and Languages (COORDINATION)*. https://doi.org/10.1007/978-3-642-38493-6_4

Mario Coppo, Mariangiola Dezani-Ciancaglini, Nobuko Yoshida, and Luca Padovani. 2016. Global Progress for Dynamically Interleaved Multiparty Sessions. *Mathematical Structures in Computer Science (MSCS)* 26, 2 (2016), 238–302. https://doi.org/10.1017/S0960129514000188

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. 2016a. Choreographies, Divided and Conquered. (02 2016). https://doi.org/10.48550/arXiv.1602.03729

Luís Cruz-Filipe and Fabrizio Montesi. 2016b. Choreographies in Practice. In *Formal Techniques for Distributed Objects, Components, and Systems (FORTE)*. https://doi.org/10.1007/978-3-319-39570-8_8

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

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

Luís Cruz-Filipe, Fabrizio Montesi, and Marco Peressotti. 2018. Communications in Choreographies, Revisited. In *Symposium on Applied Computing (SAC)*. https://doi.org/10.1145/3167132.3167267

Simon J. Gay and Vasco T. Vasconcelos. 2010. Linear Type Theory for Asynchronous Session Types. *Journal of Functional Programming (JFP)* 20, 1 (2010). https://doi.org/10.1017/S0956796809990268

Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2023. Choral: Object-Oriented Choreographic Programming. *Transactions on Programming Languages and Systems (TOPLAS)* (nov 2023). https://doi.org/10.1145/3632398

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

Kohei Honda. 1993. Types for Dyadic Interaction. In *Concurrency Theory (CONCUR)*. https://doi.org/10.1007/3-540-57208-2_35

Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. 1998. Language Primitives and Type Discipline for Structured Communication-Based Programming. In *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/BFb0053567

Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. 2022. Multiparty GV: Functional Multiparty Session Types with Certified Deadlock Freedom. In *International Conference on Functional Programming (ICFP)*. https://doi.org/10.1145/3547638

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

Matthew Alan Le Brun, Simon Fowler, and Ornela Dardha. 2025. Multiparty Session Types with a Bang! https://doi.org/10.1007/978-3-031-91121-7_6

Robin Milner. 1980. *A calculus of communicating systems*. Lecture Notes in Computer Science, Vol. 92. Springer Berlin Heidelberg. https://doi.org/10.1007/3-540-10235-3

Robin Milner, Joachim Parrow, and David Walker. 1992. A Calculus of Mobile Processes, Part I. *Information and Computation* 100, 1 (1992). https://doi.org/10.1016/0890-5401(92)90008-4

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

Ashley Samuelson, Andrew K. Hirsch, and Ethan Cecchetti. 2025. Choreographic Quick Changes: First-Class Location (Set) Polymorphism. In *Object-Oriented Programming, Systems, Languages & Applications (OOPSLA)*. https://arxiv.org/abs/2506.10913 To Appear, OOPSLA 2025.

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⟩)*.

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

Appendices

A Choreography Operational Semantics

A.1 Choreography Values

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

A.2 Redices and Evaluation Contexts

$$\begin{array}{lrcl} \text{Messages} & m & ::= & v \mid d \\ \text{Redices} & R & ::= & \rho.(e_1 \to e_2) \mid \mathsf{Fun}(R) \mid \mathsf{Arg}(R) \mid \mathsf{App}_\rho \mid \mathsf{TApp}_\rho \mid \mathsf{UnfoldFold}_\rho \\ & & \mid & \mathsf{PairL}(R) \mid \mathsf{PairR}(R) \mid \mathsf{FstPair}_\rho \mid \mathsf{SndPair}_\rho \mid \mathsf{CaseInl}_\rho \mid \mathsf{CaseInr}_\rho \\ & & \mid & \mathsf{let}\ \rho := v \mid \mathsf{let}\ \rho := t \mid L.m \rightsquigarrow \rho \mid L_1.\mathsf{fork}(L_2, C) \mid \mathsf{kill}(L) \end{array}$$

$$\begin{array}{lrcl} \text{Evaluation Contexts} & \eta & ::= & [\cdot]\ C \mid V\ [\cdot] \mid [\cdot]\ t \mid \mathsf{fold}_\rho\ [\cdot] \mid \mathsf{unfold}_\rho\ [\cdot] \\ & & \mid & ([\cdot], C)_\rho \mid (V, [\cdot])_\rho \mid \mathsf{fst}_\rho\ [\cdot] \mid \mathsf{snd}_\rho\ [\cdot] \\ & & \mid & \mathsf{inl}_\rho\ [\cdot] \mid \mathsf{inr}_\rho\ [\cdot] \mid \mathsf{case}_\rho\ [\cdot]\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inr}\ Y \Rightarrow C_2) \\ & & \mid & \mathsf{localCase}_\rho\ [\cdot]\ \mathsf{of}\ (\mathsf{inl}\ x \Rightarrow C_1)\ (\mathsf{inr}\ y \Rightarrow C_2) \\ & & \mid & \mathsf{let}\ \rho.x : t_e := [\cdot]\ \mathsf{in}\ C_2 \mid \mathsf{let}\ \rho.\alpha :: \kappa := [\cdot]\ \mathsf{in}\ C_2 \\ & & \mid & [\cdot]\ \{\ell\}\!\rightsquigarrow \rho \mid \mathsf{kill}\ L\ \mathsf{after}\ [\cdot] \end{array}$$

A.3 Projection of a Redex

For a redex R, its projection $[\![R]\!]_L$ to L is a *list* of network program labels. We denote the empty list as ϵ.

$$[\![\rho.(e_1 \to e_2)]\!]_L = \begin{cases} [\iota] & \text{if } L \in \rho \\ \epsilon & \text{otherwise} \end{cases} \qquad [\![\mathsf{Fun}(R)]\!]_L = [\![R]\!]_L \qquad [\![\mathsf{Arg}(R)]\!]_L = [\![R]\!]_L$$

$$[\![\mathsf{App}_\rho]\!]_L = \begin{cases} [\iota] & \text{if } L \in \rho \\ \epsilon & \text{otherwise} \end{cases} \qquad [\![\mathsf{TApp}_\rho]\!]_L = \begin{cases} [\iota\,,\ \iota] & \text{if } L \in \rho \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\![\mathsf{UnfoldFold}_\rho]\!]_L = \begin{cases} [\iota] & \text{if } L \in \rho \\ \epsilon & \text{otherwise} \end{cases} \qquad [\![\mathsf{PairL}(R)]\!]_L = [\![R]\!]_L \qquad [\![\mathsf{PairR}(R)]\!]_L = [\![R]\!]_L$$

$$[\![\mathsf{FstPair}_\rho]\!]_L = \begin{cases} [\iota] & \text{if } L \in \rho \\ \epsilon & \text{otherwise} \end{cases} \qquad [\![\mathsf{SndPair}_\rho]\!]_L = \begin{cases} [\iota] & \text{if } L \in \rho \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\![\mathsf{CaseInl}_\rho]\!]_L = \begin{cases} [\iota] & \text{if } L \in \rho \\ \epsilon & \text{otherwise} \end{cases} \qquad [\![\mathsf{CaseInr}_\rho]\!]_L = \begin{cases} [\iota] & \text{if } L \in \rho \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\![\mathsf{let}\ \rho := v]\!]_L = \begin{cases} [\iota] & \text{if } L \in \rho \\ \epsilon & \text{otherwise} \end{cases} \qquad [\![\mathsf{let}\ \rho := t]\!]_L = \begin{cases} [\iota\,,\ \iota] & \text{if } L \in \rho \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\![L_1.m \rightsquigarrow \rho_2]\!]_L = \begin{cases} [m \rightsquigarrow \rho] & \text{if } L = L_1 \\ [L_1.m \rightsquigarrow] & \text{if } L \neq L_1 \text{ and } L \in \rho_2 \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\![L_1.\mathsf{fork}(L_2, C)]\!]_L = \begin{cases} [\mathsf{fork}(L_2, E)] & \text{if } L = L_1 \text{ and } [\![C]\!]_{L_2} = E \\ \epsilon & \text{otherwise} \end{cases}$$

$$[\![\mathsf{kill}(L_1)]\!]_L = \begin{cases} [\mathtt{exit}] & \text{if } L = L_1 \\ \epsilon & \text{otherwise} \end{cases}$$

Similarly the projection $[\![R]\!]_{\mathcal{L}}$ of a redex R to all locations is a list of system labels. We denote the concatenation of two lists x and y as $x +\!\!+ y$.

$$[\![\rho.(e_1 \to e_2)]\!]_{\mathcal{L}} = [\iota_L \mid L \in \rho] \qquad [\![\mathsf{Fun}(R)]\!]_{\mathcal{L}} = [\![R]\!]_{\mathcal{L}} \qquad [\![\mathsf{Arg}(R)]\!]_{\mathcal{L}} = [\![R]\!]_{\mathcal{L}}$$

$$\left[\!\left[\mathsf{App}_\rho\right]\!\right]_{\mathcal{L}} = [\iota_L \mid L \in \rho] \qquad \left[\!\left[\mathsf{TApp}_\rho\right]\!\right]_{\mathcal{L}} = [\iota_L \mid L \in \rho] +\!\!+ [\iota_L \mid L \in \rho]$$

$$[\![\mathsf{UnfoldFold}_\rho]\!]_{\mathcal{L}} = [\iota_L \mid L \in \rho] \qquad [\![\mathsf{PairL}(R)]\!]_{\mathcal{L}} = [\![R]\!]_{\mathcal{L}} \qquad [\![\mathsf{PairR}(R)]\!]_{\mathcal{L}} = [\![R]\!]_{\mathcal{L}}$$

$$[\![\mathsf{FstPair}_\rho]\!]_{\mathcal{L}} = [\iota_L \mid L \in \rho] \qquad [\![\mathsf{SndPair}_\rho]\!]_{\mathcal{L}} = [\iota_L \mid L \in \rho] \qquad [\![\mathsf{CaseInl}_\rho]\!]_{\mathcal{L}} = [\iota_L \mid L \in \rho]$$

$$[\![\mathsf{CaseInr}_\rho]\!]_{\mathcal{L}} = [\iota_L \mid L \in \rho] \qquad [\![\mathsf{let}\ \rho := v]\!]_{\mathcal{L}} = [\iota_L \mid L \in \rho]$$

$$[\![\mathsf{let}\ \rho := t]\!]_{\mathcal{L}} = [\iota_L \mid L \in \rho] +\!\!+ [\iota_L \mid L \in \rho] \qquad [\![L_1.m \rightsquigarrow \rho_2]\!]_{\mathcal{L}} = [L_1.m \rightsquigarrow \rho_2]$$

$$[\![L_1.\mathsf{fork}(L_2, C)]\!]_{\mathcal{L}} = \begin{cases} [L_1.\mathtt{fork}(L_2, E)] & \text{if } [\![C]\!]_{L_2} = E \\ \epsilon & \text{otherwise} \end{cases} \qquad [\![\mathsf{kill}(L_1)]\!]_{\mathcal{L}} = [\mathtt{kill}(L_1)]$$

A.4 Redex Blocked Locations

$$\mathrm{rloc}(\rho.(e_1 \to e_2)) = \rho \qquad \mathrm{rloc}(\mathsf{Fun}(R)) = \mathrm{rloc}(R) \qquad \mathrm{rloc}(\mathsf{Arg}(R)) = \mathrm{rloc}(R)$$

$$\mathrm{rloc}(\mathsf{App}_\rho) = \rho \qquad \mathrm{rloc}(\mathsf{TApp}_\rho) = \rho \qquad \mathrm{rloc}(\mathsf{UnfoldFold}_\rho) = \rho \qquad \mathrm{rloc}(\mathsf{PairL}(R)) = \mathrm{rloc}(R)$$

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

$$\mathrm{rloc}(\mathsf{CaseInr}_\rho) = \rho \qquad \mathrm{rloc}(\mathsf{let}\ \rho \coloneqq v) = \rho \qquad \mathrm{rloc}(\mathsf{let}\ \rho \coloneqq t) = \rho$$

$$\mathrm{rloc}(L.m \rightsquigarrow \rho) = \{L\} \cup \rho \qquad \mathrm{rloc}(L_1.\mathsf{fork}(L_2, C)) = \{L_1\} \qquad \mathrm{rloc}(\mathsf{kill}(L)) = \{L\}$$

A.5 Choreography Blocked Locations

$$\mathrm{cloc}(X) = \varnothing \qquad \mathrm{cloc}(\rho.e) = \rho \qquad \mathrm{cloc}(\mathsf{fun}_\rho\, F(X) \coloneqq C) = \varnothing$$

$$\mathrm{cloc}(C_1\ \$_\rho\ C_2) = \mathrm{cloc}(C_1) \cup \mathrm{cloc}(C_2) \cup \rho \qquad \mathrm{cloc}(\mathsf{tfun}_\rho\, F(\alpha) \coloneqq C) = \varnothing$$

$$\mathrm{cloc}(C\ \$_\rho\ t) = \mathrm{cloc}(C) \cup \rho \qquad \mathrm{cloc}(\mathsf{fold}_\rho\ C) = \mathrm{cloc}(C) \qquad \mathrm{cloc}(\mathsf{unfold}_\rho\ C) = \mathrm{cloc}(C) \cup \rho$$

$$\mathrm{cloc}((C_1, C_2)_\rho) = \mathrm{cloc}(C_1) \cup \mathrm{cloc}(C_2) \qquad \mathrm{cloc}(\mathsf{fst}_\rho\ C) = \mathrm{cloc}(C) \cup \rho$$

$$\mathrm{cloc}(\mathsf{snd}_\rho\ C) = \mathrm{cloc}(C) \cup \rho \qquad \mathrm{cloc}(\mathsf{inl}_\rho\ C) = \mathrm{cloc}(C) \qquad \mathrm{cloc}(\mathsf{inr}_\rho\ C) = \mathrm{cloc}(C)$$

$$\mathrm{cloc}(\mathsf{case}_\rho\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inr}\ Y \Rightarrow C_2)) = \mathrm{cloc}(C) \cup \mathrm{cloc}(C_1) \cup \mathrm{cloc}(C_2) \cup \rho$$

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

$$\mathrm{cloc}(\mathsf{let}\ \rho.\alpha :: \kappa \coloneqq C_1\ \mathsf{in}\ C_2) = \mathrm{cloc}(C_1) \cup (\mathrm{cloc}(C_2) \setminus \alpha) \cup \rho$$

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

$$\mathrm{cloc}(\mathsf{let}\ (\alpha, x) \coloneqq \ell.\mathsf{fork}()\ \mathsf{in}\ C) = \{\ell\} \cup (\mathrm{cloc}(C) \setminus \alpha) \qquad \mathrm{cloc}(\mathsf{kill}\ L\ \mathsf{after}\ C) = \{L\} \cup \mathrm{cloc}(C)$$

A.6 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.

$$([\cdot]\ \$_\rho\ C)[R] \triangleq \mathsf{Fun}(R) \qquad (V\ \$_\rho\ [\cdot])[R] \triangleq \mathsf{Arg}(R) \qquad ([\cdot]\ t)_\rho[R] \triangleq R$$

$$(\mathsf{fold}_\rho\ [\cdot])[R] = (\mathsf{unfold}_\rho\ [\cdot])[R] \triangleq R \qquad ([\cdot], C)_\rho[R] \triangleq \mathsf{PairL}(R) \qquad (V, [\cdot])_\rho[R] \triangleq \mathsf{PairR}(R)$$

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

$$(\mathsf{case}_\rho\ [\cdot]\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inr}\ Y \Rightarrow C_2))[R] \triangleq R \qquad ([\cdot]\ \{\ell\}\!\rightsquigarrow \rho)[R] \triangleq R$$

$$(\mathsf{let}\ \rho.x \coloneqq [\cdot]\ \mathsf{in}\ C)[R] = (\mathsf{let}\ \alpha :: \kappa \coloneqq [\cdot]\ \mathsf{in}\ C)[R] \triangleq R \qquad \mathsf{kill}\ L\ \mathsf{after}\ [\cdot][R] \triangleq R$$

A.7 Location Set Relations

Here we define 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 locations and sets in question are non-ground. The principle for how the containment and subset relations behave is that of a modality of *necessity*. For instance, the containment relation $\ell \in \rho$ only holds if the location ℓ is an element of ρ for any possible values that their variables could resolve to. 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.

$$\frac{}{\ell \in \{\ell\}} \qquad \frac{\ell \in \rho_1}{\ell \in \rho_1 \cup \rho_2} \qquad \frac{\ell \in \rho_2}{\ell \in \rho_1 \cup \rho_2}$$

The disjointness relation is defined as expected:

$$(\rho_1 \cap \rho_2 = \varnothing) \triangleq \forall \ell.\, \neg(\ell \in \rho_1 \wedge \ell \in \rho_2)$$

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

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

A.8 Choreography Operational Semantics

[C-Ctx]
$$\frac{\langle C, \Omega\rangle \overset{R}{\Longrightarrow}_c \langle C', \Omega'\rangle}{\langle \eta[C], \Omega\rangle \overset{\eta[R]}{\Longrightarrow}_c \langle \eta[C'], \Omega'\rangle}$$

[C-Done]
$$\frac{e_1 \longrightarrow e_2 \qquad \rho \subseteq \Omega}{\langle \rho.e_1, \Omega\rangle \overset{\rho.(e_1 \to e_2)}{\Longrightarrow}_c \langle \rho.e_2, \Omega\rangle}$$

[C-App]
$$\frac{f = \mathsf{fun}_\rho\, F(X) := C \qquad \mathrm{Val}(V) \qquad \rho \subseteq \Omega}{\langle f\ \$_\rho\ V, \Omega\rangle \overset{\mathsf{App}_\rho}{\Longrightarrow}_c \langle C[F \mapsto f, X \mapsto V], \Omega\rangle}$$

[C-TApp]
$$\frac{f = \mathsf{tfun}_\rho\, F(\alpha) := C \qquad \rho' \subseteq \Omega}{\langle f\ \$_{\rho'}\ t, \Omega\rangle \overset{\mathsf{TApp}_{\rho'}}{\Longrightarrow}_c \langle C[F \mapsto f, \alpha \mapsto t], \Omega\rangle}$$

[C-UnfoldFold]
$$\frac{\mathrm{Val}(V) \qquad \rho \subseteq \Omega}{\langle \mathsf{unfold}_\rho\ (\mathsf{fold}_\rho\ V), \Omega\rangle \overset{\mathsf{UnfoldFold}_\rho}{\Longrightarrow}_c \langle V, \Omega\rangle}$$

[C-FstPair]
$$\frac{\mathrm{Val}(V_1) \qquad \mathrm{Val}(V_2) \qquad \rho \subseteq \Omega}{\langle \mathsf{fst}_\rho\ (V_1, V_2)_\rho, \Omega\rangle \overset{\mathsf{FstPair}_\rho}{\Longrightarrow}_c \langle V_1, \Omega\rangle}$$

[C-SndPair]
$$\frac{\mathrm{Val}(V_1) \qquad \mathrm{Val}(V_2) \qquad \rho \subseteq \Omega}{\langle \mathsf{snd}_\rho\ (V_1, V_2)_\rho, \Omega\rangle \overset{\mathsf{SndPair}_\rho}{\Longrightarrow}_c \langle V_2, \Omega\rangle}$$

[C-CaseInl]
$$\frac{\mathrm{Val}(V) \qquad \rho \subseteq \Omega}{\left\langle \begin{array}{l}\mathsf{case}_\rho\ (\mathsf{inl}_\rho\ V)\ \mathsf{of} \\ \mid \mathsf{inl}\ X \Rightarrow C_1 \\ \mid \mathsf{inr}\ Y \Rightarrow C_2\end{array}, \Omega\right\rangle \overset{\mathsf{CaseInl}_\rho}{\Longrightarrow}_c \langle C_1[X \mapsto V], \Omega\rangle}$$

[C-CaseInr]
$$\frac{\mathrm{Val}(V) \qquad \rho \subseteq \Omega}{\left\langle \begin{array}{l}\mathsf{case}_\rho\ (\mathsf{inr}_\rho\ V)\ \mathsf{of} \\ \mid \mathsf{inl}\ X \Rightarrow C_1 \\ \mid \mathsf{inr}\ Y \Rightarrow C_2\end{array}, \Omega\right\rangle \overset{\mathsf{CaseInr}_\rho}{\Longrightarrow}_c \langle C_2[X \mapsto V], \Omega\rangle}$$

[C-LocalCaseInl]
$$\frac{\mathrm{Val}(v) \qquad \rho \subseteq \Omega}{\left\langle \begin{array}{l}\mathsf{localCase}_\rho\ \rho.(\mathsf{inl}\ v)\ \mathsf{of} \\ \mid \mathsf{inl}\ x \Rightarrow C_1 \\ \mid \mathsf{inr}\ y \Rightarrow C_2\end{array}, \Omega\right\rangle \overset{\mathsf{LocalCaseInl}_\rho}{\Longrightarrow}_c \langle C_1[\rho|x \mapsto v], \Omega\rangle}$$

[C-LocalCaseInr]

$$\frac{\mathrm{Val}(v) \qquad \rho \subseteq \Omega}{\left\langle \begin{array}{l}\mathsf{localCase}_{\rho}\ \rho.(\mathsf{inr}\ v)\ \mathsf{of} \\ \mid \mathsf{inl}\ x \Rightarrow C_1 \\ \mid \mathsf{inr}\ y \Rightarrow C_2\end{array} , \Omega \right\rangle \xrightarrow{\mathsf{LocalCaseInr}_{\rho}}_c \langle C_2[\rho | x \mapsto v], \Omega \rangle}$$

[C-LetV]

$$\frac{\mathrm{Val}(v) \qquad \rho \subseteq \Omega}{\left\langle \mathsf{let}\ \rho.x : t_e := \rho'.v\ \mathsf{in}\ C, \Omega \right\rangle \xrightarrow{\mathsf{let}\ \rho := v}_c \langle C[\rho | x \mapsto v], \Omega \rangle}$$

[C-TyLetV]

$$\frac{\mathrm{Val}(\lceil t \rfloor) \qquad \rho \subseteq \Omega}{\left\langle \mathsf{let}\ \rho.\alpha :: \kappa := \rho'.\lceil t \rfloor\ \mathsf{in}\ C, \Omega \right\rangle \xrightarrow{\mathsf{let}\ \rho.\alpha := t}_c \langle C[\alpha \mapsto t], \Omega \rangle}$$

[C-SendV]

$$\frac{\mathrm{Val}(v) \qquad L_1 \in \rho_1 \qquad L_1 \in \Omega \qquad \rho_2 \subseteq \Omega}{\langle \rho_1.v\ \{L_1\}\rightsquigarrow\ \rho_2, \Omega \rangle \xrightarrow{L_1.v \rightsquigarrow \rho_2}_c \langle (\rho_1 \cup \rho_2).v, \Omega \rangle}$$

[C-Sync]

$$\frac{L \in \Omega \qquad \rho \subseteq \Omega}{\langle L[d] \rightsquigarrow \rho\ ;\ C, \Omega \rangle \xrightarrow{L.d \rightsquigarrow \rho}_c \langle C, \Omega \rangle}$$

[C-SyncI]

$$\frac{\langle C, \Omega \rangle \xrightarrow{R}_c \langle C', \Omega' \rangle \qquad \ell \notin \mathrm{rloc}(R) \qquad \rho \cap \mathrm{rloc}(R) = \varnothing \qquad \mathrm{fv}(\rho) = \mathrm{fv}(\ell) = \varnothing}{\langle \ell[d] \rightsquigarrow \rho\ ;\ C, \Omega \rangle \xrightarrow{R}_c \langle \ell[d] \rightsquigarrow \rho\ ;\ C', \Omega' \rangle}$$

[C-CaseI]

$$\frac{\langle C_1, \Omega \rangle \xrightarrow{R}_c \langle C_1', \Omega' \rangle \qquad \langle C_2, \Omega \rangle \xrightarrow{R}_c \langle C_2', \Omega' \rangle \qquad \mathrm{cloc}(C) \cap \mathrm{rloc}(R) = \varnothing \qquad \rho \cap \mathrm{rloc}(R) = \varnothing \qquad \mathrm{fv}(\rho) = \varnothing}{\left\langle \begin{array}{l}\mathsf{case}_{\rho}\ C\ \mathsf{of} \\ \mid \mathsf{inl}\ X \Rightarrow C_1 \\ \mid \mathsf{inr}\ Y \Rightarrow C_2\end{array} , \Omega \right\rangle \xrightarrow{R}_c \left\langle \begin{array}{l}\mathsf{case}_{\rho}\ C\ \mathsf{of} \\ \mid \mathsf{inl}\ X \Rightarrow C_1' \\ \mid \mathsf{inr}\ Y \Rightarrow C_2'\end{array} , \Omega' \right\rangle}$$

[C-LocalCaseI]

$$\frac{\langle C_1, \Omega \rangle \xrightarrow{R}_c \langle C_1', \Omega' \rangle \qquad \langle C_2, \Omega \rangle \xrightarrow{R}_c \langle C_2', \Omega' \rangle \qquad \mathrm{cloc}(C) \cap \mathrm{rloc}(R) = \varnothing \qquad \rho \cap \mathrm{rloc}(R) = \varnothing \qquad \mathrm{fv}(\rho) = \varnothing}{\left\langle \begin{array}{l}\mathsf{localCase}_{\rho}\ C\ \mathsf{of} \\ \mid \mathsf{inl}\ x \Rightarrow C_1 \\ \mid \mathsf{inr}\ y \Rightarrow C_2\end{array} , \Omega \right\rangle \xrightarrow{R}_c \left\langle \begin{array}{l}\mathsf{localCase}_{\rho}\ C\ \mathsf{of} \\ \mid \mathsf{inl}\ x \Rightarrow C_1' \\ \mid \mathsf{inr}\ y \Rightarrow C_2'\end{array} , \Omega' \right\rangle}$$

[C-AppI]

$$\frac{\langle C_2, \Omega \rangle \xrightarrow{R}_c \langle C_2', \Omega' \rangle \qquad \mathrm{cloc}(C_1) \cap \mathrm{rloc}(R) = \varnothing}{\langle C_1\ \$_{\rho}\ C_2, \Omega \rangle \xrightarrow{R}_c \langle C_1\ \$_{\rho}\ C_2', \Omega' \rangle}$$

[C-PairI]

$$\frac{\langle C_2, \Omega \rangle \xrightarrow{R}_c \langle C_2', \Omega' \rangle \qquad \mathrm{cloc}(C_1) \cap \mathrm{rloc}(R) = \varnothing}{\langle (C_1, C_2)_{\rho}, \Omega \rangle \xrightarrow{R}_c \langle (C_1, C_2')_{\rho}, \Omega' \rangle}$$

[C-LetI]

$$\frac{\langle C_2, \Omega \rangle \xrightarrow{R}_c \langle C_2', \Omega' \rangle \qquad \mathrm{cloc}(C_1) \cap \mathrm{rloc}(R) = \varnothing \qquad \rho \cap \mathrm{rloc}(R) = \varnothing \qquad \mathrm{fv}(\rho) = \varnothing}{\langle \mathsf{let}\ \rho.x : t_e := C_1\ \mathsf{in}\ C_2, \Omega \rangle \xrightarrow{R}_c \langle \mathsf{let}\ \rho.x : t_e := C_1\ \mathsf{in}\ C_2', \Omega' \rangle}$$

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

[C-Fork]
$$\frac{L'\ \text{globally fresh} \quad \mathrm{fv}(C') = \varnothing \quad L \in \Omega \quad C' = C\left[\alpha \mapsto L', x \mapsto \lceil L' \rfloor\right]}{\langle \mathsf{let}\ (\alpha, x) := L.\mathsf{fork}()\ \mathsf{in}\ C, \Omega\rangle \overset{L.\mathsf{fork}(L', C')}{\Longrightarrow}_c \langle \mathsf{kill}\ L'\ \mathsf{after}\ C', \Omega \cup \{L'\}\rangle}$$

[C-ForkI]
$$\frac{\langle C, \Omega\rangle \overset{R}{\Longrightarrow}_c \langle C', \Omega'\rangle \quad L \notin \mathrm{rloc}(R)}{\langle \mathsf{let}\ (\alpha, x) := L.\mathsf{fork}()\ \mathsf{in}\ C, \Omega\rangle \overset{R}{\Longrightarrow}_c \langle \mathsf{let}\ (\alpha, x) := L.\mathsf{fork}()\ \mathsf{in}\ C', \Omega'\rangle}$$

[C-Kill]
$$\frac{\mathrm{Val}(V) \quad L \in \Omega}{\langle \mathsf{kill}\ L\ \mathsf{after}\ V, \Omega\rangle \overset{\mathsf{kill}(L)}{\Longrightarrow}_c \langle V, \Omega \setminus \{L\}\rangle}$$

[C-KillI]
$$\frac{L \notin \mathrm{cloc}(C) \quad L \in \Omega}{\langle \mathsf{kill}\ L\ \mathsf{after}\ C, \Omega\rangle \overset{\mathsf{kill}(L)}{\Longrightarrow}_c \langle V, \Omega \setminus \{L\}\rangle}$$

B Static Semantics

B.1 $\lambda\Uparrow$ Kinding System

First we note that, in order to prevent kind dependency, the kinding context should be split into two contexts—Γ_ℓ for locations and location sets of kind $\kappa_\ell \in \{*_{\mathsf{loc}}, *_{\mathsf{locset}}\}$, and Γ for local and program kinds. We have elided this detail from the presentation of the kinding and typing rules for simplicity, but fully address the details in Appendix E.1 and subsequent appendices.

[K-Var] $\dfrac{\vdash \Gamma \quad \alpha :: \kappa \in \Gamma}{\Gamma \vdash \alpha :: \kappa}$ [K-Loc] $\dfrac{\mathsf{L} \in \mathcal{L}}{\Gamma \vdash \mathsf{L} :: *_{\mathsf{loc}}}$ [K-Sng] $\dfrac{\Gamma \vdash \ell :: *_{\mathsf{loc}}}{\Gamma \vdash \{\ell\} :: *_{\mathsf{locset}}}$

[K-Union] $\dfrac{\Gamma \vdash \rho_1 :: *_{\mathsf{locset}} \quad \Gamma \vdash \rho_2 :: *_{\mathsf{locset}}}{\Gamma \vdash \rho_1 \cup \rho_2 :: *_{\mathsf{locset}}}$ [K-Local] $\dfrac{\vdash \Gamma \quad \Gamma \Vdash t_e :: *_e}{\Gamma \vdash t_e :: *_e}$

[K-At] $\dfrac{\Gamma \vdash t_e :: *_e \quad \Gamma \vdash \rho :: *_{\mathsf{locset}}}{\Gamma \vdash t_e @ \rho :: *_\rho}$ [K-Arrow] $\dfrac{\Gamma \vdash \rho :: *_{\mathsf{locset}} \quad \Gamma \vdash \tau_1 :: *_{\rho_1} \quad \Gamma \vdash \tau_2 :: *_{\rho_2}}{\Gamma \vdash \tau_1 \xrightarrow{\rho} \tau_2 :: *_{\rho_1 \cup \rho_2 \cup \rho}}$

[K-Prod] $\dfrac{\Gamma \vdash \tau_1 :: *_{\rho_1} \quad \Gamma \vdash \tau_2 :: *_{\rho_2}}{\Gamma \vdash \tau_1 \times \tau_2 :: *_{\rho_1 \cup \rho_2}}$ [K-Sum] $\dfrac{\Gamma \vdash \tau_1 :: *_{\rho_1} \quad \Gamma \vdash \tau_2 :: *_{\rho_2} \quad \Gamma \vdash \rho :: *_{\mathsf{locset}} \quad \rho_1 \cup \rho_2 \subseteq \rho}{\Gamma \vdash \tau_1 +_\rho \tau_2 :: *_\rho}$

[K-Rec] $\dfrac{\Gamma, \alpha :: *_\rho \vdash \tau :: *_\rho}{\Gamma \vdash \mu_\rho \alpha.\, \tau :: *_\rho}$ [K-AllLoc] $\dfrac{\kappa_\ell \in \{*_{\mathsf{loc}}, *_{\mathsf{locset}}\} \quad \Gamma, \alpha :: \kappa_\ell \vdash \tau :: *_{\rho_\tau} \quad \Gamma, \alpha :: \kappa_\ell \vdash \rho :: *_{\mathsf{locset}} \quad \rho' = ((\rho \cup \rho_\tau) \setminus \alpha) \cup \top}{\Gamma \vdash \forall \alpha :: \kappa_\ell [\rho].\, \tau :: *_{\rho'}}$

[K-AllLocal] $\dfrac{\Gamma, \alpha :: *_e \vdash \tau :: *_\rho \quad \Gamma \vdash \rho' :: *_{\mathsf{loc}}}{\Gamma \vdash \forall \alpha :: *_e [\rho'].\, \tau :: *_{\rho \cup \rho'}}$ [K-All] $\dfrac{\Gamma, \alpha :: *_{\rho_1} \vdash \tau :: *_{\rho_2} \quad \Gamma \vdash \rho' :: *_{\mathsf{loc}}}{\Gamma \vdash \forall \alpha :: *_{\rho_1} [\rho'].\, \tau :: *_{\rho_2 \cup \rho'}}$

B.2 $\lambda\Uparrow$ Type System

In these rules we denote $\Theta = \Gamma; \Delta_e; \Delta$ for brevity, and abuse notation to add variables to and use the kinding judgment on the required sub-contexts of Θ as appropriate. In the subsequent sections we may use the shorthand $\mathrm{tloc}(\Theta; \tau)$ to denote the set ρ of locations such that $\Theta \vdash \tau :: *_\rho$.

$$[\text{T-Var}]\ \frac{\vdash \Theta \qquad X : \tau \in \Theta}{\Theta \vdash X : \tau \rhd \varnothing} \qquad [\text{T-Done}]\ \frac{\vdash \Theta \qquad \Theta \vdash \rho :: *_{\text{locset}} \qquad \Theta|_\rho \Vdash e : t_e \qquad \rho' = \text{if Val}(e) \text{ then } \varnothing \text{ else } \rho}{\Theta \vdash \rho.e : t_e@\rho \rhd \rho'}$$

$$[\text{T-Fun}]\ \frac{\Theta, F : \tau_1 \xrightarrow{\rho} \tau_2, X : \tau_1 \vdash C : \tau_2 \rhd \rho \qquad \Theta \vdash \tau_1 :: *_{\rho_a} \qquad \Theta \vdash \tau_2 :: *_{\rho_b} \qquad \rho' = \rho_a \cup \rho_b \cup \rho}{\Theta \vdash \mathsf{fun}_{\rho'}\ F(X) := C : \tau_1 \xrightarrow{\rho} \tau_2 \rhd \varnothing}$$

$$[\text{T-App}]\ \frac{\Theta \vdash C_1 : \tau_1 \xrightarrow{\rho} \tau_2 \rhd \rho_1 \qquad \Theta \vdash C_2 : \tau_1 \rhd \rho_2 \qquad \Theta \vdash \tau_1 :: *_{\rho_a} \qquad \Theta \vdash \tau_2 :: *_{\rho_b} \qquad \rho' = \rho_a \cup \rho_b \cup \rho}{\Theta \vdash C_1\ \$_{\rho'}\ C_2 : \tau_2 \rhd \rho_1 \cup \rho_2 \cup \rho'}$$

$$[\text{T-TFunLoc}]\ \frac{\kappa_\ell \in \{*_{\text{loc}}, *_{\text{locset}}\} \qquad \rho' = (\rho \setminus \alpha) \cup \top \qquad \Theta, F : \forall \alpha :: \kappa_\ell[\rho].\ \tau, \alpha :: \kappa_\ell \vdash C : \tau \rhd \rho}{\Theta \vdash \mathsf{tfun}_{\rho'}\ F(\alpha :: \kappa_\ell) := C : \forall \alpha :: \kappa_\ell[\rho].\ \tau \rhd \varnothing}$$

$$[\text{T-TAppLoc}]\ \frac{\kappa_\ell \in \{*_{\text{loc}}, *_{\text{locset}}\} \qquad \Theta \vdash C : \forall \alpha :: \kappa_\ell[\rho].\ \tau \rhd \rho_1 \qquad \Theta \vdash t :: \kappa_\ell \qquad \Theta \vdash \tau[\alpha \mapsto t] :: *_{\rho_\tau} \qquad \rho' = \rho_\tau \cup \rho[\alpha \mapsto t]}{\Theta \vdash C\ \$_{\rho'}\ t : \tau[\alpha \mapsto t] \rhd \rho_1 \cup \rho'}$$

$$[\text{T-TFun}]\ \frac{\kappa \in \{*_e, *_\rho\} \qquad \Theta, F : \forall \alpha :: \kappa[\rho].\ \tau, \alpha :: \kappa \vdash C : \tau \rhd \rho \qquad \Theta, \alpha :: \kappa \vdash \tau :: *_{\rho_\tau} \qquad \rho' = \rho_\tau \cup \rho}{\Theta \vdash \mathsf{tfun}_{\rho'}\ F(\alpha :: \kappa) := C : \forall \alpha :: \kappa[\rho].\ \tau \rhd \varnothing}$$

$$[\text{T-TApp}]\ \frac{\kappa \in \{*_e, *_\rho\} \qquad \Theta \vdash C : \forall \alpha :: \kappa[\rho].\ \tau \rhd \rho_1 \qquad \Theta \vdash t :: \kappa \qquad \Theta \vdash \tau[\alpha \mapsto t] :: *_{\rho_\tau} \qquad \rho' = \rho_\tau \cup \rho}{\Theta \vdash C\ \$_{\rho'}\ t : \tau[\alpha \mapsto t] \rhd \rho_1 \cup \rho'}$$

$$[\text{T-Pair}]\ \frac{\Theta \vdash C_1 : \tau_1 \rhd \rho_1 \qquad \Theta \vdash C_2 : \tau_2 \rhd \rho_2 \qquad \Theta \vdash \tau_1 :: *_{\rho_a} \qquad \Theta \vdash \tau_2 :: *_{\rho_a} \qquad \rho = \rho_a \cup \rho_b}{\Theta \vdash (C_1, C_2)_\rho : \tau_1 \times \tau_2 \rhd \rho_1 \cup \rho_2}$$

$$[\text{T-Fst}]\ \frac{\Theta \vdash C : \tau_1 \times \tau_2 \rhd \rho' \qquad \Theta \vdash \tau_1 :: *_{\rho_a} \qquad \Theta \vdash \tau_2 :: *_{\rho_a} \qquad \rho = \rho_a \cup \rho_b}{\Theta \vdash \mathsf{fst}_\rho\ C : \tau_1 \rhd \rho \cup \rho'}$$

$$[\text{T-Snd}]\ \frac{\Theta \vdash C : \tau_1 \times \tau_2 \rhd \rho' \qquad \Theta \vdash \tau_1 :: *_{\rho_a} \qquad \Theta \vdash \tau_2 :: *_{\rho_a} \qquad \rho = \rho_a \cup \rho_b}{\Theta \vdash \mathsf{snd}_\rho\ C : \tau_2 \rhd \rho \cup \rho'}$$

$$[\text{T-Inl}]\ \frac{\Theta \vdash C : \tau_1 \rhd \rho \qquad \Theta \vdash \rho' :: *_{\text{locset}} \qquad \Theta \vdash \tau_1 :: *_{\rho_a} \qquad \Theta \vdash \tau_2 :: *_{\rho_a} \qquad \rho_a \cup \rho_b \subseteq \rho'}{\Theta \vdash \mathsf{inl}_{\rho'}\ C : \tau_1 +_{\rho'} \tau_2 \rhd \rho}$$

$$[\text{T-Inr}]\ \frac{\Theta \vdash C : \tau_2 \rhd \rho \qquad \Theta \vdash \rho' :: *_{\text{locset}} \qquad \Theta \vdash \tau_1 :: *_{\rho_a} \qquad \Theta \vdash \tau_2 :: *_{\rho_a} \qquad \rho_a \cup \rho_b \subseteq \rho'}{\Theta \vdash \mathsf{inr}_{\rho'}\ C : \tau_1 +_{\rho'} \tau_2 \rhd \rho}$$

$$[\text{T-Case}]\ \frac{\Theta \vdash C : \tau_1 +_{\rho'} \tau_2 \rhd \rho \qquad \Theta, X : \tau_1 \vdash C_1 : \tau \rhd \rho_1 \qquad \Theta, Y : \tau_2 \vdash C_2 : \tau \rhd \rho_2}{\begin{array}{l}\mathsf{case}_{\rho'}\ C\ \mathsf{of} \\ \Theta \vdash \quad \mid \mathsf{inl}\ X \Rightarrow C_1 \quad : \tau \rhd \rho \cup \rho' \cup \rho_1 \cup \rho_2 \\ \quad\ \mid \mathsf{inr}\ Y \Rightarrow C_2\end{array}}$$

$$[\text{T-LocalCase}]\ \frac{\begin{array}{c}\text{isSum}(s,t_1,t_2) \qquad \Theta \vdash C : s@\rho' \rhd \rho \\ \Theta, \rho'.x : t_1 \vdash C_1 : \tau \rhd \rho_1 \qquad \Theta, \rho'.y : t_2 \vdash C_2 : \tau \rhd \rho_2\end{array}}{\begin{array}{ll}\Theta \vdash & \text{localCase}_{\rho'}\ C\ \text{of} \\ & \mid \text{inl}\ x \Rightarrow C_1 \\ & \mid \text{inr}\ y \Rightarrow C_2\end{array} \quad : \tau \rhd \rho \cup \rho' \cup \rho_1 \cup \rho_2}$$

$$[\text{T-Fold}]\ \frac{\Theta \vdash C : \tau[\alpha \mapsto \mu_\rho \alpha.\, \tau] \rhd \rho'}{\Theta \vdash \text{fold}_\rho\ C : \mu_\rho \alpha.\, \tau \rhd \rho'} \qquad [\text{T-Unfold}]\ \frac{\Theta \vdash C : \mu_\rho \alpha.\, \tau \rhd \rho'}{\Theta \vdash \text{unfold}_\rho\ C : \tau[\alpha \mapsto \mu\alpha.\, \tau] \rhd \rho \cup \rho'}$$

$$[\text{T-LetLocal}]\ \frac{\begin{array}{c}\Theta \vdash C_1 : t_e@\rho_2 \rhd \rho \qquad \rho_1 \subseteq \rho_2 \\ \Theta, \rho_1.x : t_e \vdash C_2 : \tau \rhd \rho'\end{array}}{\Theta \vdash \text{let}\ \rho_1.x \coloneqq C_1\ \text{in}\ C_2 : \tau \rhd \rho \cup \rho' \cup \rho_1}$$

$$[\text{T-LetLoc}]\ \frac{\begin{array}{c}\Theta \vdash C_1 : \text{loc}_{\rho_1}@\rho_3 \rhd \rho \qquad \rho_1 \subseteq \rho_2 \subseteq \rho_3 \\ \Theta \vdash \tau :: *_{\rho_\tau} \qquad \Theta, \alpha :: *_{\text{loc}} \vdash C_2 : \tau \rhd \rho'\end{array}}{\Theta \vdash \text{let}\ \rho_2.\alpha :: *_{\text{loc}} \coloneqq C_1\ \text{in}\ C_2 : \tau \rhd \rho \cup (\rho' \setminus \{\alpha\}) \cup \rho_2}$$

$$[\text{T-LetLocSet}]\ \frac{\begin{array}{c}\Theta \vdash C_1 : \text{locset}_{\rho_1}@\rho_3 \rhd \rho \qquad \rho_1 \subseteq \rho_2 \subseteq \rho_3 \\ \Gamma \vdash \tau :: *_{\rho_\tau} \qquad \Theta, \alpha :: *_{\text{locset}} \vdash C_2 : \tau \rhd \rho'\end{array}}{\Theta \vdash \text{let}\ \rho_2.\alpha :: *_{\text{locset}} \coloneqq C_1\ \text{in}\ C_2 : \tau \rhd \rho \cup (\rho' \setminus \alpha) \cup \rho_2}$$

$$[\text{T-Send}]\ \frac{\begin{array}{c}\Theta \vdash C : t_e@\rho_1 \rhd \rho \\ \ell_1 \in \rho_1 \qquad \Theta \vdash \rho_2 :: *_{\text{locset}}\end{array}}{\Theta \vdash C\ \{\ell_1\}\rightsquigarrow\ \rho_2 : t_e@(\rho_1 \cup \rho_2) \rhd \rho \cup \{\ell_1\} \cup \rho_2} \qquad [\text{T-Sync}]\ \frac{\begin{array}{c}\Theta \vdash C : \tau \rhd \rho' \\ \Theta \vdash \ell :: *_{\text{loc}} \qquad \Theta \vdash \rho :: *_{\text{locset}}\end{array}}{\Theta \vdash \ell[d] \rightsquigarrow \rho\ ;\ C : \tau \rhd \{\ell\} \cup \rho \cup \rho'}$$

$$[\text{T-Fork}]\ \frac{\begin{array}{c}\Theta, \alpha :: *_{\text{loc}}, \{\ell, \alpha\}.x : \text{loc}_\alpha \vdash C : \tau \rhd \rho \\ \Theta \vdash \ell :: *_{\text{loc}} \qquad \Theta \vdash \tau :: *_{\rho_\tau}\end{array}}{\Theta \vdash \text{let}\ (\alpha, x) \coloneqq \ell.\text{fork}()\ \text{in}\ C : \tau \rhd \{\ell\} \cup (\rho \setminus \alpha)} \qquad [\text{T-Kill}]\ \frac{\Theta \vdash C : \tau \rhd \rho}{\Theta \vdash \text{kill}\ L\ \text{after}\ C : \tau \rhd \rho \cup \{L\}}$$

B.3 Spawned Thread Well-Scopedness Judgment

Because $\lambda\Uparrow$ programs may spawn multiple threads, we need to guarantee that the names of these spawned locations are distinct from other threads, and also distinct from any other locations who may have already been executing the choreography.

For instance, consider the simple program (kill B after A.1, B.2 ⇝ A). While it may be obvious that such a program cannot be reached by means of reducing a surface-language expression (as B could not be chosen as the spawned thread), the type system defined above does not rule it out. Indeed, the specific issue we identify with this program is that the participant B in the right-hand side B.2 ⇝ A is one of the spawned locations in the left-hand side kill B after A.1. The expression (kill B after A.1, kill B after A.2) should be a similarly impossible state to reach as the thread B has been spawned in two separate expressions, which, by the C-Fork rule, could not occur as each simultaneously spawned thread must be distinct.

To rule out scenarios like those described above, we use a secondary judgment $\Theta \vdash C$ `loc-ok` that ensures the spawned threads in a choreography C are well-scoped—only participating in the scope of the kill-after expression that they are declared to be operating in.

In the following rules, the syntax $\Theta \vdash C \rhd \rho$ is used to mean that C type-checks under context Θ and with participants ρ, but the type may be arbitrary. In effect, $\Theta \vdash C \rhd \rho \Leftrightarrow \exists \tau.\, \Theta \vdash C : \tau \rhd \rho$.

$$[\text{S-Var}]\ \frac{}{\Theta \vdash X\ \texttt{loc-ok}} \qquad [\text{S-Done}]\ \frac{}{\Theta \vdash \rho.e\ \texttt{loc-ok}} \qquad [\text{S-Fun}]\ \frac{\Theta, F : \tau_1 \xrightarrow{\rho} \tau_2, X : \tau_1 \vdash C\ \texttt{loc-ok} \quad \mathrm{SL}(C) = \varnothing}{\Theta \vdash \mathsf{fun}_\rho\ F(X) := C\ \texttt{loc-ok}}$$

$$[\text{S-App}]\ \frac{\begin{array}{c}\Theta \vdash C_1\ \texttt{loc-ok} \quad \Theta \vdash C_2\ \texttt{loc-ok} \\ \Theta \vdash C_1 \rhd \rho_1 \quad \Theta \vdash C_2 \rhd \rho_2 \\ \mathrm{NL}(\rho_1) \cap \mathrm{SL}(C_2) = \varnothing \\ \mathrm{NL}(\rho_2) \cap \mathrm{SL}(C_1) = \varnothing \\ \mathrm{NL}(\rho) \cap (\mathrm{SL}(C_1) \cup \mathrm{SL}(C_2)) = \varnothing\end{array}}{\Theta \vdash C_1\ \$_\rho\ C_2\ \texttt{loc-ok}} \qquad [\text{S-TFun}]\ \frac{\Theta, F : \forall \alpha :: \kappa[\rho].\, \tau \vdash C\ \texttt{loc-ok} \quad \mathrm{SL}(C) = \varnothing}{\Theta \vdash \mathsf{tfun}_\rho\ F(\alpha :: \kappa_\ell) := C\ \texttt{loc-ok}}$$

$$[\text{S-TApp}]\ \frac{\Theta \vdash C\ \texttt{loc-ok} \quad \mathrm{NL}(\rho) \cap \mathrm{SL}(C) = \varnothing}{\Theta \vdash C\ \$_\rho\ t\ \texttt{loc-ok}} \qquad [\text{S-Pair}]\ \frac{\begin{array}{c}\Theta \vdash C_1\ \texttt{loc-ok} \quad \Theta \vdash C_2\ \texttt{loc-ok} \\ \Theta \vdash C_1 \rhd \rho_1 \quad \Theta \vdash C_2 \rhd \rho_2 \\ \mathrm{NL}(\rho_1) \cap \mathrm{SL}(C_2) = \varnothing \\ \mathrm{NL}(\rho_2) \cap \mathrm{SL}(C_1) = \varnothing\end{array}}{\Theta \vdash (C_1, C_2)_\rho\ \texttt{loc-ok}}$$

$$[\text{S-Fst}]\ \frac{\Theta \vdash C\ \texttt{loc-ok} \quad \mathrm{NL}(\rho) \cap \mathrm{SL}(C) = \varnothing}{\Theta \vdash \mathsf{fst}_\rho\ C\ \texttt{loc-ok}} \qquad [\text{S-Snd}]\ \frac{\Theta \vdash C\ \texttt{loc-ok} \quad \mathrm{NL}(\rho) \cap \mathrm{SL}(C) = \varnothing}{\Theta \vdash \mathsf{snd}_\rho\ C\ \texttt{loc-ok}}$$

$$[\text{S-Inl}]\ \frac{\Theta \vdash C\ \texttt{loc-ok}}{\Theta \vdash \mathsf{inl}_\rho\ C\ \texttt{loc-ok}} \qquad [\text{S-Inr}]\ \frac{\Theta \vdash C\ \texttt{loc-ok}}{\Theta \vdash \mathsf{inr}_\rho\ C\ \texttt{loc-ok}}$$

$$[\text{S-Case}]\ \frac{\begin{array}{c}\Theta \vdash C\ \texttt{loc-ok} \quad \Theta, X : \tau_1 \vdash C_1\ \texttt{loc-ok} \quad \Theta, Y : \tau_2 \vdash C_2\ \texttt{loc-ok} \\ \Theta \vdash C \rhd \rho \quad \Theta, X : \tau_1 \vdash C_1 \rhd \rho_1 \quad \Theta, Y : \tau_2 \vdash C_2 \rhd \rho_2 \\ \mathrm{NL}(\rho) \cap (\mathrm{SL}(C) \cup \mathrm{SL}(C_1) \cup \mathrm{SL}(C_2)) = \varnothing \\ \mathrm{NL}(\rho_1) \cap (\mathrm{SL}(C) \cup \mathrm{SL}(C_2)) = \varnothing \\ \mathrm{NL}(\rho_2) \cap (\mathrm{SL}(C) \cup \mathrm{SL}(C_1)) = \varnothing\end{array}}{\Theta \vdash \mathsf{case}_\rho\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inr}\ Y \Rightarrow C_2)\ \texttt{loc-ok}}$$

$$[\text{S-LocalCase}]\ \frac{\begin{array}{c}\Theta \vdash C\ \texttt{loc-ok} \quad \Theta, \rho.x : t_1 \vdash C_1\ \texttt{loc-ok} \quad \Theta, \rho.y : t_2 \vdash C_2\ \texttt{loc-ok} \\ \Theta \vdash C \rhd \rho \quad \Theta, X : \tau_1 \vdash C_1 \rhd \rho_1 \quad \Theta, Y : \tau_2 \vdash C_2 \rhd \rho_2 \\ \mathrm{NL}(\rho) \cap (\mathrm{SL}(C) \cup \mathrm{SL}(C_1) \cup \mathrm{SL}(C_2)) = \varnothing \\ \mathrm{NL}(\rho_1) \cap (\mathrm{SL}(C) \cup \mathrm{SL}(C_2)) = \varnothing \\ \mathrm{NL}(\rho_2) \cap (\mathrm{SL}(C) \cup \mathrm{SL}(C_1)) = \varnothing\end{array}}{\Theta \vdash \mathsf{localCase}_\rho\ C\ \mathsf{of}\ (\mathsf{inl}\ x \Rightarrow C_1)\ (\mathsf{inr}\ y \Rightarrow C_2)\ \texttt{loc-ok}}$$

$$[\text{S-Fold}]\ \frac{\Theta \vdash C\ \texttt{loc-ok}}{\Theta \vdash \mathsf{fold}_\rho\ C\ \texttt{loc-ok}} \qquad [\text{S-Unfold}]\ \frac{\Theta \vdash C\ \texttt{loc-ok} \quad \mathrm{NL}(\rho) \cap \mathrm{SL}(C) = \varnothing}{\Theta \vdash \mathsf{unfold}_\rho\ C\ \texttt{loc-ok}}$$

$$[\text{S-LetLocal}]\ \frac{\begin{array}{c}\Theta \vdash C_1\ \texttt{loc-ok} \quad \Theta, \rho.x : t_e \vdash C_2\ \texttt{loc-ok} \\ \Theta \vdash C_1 \rhd \rho_1 \quad \Theta, \rho.x : t_e \vdash C_2 \rhd \rho_2 \\ \mathrm{NL}(\rho_1) \cap \mathrm{SL}(C_2) = \varnothing \\ \mathrm{NL}(\rho_2) \cap \mathrm{SL}(C_1) = \varnothing \\ \mathrm{NL}(\rho) \cap (\mathrm{SL}(C_1) \cup \mathrm{SL}(C_2)) = \varnothing\end{array}}{\Theta \vdash \mathsf{let}\ \rho.x := C_1\ \mathsf{in}\ C_2\ \texttt{loc-ok}}$$

$$[\text{S-LetLoc}]\ \frac{\begin{array}{c}\Theta \vdash C_1\ \texttt{loc-ok} \qquad \Theta, \alpha :: *_{\text{loc}} \vdash C_2\ \texttt{loc-ok} \\ \Theta \vdash C_1 \rhd \rho_1 \qquad \Theta, \alpha :: *_{\text{loc}} \vdash C_2 \rhd \rho_2 \\ \text{NL}(\rho_1) \cap \text{SL}(C_2) = \varnothing \\ \text{NL}(\rho_2) \cap \text{SL}(C_1) = \varnothing \\ \text{NL}(\rho) \cap (\text{SL}(C_1) \cup \text{SL}(C_2)) = \varnothing\end{array}}{\Theta \vdash \texttt{let}\ \rho.\alpha :: *_{\text{loc}} \coloneqq C_1\ \texttt{in}\ C_2\ \texttt{loc-ok}}$$

$$[\text{S-LetLocSet}]\ \frac{\begin{array}{c}\Theta \vdash C_1\ \texttt{loc-ok} \qquad \Theta, \alpha :: *_{\text{locset}} \vdash C_2\ \texttt{loc-ok} \\ \Theta \vdash C_1 \rhd \rho_1 \qquad \Theta, \alpha :: *_{\text{locset}} \vdash C_2 \rhd \rho_2 \\ \text{NL}(\rho_1) \cap \text{SL}(C_2) = \varnothing \\ \text{NL}(\rho_2) \cap \text{SL}(C_1) = \varnothing \\ \text{NL}(\rho) \cap (\text{SL}(C_1) \cup \text{SL}(C_2)) = \varnothing\end{array}}{\Theta \vdash \texttt{let}\ \rho.\alpha :: *_{\text{locset}} \coloneqq C_1\ \texttt{in}\ C_2\ \texttt{loc-ok}}$$

$$[\text{S-Send}]\ \frac{\Theta \vdash C\ \texttt{loc-ok} \qquad (\text{NL}(\ell) \cup \text{NL}(\rho)) \cap \text{SL}(C) = \varnothing}{\Theta \vdash C\ \{\ell\}\rightsquigarrow \rho\ \texttt{loc-ok}}$$

$$[\text{S-Sync}]\ \frac{\Theta \vdash C\ \texttt{loc-ok} \qquad (\text{NL}(\ell) \cup \text{NL}(\rho)) \cap \text{SL}(C) = \varnothing}{\Theta \vdash \ell[d] \rightsquigarrow \rho\ ;\ C\ \texttt{loc-ok}}$$

$$[\text{S-Fork}]\ \frac{\Theta, \alpha :: *_{\text{loc}}, \{\ell, \alpha\}.x :: \text{loc}_\ell \vdash C\ \texttt{loc-ok} \qquad \text{NL}(\ell) \cap \text{SL}(C) = \varnothing}{\Theta \vdash \texttt{let}\ (\alpha, x) \coloneqq \ell.\texttt{fork}()\ \texttt{in}\ C\ \texttt{loc-ok}}$$

$$[\text{S-Kill}]\ \frac{\Theta \vdash C\ \texttt{loc-ok} \qquad L \notin \text{SL}(C)}{\Theta \vdash \texttt{kill}\ L\ \texttt{after}\ C\ \texttt{loc-ok}}$$

C Network Language

C.1 Network Language Expressions

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

C.2 Transition Labels and Evaluation Contexts

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

C.3 Network Language Operational Semantics

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

$$[\text{N-App}]\ \frac{f = \mathsf{fun}\ F(X) \coloneqq E \qquad \mathrm{Val}(V)}{L \triangleright f\ V \xrightarrow{\iota} E[F \mapsto f, X \mapsto V]} \qquad [\text{N-TApp}]\ \frac{f = \mathsf{tfun}\ F(\alpha) \coloneqq E}{L \triangleright f\ t \xrightarrow{\iota} E[F \mapsto f, \alpha \mapsto t]}$$

$$[\text{N-UnfoldFold}]\ \frac{\mathrm{Val}(V)}{L \triangleright \mathsf{unfold}\ (\mathsf{fold}\ V) \xrightarrow{\iota} V} \qquad [\text{N-FstPair}]\ \frac{\mathrm{Val}(V_1) \qquad \mathrm{Val}(V_2)}{L \triangleright \mathsf{fst}\ (V_1, V_2) \xrightarrow{\iota} V_1}$$

$$[\text{N-SndPair}]\ \frac{\mathrm{Val}(V_1) \qquad \mathrm{Val}(V_2)}{L \triangleright \mathsf{snd}\ (V_1, V_2) \xrightarrow{\iota} V_2}$$

$$[\text{N-CaseInl}]\ \frac{\mathrm{Val}(V)}{L \triangleright \mathsf{case}\ (\mathsf{inl}\ V)\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow E_1)\ (\mathsf{inr}\ Y \Rightarrow E_2) \xrightarrow{\iota} E_1[X \mapsto V]}$$

$$[\text{N-CaseInr}]\ \frac{\mathrm{Val}(V)}{L \triangleright \mathsf{case}\ (\mathsf{inr}\ V)\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow E_1)\ (\mathsf{inr}\ Y \Rightarrow E_2) \xrightarrow{\iota} E_2[Y \mapsto V]}$$

$$[\text{N-LocalCaseInl}]\ \frac{\mathrm{Val}(v)}{L \triangleright \mathsf{localCase}\ \mathsf{ret}(\mathsf{inl}\ v)\ \mathsf{of}\ (\mathsf{inl}\ x \Rightarrow E_1)\ (\mathsf{inr}\ y \Rightarrow E_2) \xrightarrow{\iota} E_1[x \mapsto v]}$$

$$[\text{N-LocalCaseInr}]\ \frac{\mathrm{Val}(v)}{L \triangleright \mathsf{localCase}\ \mathsf{ret}(\mathsf{inr}\ v)\ \mathsf{of}\ (\mathsf{inl}\ x \Rightarrow E_1)\ (\mathsf{inr}\ y \Rightarrow E_2) \xrightarrow{\iota} E_2[y \mapsto v]}$$

$$[\text{N-Let}]\ \frac{\mathrm{Val}(v)}{L \triangleright \mathsf{let}\ x \coloneqq \mathsf{ret}(v)\ \mathsf{in}\ C \xrightarrow{\iota} C[x \mapsto v]}$$

$$[\text{N-TyLet}]\ \frac{\mathrm{Val}(\lceil t \rfloor)}{L \triangleright \mathsf{let}\ \alpha :: \kappa \coloneqq \mathsf{ret}(\lceil t \rfloor)\ \mathsf{in}\ E \xrightarrow{\iota} E[\alpha \mapsto t]}$$

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

$$[\text{N-Choose}]\ \frac{\mathrm{fv}(\rho) = \varnothing}{L \triangleright \mathsf{choose}\ d\ \mathsf{for}\ \rho\ ;\ E \xrightarrow{d \rightsquigarrow \rho \setminus \{L\}} E}$$

$$[\text{N-AllowL}]\ \frac{L' \neq L}{L \triangleright \mathbf{allow}\ L'\ \mathbf{choice}\ (\mathbf{L} \Rightarrow E_1)\ (\mathbf{R} \Rightarrow E_{2\perp}) \xrightarrow{L'.\mathbf{L} \rightsquigarrow} E_1}$$

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

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

$$\text{[N-Fork]}\ \frac{\begin{array}{cc} E_1' = E_1\left[\alpha \mapsto L', x \mapsto \lceil L' \rfloor\right] & L' \text{ globally fresh} \\ E_2' = E_2\left[\alpha \mapsto L', x \mapsto \lceil L' \rfloor\right] & \text{fv}(E_1') = \varnothing \end{array}}{L \triangleright \texttt{let}\ (\alpha, x) := \texttt{fork}(E_1)\ \texttt{in}\ E_2 \xrightarrow{\texttt{fork}(L', E_1')} E_2'} \qquad \text{[N-Exit]}\ \frac{}{L \triangleright \texttt{exit} \xrightarrow{\texttt{exit}} ()}$$

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.

$$\text{undefined} \sqcup \text{undefined} \triangleq \text{undefined}$$

$$\text{undefined} \sqcup E_2 \triangleq E_2$$

$$E_1 \sqcup \text{undefined} \triangleq E_1$$

$$X \sqcup X \triangleq X$$

$$() \sqcup () \triangleq ()$$

$$\texttt{ret}(e) \sqcup \texttt{ret}(e) \triangleq \texttt{ret}(e)$$

$$(E_{1,1}\ ;\ E_{1,2}) \sqcup (E_{2,1}\ ;\ E_{2,2}) \triangleq E_{1,1} \sqcup E_{2,1}\ ;\ E_{1,2} \sqcup E_{2,2}$$

$$(\texttt{fun}\ F(X) := E_1) \sqcup (\texttt{fun}\ F(X) := E_2) \triangleq \texttt{fun}\ F(X) := (E_1 \sqcup E_2)$$

$$(E_{1,1}\ E_{1,2}) \sqcup (E_{2,1}\ E_{2,2}) \triangleq (E_{1,1} \sqcup E_{2,1})\ (E_{1,2} \sqcup E_{2,2})$$

$$(\texttt{tfun}\ F(\alpha) := E_1) \sqcup (\texttt{tfun}\ F(\alpha) := E_2) \triangleq \texttt{tfun}\ F(\alpha) := (E_1 \sqcup E_2)$$

$$(E_1\ t) \sqcup (E_2\ t) \triangleq (E_1 \sqcup E_2)\ t$$

$$(\texttt{fold}\ E_1) \sqcup (\texttt{fold}\ E_2) \triangleq \texttt{fold}\ (E_1 \sqcup E_2)$$

$$(\texttt{unfold}\ E_1) \sqcup (\texttt{unfold}\ E_2) \triangleq \texttt{unfold}\ (E_1 \sqcup E_2)$$

$$(E_{1,1}, E_{1,2}) \sqcup (E_{2,1}, E_{2,2}) \triangleq ((E_{1,1} \sqcup E_{2,1}), (E_{1,2} \sqcup E_{2,2}))$$

$$(\texttt{fst}\ E_1) \sqcup (\texttt{fst}\ E_2) \triangleq \texttt{fst}\ (E_1 \sqcup E_2)$$

$$(\texttt{snd}\ E_1) \sqcup (\texttt{snd}\ E_2) \triangleq \texttt{snd}\ (E_1 \sqcup E_2)$$

$$(\texttt{inl}\ E_1) \sqcup (\texttt{inl}\ E_2) \triangleq \texttt{inl}\ (E_1 \sqcup E_2)$$

$$(\texttt{inr}\ E_1) \sqcup (\texttt{inr}\ E_2) \triangleq \texttt{inr}\ (E_1 \sqcup E_2)$$

$$\left(\begin{array}{l}\texttt{case}\ E_{1,1}\ \texttt{of} \\ \mid \texttt{inl}\ X \Rightarrow E_{1,2} \\ \mid \texttt{inr}\ Y \Rightarrow E_{1,3}\end{array}\right) \sqcup \left(\begin{array}{l}\texttt{case}\ E_{2,1}\ \texttt{of} \\ \mid \texttt{inl}\ X \Rightarrow E_{2,2} \\ \mid \texttt{inr}\ Y \Rightarrow E_{2,3}\end{array}\right) \triangleq \begin{array}{l}\texttt{case}\ (E_{1,1} \sqcup E_{2,1})\ \texttt{of} \\ \mid \texttt{inl}\ X \Rightarrow E_{1,2} \sqcup E_{2,2} \\ \mid \texttt{inr}\ Y \Rightarrow E_{1,3} \sqcup E_{2,3}\end{array}$$

$$\left(\begin{array}{l}\texttt{localCase}\ E_{1,1}\ \texttt{of}\\ \mid \texttt{inl}\ x \Rightarrow E_{1,2}\\ \mid \texttt{inr}\ y \Rightarrow E_{1,3}\end{array}\right) \sqcup \left(\begin{array}{l}\texttt{localCase}\ E_{2,1}\ \texttt{of}\\ \mid \texttt{inl}\ x \Rightarrow E_{2,2}\\ \mid \texttt{inr}\ y \Rightarrow E_{2,3}\end{array}\right) \triangleq \begin{array}{l}\texttt{localCase}\ (E_{1,1} \sqcup E_{2,1})\ \texttt{of}\\ \mid \texttt{inl}\ x \Rightarrow E_{1,2} \sqcup E_{2,2}\\ \mid \texttt{inr}\ y \Rightarrow E_{1,3} \sqcup E_{2,3}\end{array}$$

$$(\texttt{let}\ x \coloneqq E_{1,1}\ \texttt{in}\ E_{1,2}) \sqcup (\texttt{let}\ x \coloneqq E_{2,1}\ \texttt{in}\ E_{2,2}) \triangleq \texttt{let}\ x \coloneqq (E_{1,1} \sqcup E_{2,1})\ \texttt{in}\ (E_{1,2} \sqcup E_{2,2})$$

$$(\texttt{let}\ \alpha :: \kappa \coloneqq E_{1,1}\ \texttt{in}\ E_{1,2}) \sqcup (\texttt{let}\ \alpha :: \kappa \coloneqq E_{2,1}\ \texttt{in}\ E_{2,2}) \triangleq \texttt{let}\ \alpha :: \kappa \coloneqq (E_{1,1} \sqcup E_{2,1})\ \texttt{in}\ (E_{1,2} \sqcup E_{2,2})$$

$$(\texttt{send}\ E_1\ \texttt{to}\ \rho) \sqcup (\texttt{send}\ E_2\ \texttt{to}\ \rho) \triangleq \texttt{send}\ (E_1 \sqcup E_2)\ \texttt{to}\ \rho$$

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

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

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

$$\left(\begin{array}{r}\texttt{AmI}\in \rho\ \texttt{then}\ E_{1,1}\\ \texttt{else}\ E_{1,2}\end{array}\right) \sqcup \left(\begin{array}{r}\texttt{AmI}\in \rho\ \texttt{then}\ E_{2,1}\\ \texttt{else}\ E_{2,2}\end{array}\right) \triangleq \begin{array}{r}\texttt{AmI}\in \rho\ \texttt{then}\ (E_{1,1} \sqcup E_{2,1})\\ \texttt{else}\ (E_{1,2} \sqcup E_{2,2})\end{array}$$

$$\left(\begin{array}{l}\texttt{let}\ (\alpha, x) \coloneqq \texttt{fork}(E_{1,1})\\ \texttt{in}\ E_{1,2}\end{array}\right) \sqcup \left(\begin{array}{l}\texttt{let}\ (\alpha, x) \coloneqq \texttt{fork}(E_{2,1})\\ \texttt{in}\ E_{2,2}\end{array}\right) \triangleq \begin{array}{l}\texttt{let}\ (\alpha, x) \coloneqq \texttt{fork}(E_{1,1} \sqcup E_{2,1})\\ \texttt{in}\ (E_{1,2} \sqcup E_{2,2})\end{array}$$

$$\texttt{exit} \sqcup \texttt{exit} \triangleq \texttt{exit}$$

D.2 Endpoint Projection

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

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

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

$$[\![\texttt{fun}_\rho\ F(X) \coloneqq C]\!]_L = \begin{cases}\texttt{fun}\ F(X) \coloneqq [\![C]\!]_L & \text{if } L \in \rho\\ () & \text{if } L \notin \rho \text{ and } [\![C]\!]_L \neq \text{undefined}\\ \text{undefined} & \text{otherwise}\end{cases}$$

$$[\![C_1\ \$_\rho\ C_2]\!]_L = \begin{cases}[\![C_1]\!]_L\ [\![C_2]\!]_L & \text{if } L \in \rho\\ [\![C_1]\!]_L\ ;\!;\ [\![C_2]\!]_L\ ;\!;\ () & \text{otherwise}\end{cases}$$

$$[\![\texttt{tfun}_\rho\ F(\alpha :: *_{\text{loc}}) \coloneqq C]\!]_L = \begin{cases}\begin{array}{l}\texttt{tfun}\ F(\alpha) \coloneqq\\ \texttt{AmI}\ \alpha\ \texttt{then}\ [\![C[\alpha \mapsto L]]\!]_L\\ \qquad \texttt{else}\ [\![C]\!]_L\end{array} & \text{if } L \in \rho\\ () & \begin{array}{l}\text{if } L \notin \rho \text{ and}\\ [\![C]\!]_L, [\![C[\alpha \mapsto L]]\!]_L \neq \text{undefined}\end{array}\\ \text{undefined} & \text{otherwise}\end{cases}$$

$$\llbracket \mathsf{tfun}_\rho\ F(\alpha :: *_{\mathsf{locset}}) \coloneqq C \rrbracket_L = \begin{cases} \mathtt{tfun}\ F(\alpha) \coloneqq & \\ \quad \mathtt{AmI} \in \alpha\ \mathtt{then}\ \llbracket C[\alpha \mapsto \{L\} \cup \alpha] \rrbracket_L & \text{if } L \in \rho \\ \quad \mathtt{else}\ \llbracket C \rrbracket_L & \\ () & \text{if } L \notin \rho \text{ and} \\ & \llbracket C \rrbracket_L, \llbracket C[\alpha \mapsto \{L\} \cup \alpha] \rrbracket_L \neq \text{undefined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{tfun}_\rho\ F(\alpha :: \kappa) \coloneqq C \rrbracket_L = \begin{cases} \mathtt{tfun}\ F(\alpha) \coloneqq \llbracket C \rrbracket_L & \text{if } L \in \rho \\ () & \text{if } L \notin \rho \text{ and } \llbracket C \rrbracket_L \neq \text{undefined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\llbracket C\ \$_\rho\ t \rrbracket_L = \begin{cases} \llbracket C \rrbracket_L\ t & \text{if } L \in \rho \\ \llbracket C \rrbracket_L \fatsemi () & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{fold}_\rho\ C \rrbracket_L = \begin{cases} \mathtt{fold}\ \llbracket C \rrbracket_L & \text{if } L \in \rho \\ \llbracket C \rrbracket_L & \text{otherwise} \end{cases}$$

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

$$\llbracket (C_1, C_2)_\rho \rrbracket_L = \begin{cases} (\llbracket C_1 \rrbracket_L, \llbracket C_2 \rrbracket_L) & \text{if } L \in \rho \\ \llbracket C_1 \rrbracket_L \fatsemi \llbracket C_2 \rrbracket_L & \text{otherwise} \end{cases}$$

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

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

$$\llbracket \mathsf{inl}_\rho\ C \rrbracket_L = \begin{cases} \mathtt{inl}\ \llbracket C \rrbracket_L & \text{if } L \in \rho \\ \llbracket C \rrbracket_L & \text{otherwise} \end{cases}$$

$$\llbracket \mathsf{inr}_\rho\ C \rrbracket_L = \begin{cases} \mathtt{inr}\ \llbracket C \rrbracket_L & \text{if } L \in \rho \\ \llbracket C \rrbracket_L & \text{otherwise} \end{cases}$$

$$\left\llbracket \begin{array}{l} \mathsf{case}_\rho\ C\ \mathtt{of} \\ \mid \mathtt{inl}\ X \Rightarrow C_1 \\ \mid \mathtt{inr}\ Y \Rightarrow C_2 \end{array} \right\rrbracket_L = \begin{cases} \mathtt{case}\ \llbracket C \rrbracket_L\ \mathtt{of} & \\ \quad \mid \mathtt{inl}\ X \Rightarrow \llbracket C_1 \rrbracket_L & \text{if } L \in \rho \\ \quad \mid \mathtt{inr}\ Y \Rightarrow \llbracket C_2 \rrbracket_L & \\ \llbracket C \rrbracket_L \fatsemi \llbracket C_1 \rrbracket_L \sqcup \llbracket C_2 \rrbracket_L & \text{if } L \notin \rho \text{ and } X \notin \text{fv}(\llbracket C_1 \rrbracket_L) \text{ and } Y \notin \text{fv}(\llbracket C_2 \rrbracket_L) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\left\llbracket \begin{array}{l} \mathsf{localCase}_\rho\ C\ \mathtt{of} \\ \mid \mathtt{inl}\ x \Rightarrow C_1 \\ \mid \mathtt{inr}\ y \Rightarrow C_2 \end{array} \right\rrbracket_L = \begin{cases} \mathtt{localCase}\ \llbracket C \rrbracket_L\ \mathtt{of} & \\ \quad \mid \mathtt{inl}\ x \Rightarrow \llbracket C_1 \rrbracket_L & \text{if } L \in \rho \\ \quad \mid \mathtt{inr}\ y \Rightarrow \llbracket C_2 \rrbracket_L & \\ \llbracket C \rrbracket_L \fatsemi \llbracket C_1 \rrbracket_L \sqcup \llbracket C_2 \rrbracket_L & \text{if } L \notin \rho \text{ and } x \notin \text{fv}(\llbracket C_1 \rrbracket_L) \text{ and } y \notin \text{fv}(\llbracket C_2 \rrbracket_L) \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

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

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

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

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

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

$$[\![\texttt{let}\ (\alpha, x) \coloneqq \ell.\texttt{fork}()\ \texttt{in}\ C]\!]_L = \begin{cases} \texttt{let}\ (\alpha, x) \coloneqq \texttt{fork}([\![C]\!]_\alpha)\ \texttt{in}\ [\![C]\!]_L & \text{if } L = \ell \\ [\![C]\!]_L & \text{if } L \neq \ell \text{ and } \alpha, x \notin \mathrm{fv}([\![C]\!]_L) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$[\![\texttt{kill}\ L'\ \texttt{after}\ C]\!]_L = \begin{cases} [\![C]\!]_L \fatsemi \texttt{exit} & \text{if } L = L' \\ [\![C]\!]_L & \text{otherwise} \end{cases}$$

D.3 Locations Named by a Type or Choreography

$$\mathrm{NL}(\alpha) = \varnothing$$

$$\mathrm{NL}(L) = \{L\}$$

$$\mathrm{NL}(\{\ell\}) = \mathrm{NL}(\ell)$$

$$\mathrm{NL}(\rho_1 \cup \rho_2) = \mathrm{NL}(\rho_1) \cup \mathrm{NL}(\rho_2)$$

$$\mathrm{NL}(\top) = \varnothing$$

$$\mathrm{NL}(t_e@\rho) = \mathrm{NL}(\rho)$$

$$\mathrm{NL}(\tau_1 \xrightarrow{\rho} \tau_2) = \mathrm{NL}(\tau_1) \cup \mathrm{NL}(\tau_2) \cup \mathrm{NL}(\rho)$$

$$\begin{aligned}
\mathrm{NL}(\tau_1 +_\rho \tau_2) &= \mathrm{NL}(\tau_1) \cup \mathrm{NL}(\tau_2) \cup \mathrm{NL}(\rho) \\
\mathrm{NL}(\tau_1 \times \tau_2) &= \mathrm{NL}(\tau_1) \cup \mathrm{NL}(\tau_2) \\
\mathrm{NL}(\mu_\rho \alpha.\, \tau) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(\tau) \\
\mathrm{NL}(\forall \alpha :: \kappa[\rho].\, \tau) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(\tau) \\
\mathrm{NL}(X) &= \varnothing \\
\mathrm{NL}(\rho.e) &= \mathrm{NL}(\rho) \\
\mathrm{NL}(\mathsf{fun}_\rho\, F(X) \coloneqq C) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C) \\
\mathrm{NL}(C_1\ \$_\rho\ C_2) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C_1) \cup \mathrm{NL}(C_2) \\
\mathrm{NL}(\mathsf{tfun}_\rho\, F(\alpha) \coloneqq C) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C) \\
\mathrm{NL}(C\ \$_\rho\ \ell) &= \mathrm{NL}(C) \cup \mathrm{NL}(\rho) \cup \mathrm{NL}(\ell) \\
\mathrm{NL}(C\ \$_\rho\ \rho') &= \mathrm{NL}(C) \cup \mathrm{NL}(\rho) \cup \mathrm{NL}(\rho') \\
\mathrm{NL}(C\ \$_\rho\ t) &= \mathrm{NL}(C) \cup \mathrm{NL}(\rho) \\
\mathrm{NL}(\mathsf{fold}_\rho\ C) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C) \\
\mathrm{NL}(\mathsf{unfold}_\rho\ C) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C) \\
\mathrm{NL}((C_1, C_2)_\rho) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C_1) \cup \mathrm{NL}(C_2) \\
\mathrm{NL}(\mathsf{fst}_\rho\ C) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C) \\
\mathrm{NL}(\mathsf{snd}_\rho\ C) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C) \\
\mathrm{NL}(\mathsf{inl}_\rho\ C) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C) \\
\mathrm{NL}(\mathsf{inr}_\rho\ C) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C) \\
\mathrm{NL}\left(\begin{array}{l}\mathsf{case}_\rho\ C\ \mathsf{of} \\ \mid \mathsf{inl}\ X \Rightarrow C_1 \\ \mid \mathsf{inr}\ Y \Rightarrow C_2\end{array}\right) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C) \cup \mathrm{NL}(C_1) \cup \mathrm{NL}(C_2) \\
\mathrm{NL}\left(\begin{array}{l}\mathsf{localCase}_\rho\ C\ \mathsf{of} \\ \mid \mathsf{inl}\ x \Rightarrow C_1 \\ \mid \mathsf{inr}\ y \Rightarrow C_2\end{array}\right) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C) \cup \mathrm{NL}(C_1) \cup \mathrm{NL}(C_2) \\
\mathrm{NL}(\mathsf{let}\ \rho.x \coloneqq C_1\ \mathsf{in}\ C_2) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C_1) \cup \mathrm{NL}(C_2) \\
\mathrm{NL}(\mathsf{let}\ \rho.\alpha \coloneqq C_1\ \mathsf{in}\ C_2) &= \mathrm{NL}(\rho) \cup \mathrm{NL}(C_1) \cup \mathrm{NL}(C_2) \\
\mathrm{NL}(C\ \{\ell\}\!\rightsquigarrow \rho) &= \mathrm{NL}(\ell) \cup \mathrm{NL}(\rho) \cup \mathrm{NL}(C) \\
\mathrm{NL}(\ell[d] \rightsquigarrow \rho\ ;\ C) &= \mathrm{NL}(\ell) \cup \mathrm{NL}(\rho) \cup \mathrm{NL}(C) \\
\mathrm{NL}(\mathsf{let}\ (\alpha, x) \coloneqq \ell.\mathsf{fork}()\ \mathsf{in}\ C) &= \mathrm{NL}(\ell) \cup \mathrm{NL}(C) \\
\mathrm{NL}(\mathsf{kill}\ L\ \mathsf{after}\ C) &= \{L\} \cup \mathrm{NL}(C)
\end{aligned}$$

D.4 Spawned Locations in a Choreography

$$\mathrm{SL}(X) = \varnothing$$

$$\mathrm{SL}(\rho.e) = \varnothing$$

$$\mathrm{SL}(\mathsf{fun}_\rho\, F(X) \coloneqq C) = \mathrm{SL}(C)$$

$$\mathrm{SL}(C_1\ \$_\rho\ C_2) = \mathrm{SL}(C_1) \cup \mathrm{SL}(C_2)$$

$$\mathrm{SL}(\mathsf{tfun}_\rho\, F(\alpha) \coloneqq C) = \mathrm{SL}(C)$$

$$\mathrm{SL}(C\ \$_\rho\ t) = \mathrm{SL}(C)$$

$$\mathrm{SL}(\mathsf{fold}_\rho\ C) = \mathrm{SL}(C)$$

$$\mathrm{SL}(\mathsf{unfold}_\rho\ C) = \mathrm{SL}(C)$$

$$\mathrm{SL}((C_1, C_2)_\rho) = \mathrm{SL}(C_1) \cup \mathrm{SL}(C_2)$$

$$\mathrm{SL}(\mathsf{fst}_\rho\ C) = \mathrm{SL}(C)$$

$$\mathrm{SL}(\mathsf{snd}_\rho\ C) = \mathrm{SL}(C)$$

$$\mathrm{SL}(\mathsf{inl}_\rho\ C) = \mathrm{SL}(C)$$

$$\mathrm{SL}(\mathsf{inr}_\rho\ C) = \mathrm{SL}(C)$$

$$\mathrm{SN}\left(\begin{array}{l}\mathsf{case}_\rho\ C\ \mathsf{of}\\ \mid\ \mathsf{inl}\ X \Rightarrow C_1\\ \mid\ \mathsf{inr}\ Y \Rightarrow C_2\end{array}\right) = \mathrm{SL}(C) \cup \mathrm{SL}(C_1) \cup \mathrm{SL}(C_2)$$

$$\mathrm{SN}\left(\begin{array}{l}\mathsf{localCase}_\rho\ C\ \mathsf{of}\\ \mid\ \mathsf{inl}\ x \Rightarrow C_1\\ \mid\ \mathsf{inr}\ y \Rightarrow C_2\end{array}\right) = \mathrm{SL}(C) \cup \mathrm{SL}(C_1) \cup \mathrm{SL}(C_2)$$

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

$$\mathrm{SL}(\mathsf{let}\ \rho.\alpha \coloneqq C_1\ \mathsf{in}\ C_2) = \mathrm{SL}(C_1) \cup \mathrm{SL}(C_2)$$

$$\mathrm{SL}(C\ \{\ell\}\!\rightsquigarrow \rho) = \mathrm{SL}(C)$$

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

$$\mathrm{SL}(\mathsf{let}\ (\alpha, x) \coloneqq \ell.\mathsf{fork}()\ \mathsf{in}\ C) = \mathrm{SL}(C)$$

$$\mathrm{SL}(\mathsf{kill}\ L\ \mathsf{after}\ C) = \{L\} \cup \mathrm{SL}(C)$$

D.5 The Less-Than Relation

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

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

$$\frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{E_{1,1}\ E_{1,2} \preceq E_{2,1}\ E_{2,2}} \qquad \frac{E_1 \preceq E_2}{\mathsf{tfun}\, F(\alpha) \coloneqq E_1 \preceq \mathsf{tfun}\, F(\alpha) \coloneqq E_2} \qquad \frac{E_1 \preceq E_2}{E_1\ t \preceq E_2\ t}$$

$$\frac{E_1 \preceq E_2}{\mathsf{fold}\ E_1 \preceq \mathsf{fold}\ E_2} \qquad \frac{E_1 \preceq E_2}{\mathsf{unfold}\ E_1 \preceq \mathsf{unfold}\ E_2} \qquad \frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{(E_{1,1}, E_{1,2}) \preceq (E_{2,1}, E_{2,2})}$$

$$\frac{E_1 \preceq E_2}{\texttt{fst}\ E_1 \preceq \texttt{fst}\ E_2} \qquad \frac{E_1 \preceq E_2}{\texttt{snd}\ E_1 \preceq \texttt{snd}\ E_2} \qquad \frac{E_1 \preceq E_2}{\texttt{inl}\ E_1 \preceq \texttt{inl}\ E_2} \qquad \frac{E_1 \preceq E_2}{\texttt{inr}\ E_1 \preceq \texttt{inr}\ E_2}$$

$$\frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2} \qquad E_{1,3} \preceq E_{2,3}}{\begin{array}{l}\texttt{case}\ E_{1,1}\ \texttt{of} \\ \mid \texttt{inl}\ X \Rightarrow E_{1,2} \\ \mid \texttt{inr}\ Y \Rightarrow E_{1,3}\end{array} \preceq \begin{array}{l}\texttt{case}\ E_{2,1}\ \texttt{of} \\ \mid \texttt{inl}\ X \Rightarrow E_{2,2} \\ \mid \texttt{inr}\ Y \Rightarrow E_{2,3}\end{array}} \qquad \frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2} \qquad E_{1,3} \preceq E_{2,3}}{\begin{array}{l}\texttt{localCase}\ E_{1,1}\ \texttt{of} \\ \mid \texttt{inl}\ x \Rightarrow E_{1,2} \\ \mid \texttt{inr}\ y \Rightarrow E_{1,3}\end{array} \preceq \begin{array}{l}\texttt{localCase}\ E_{2,1}\ \texttt{of} \\ \mid \texttt{inl}\ x \Rightarrow E_{2,2} \\ \mid \texttt{inr}\ y \Rightarrow E_{2,3}\end{array}}$$

$$\frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{\texttt{let}\ x := E_{1,1}\ \texttt{in}\ E_{1,2} \preceq \texttt{let}\ x := E_{2,1}\ \texttt{in}\ E_{2,2}}$$

$$\frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{\texttt{let}\ \alpha :: \kappa := E_{1,1}\ \texttt{in}\ E_{1,2} \preceq \texttt{let}\ \alpha :: \kappa := E_{2,1}\ \texttt{in}\ E_{2,2}} \qquad \frac{E_1 \preceq E_2}{\texttt{send}\ E_1\ \texttt{to}\ \rho \preceq \texttt{send}\ E_2\ \texttt{to}\ \rho}$$

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

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

$$\frac{E_{1,1} \preceq E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{\begin{array}{l}\texttt{let}\ (\alpha, x) := \texttt{fork}(E_{1,1}) \\ \texttt{in}\ E_{1,2}\end{array} \preceq \begin{array}{l}\texttt{let}\ (\alpha, x) := \texttt{fork}(E_{2,1}) \\ \texttt{in}\ E_{2,2}\end{array}} \qquad \frac{}{\texttt{exit} \preceq \texttt{exit}}$$

D.6 The Simulating Less-Than Relation

Let the *simulating less-than relation* $\precsim$ be the following subrelation of $\preceq$. This relation is similar to the less-than relation, but differs in the following ways:

(1) it does not contain a rule to allow $E_1 \preceq V\ ;\ E_2$,
(2) in order for two expressions to be related, their heads must be related by $\precsim$, but their bodies must be related by $\preceq$, and
(3) the rules for function applications and pairs differ in how their right-hand arguments must be related depending on whether their left-hand arguments are values.

This relation is so-named because if $E_1 \precsim E_2$, then the next step that E_1 and E_2 make—if any—must be identical (i.e., E_1 and E_2 simulate each other for a single step), whereas this is not the case for $\preceq$.

$$\frac{}{\text{undefined} \precsim E} \qquad \frac{}{X \precsim X} \qquad \frac{}{() \precsim ()} \qquad \frac{}{X \precsim ()} \qquad \frac{}{() \precsim X} \qquad \frac{}{\texttt{ret}(e) \precsim \texttt{ret}(e)}$$

$$\frac{E_{1,1} \precsim E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{E_{1,1}\ ;\ E_{1,2} \precsim E_{2,1}\ ;\ E_{2,2}} \qquad \frac{E_1 \preceq E_2}{\texttt{fun}\ F(X) := E_1 \precsim \texttt{fun}\ F(X) := E_2}$$

$$\frac{\neg\, \text{Val}(E_{1,1}) \qquad E_{1,1} \precsim E_{2,1} \qquad E_{1,2} \preceq E_{2,2}}{E_{1,1}\ E_{1,2} \precsim E_{2,1}\ E_{2,2}} \qquad \frac{\text{Val}(E_{1,1}) \qquad E_{1,1} \precsim E_{2,1} \qquad E_{1,2} \precsim E_{2,2}}{E_{1,1}\ E_{1,2} \precsim E_{2,1}\ E_{2,2}}$$

$$\frac{E_1 \precsim E_2}{\texttt{tfun}\, F(\alpha) := E_1 \precsim \texttt{tfun}\, F(\alpha) := E_2} \qquad \frac{E_1 \precsim E_2}{E_1\ t \precsim E_2\ t} \qquad \frac{E_1 \precsim E_2}{\texttt{fold}\ E_1 \precsim \texttt{fold}\ E_2}$$

$$\frac{E_1 \precsim E_2}{\texttt{unfold}\ E_1 \precsim \texttt{unfold}\ E_2} \qquad \frac{\neg \mathrm{Val}(E_{1,1}) \qquad E_{1,1} \precsim E_{2,1} \qquad E_{1,2} \precsim E_{2,2}}{(E_{1,1}, E_{1,2}) \precsim (E_{2,1}, E_{2,2})}$$

$$\frac{\mathrm{Val}(E_{1,1}) \qquad E_{1,1} \precsim E_{2,1} \qquad E_{1,2} \precsim E_{2,2}}{(E_{1,1}, E_{1,2}) \precsim (E_{2,1}, E_{2,2})} \qquad \frac{E_1 \precsim E_2}{\texttt{fst}\ E_1 \precsim \texttt{fst}\ E_2} \qquad \frac{E_1 \precsim E_2}{\texttt{snd}\ E_1 \precsim \texttt{snd}\ E_2}$$

$$\frac{E_1 \precsim E_2}{\texttt{inl}\ E_1 \precsim \texttt{inl}\ E_2} \qquad \frac{E_1 \precsim E_2}{\texttt{inr}\ E_1 \precsim \texttt{inr}\ E_2}$$

$$\frac{E_{1,1} \precsim E_{2,1} \qquad E_{1,2} \precsim E_{2,2} \qquad E_{1,3} \precsim E_{2,3}}{\begin{array}{l}\texttt{case}\ E_{1,1}\ \texttt{of} \\ \mid \texttt{inl}\ X \Rightarrow E_{1,2} \\ \mid \texttt{inr}\ Y \Rightarrow E_{1,3}\end{array} \precsim \begin{array}{l}\texttt{case}\ E_{2,1}\ \texttt{of} \\ \mid \texttt{inl}\ X \Rightarrow E_{2,2} \\ \mid \texttt{inr}\ Y \Rightarrow E_{2,3}\end{array}}$$

$$\frac{E_{1,1} \precsim E_{2,1} \qquad E_{1,2} \precsim E_{2,2} \qquad E_{1,3} \precsim E_{2,3}}{\begin{array}{l}\texttt{localCase}\ E_{1,1}\ \texttt{of} \\ \mid \texttt{inl}\ x \Rightarrow E_{1,2} \\ \mid \texttt{inr}\ y \Rightarrow E_{1,3}\end{array} \precsim \begin{array}{l}\texttt{localCase}\ E_{2,1}\ \texttt{of} \\ \mid \texttt{inl}\ x \Rightarrow E_{2,2} \\ \mid \texttt{inr}\ y \Rightarrow E_{2,3}\end{array}} \qquad \frac{E_{1,1} \precsim E_{2,1} \qquad E_{1,2} \precsim E_{2,2}}{\texttt{let}\ x := E_{1,1}\ \texttt{in}\ E_{1,2} \precsim \texttt{let}\ x := E_{2,1}\ \texttt{in}\ E_{2,2}}$$

$$\frac{E_{1,1} \precsim E_{2,1} \qquad E_{1,2} \precsim E_{2,2}}{\texttt{let}\ \alpha :: \kappa := E_{1,1}\ \texttt{in}\ E_{1,2} \precsim \texttt{let}\ \alpha :: \kappa := E_{2,1}\ \texttt{in}\ E_{2,2}} \qquad \frac{E_1 \precsim E_2}{\texttt{send}\ E_1\ \texttt{to}\ \rho \precsim \texttt{send}\ E_2\ \texttt{to}\ \rho}$$

$$\frac{}{\texttt{recv from}\ \ell \precsim \texttt{recv from}\ \ell} \qquad \frac{E_1 \precsim E_2}{\texttt{choose}\ d\ \texttt{for}\ \ell\ ;\ E_1 \precsim \texttt{choose}\ d\ \texttt{for}\ \ell\ ;\ E_2}$$

$$\frac{E_{1,1} \precsim E_{2,1} \qquad E_{1,2} \precsim E_{2,2}}{\begin{array}{l}\texttt{allow}\ \ell\ \texttt{choice} \\ \mid \mathbf{L} \Rightarrow E_{1,1} \\ \mid \mathbf{R} \Rightarrow E_{1,2}\end{array} \precsim \begin{array}{l}\texttt{allow}\ \ell\ \texttt{choice} \\ \mid \mathbf{L} \Rightarrow E_{2,1} \\ \mid \mathbf{R} \Rightarrow E_{2,2}\end{array}} \qquad \frac{E_{1,1} \precsim E_{2,1} \qquad E_{1,2} \precsim E_{2,2}}{\begin{array}{r}\texttt{AmI}\in \rho\ \texttt{then}\ E_{1,1} \\ \texttt{else}\ E_{1,2}\end{array} \precsim \begin{array}{r}\texttt{AmI}\in \rho\ \texttt{then}\ E_{2,1} \\ \texttt{else}\ E_{2,2}\end{array}}$$

$$\frac{E_{1,1} \precsim E_{2,1} \qquad E_{1,2} \precsim E_{2,2}}{\begin{array}{l}\texttt{let}\ (\alpha, x) := \texttt{fork}(E_{1,1}) \\ \texttt{in}\ E_{1,2}\end{array} \precsim \begin{array}{l}\texttt{let}\ (\alpha, x) := \texttt{fork}(E_{2,1}) \\ \texttt{in}\ E_{2,2}\end{array}} \qquad \frac{}{\texttt{exit} \precsim \texttt{exit}}$$

E Proofs

E.1 Substitution Lemmas

The following lemmas quantify the behavior of the kinding and type systems with respect to the substitution operations. Each lemma is proven with respect to an infinite parallel substitution σ mapping all variables to choreographies (or types, or local expressions), of which a single-variable substitution $[X \mapsto C]$ can be recovered as a special case by setting $\sigma(X) = C$ and $\sigma(Y) = Y$ for $Y \neq X$. We make use of these lemmas frequently, and so may elide explicitly referencing them in any following proofs.

Lemma 1 (Location Substitution Preserves Containment). *If $\ell \in \rho$ then $\ell[\sigma] \in \rho[\sigma]$.*

Proof. By induction on ρ. □

Lemma 2 (Location Substitution Preserves Subsets). *If $\rho_1 \subseteq \rho_2$ then $\rho_1[\sigma] \subseteq \rho_2[\sigma]$.*

Proof. By induction on the definition of the $\subseteq$ relation. The only interesting case is when $\rho_1 = \{\ell\}$, which follows by Lemma 1. □

Lemma 3. *If σ is a location substitution where $\forall \alpha. \mathrm{NL}(\sigma(\alpha)) \subseteq \rho$, then $\mathrm{NL}(t) \subseteq \mathrm{NL}(t[\sigma]) \subseteq \mathrm{NL}(t) \cup \rho$.*

Proof. By induction on t. □

Lemma 4. *If σ is a type substitution, then $\mathrm{NL}(t[\sigma]) = \mathrm{NL}(t)$.*

Proof. By induction on t. □

Lemma 5. *For any location substitution σ, $\mathrm{SL}(C[\sigma]) = \mathrm{SL}(C)$.*

Proof. By induction on C. □

Lemma 6. *For any type substitution σ, $\mathrm{SL}(C[\sigma]) = \mathrm{SL}(C)$.*

Proof. By induction on C. □

Lemma 7. *For any local substitution σ, $\mathrm{SL}(C[\rho|\sigma]) = \mathrm{SL}(C)$.*

Proof. By induction on C. □

Lemma 8. *If σ is a substitution where $\forall X. \mathrm{SL}(\sigma(X)) = \varnothing$, then $\mathrm{SL}(C[\sigma]) = \mathrm{SL}(C)$.*

Proof. By induction on C. □

Definition 1 (Well-formed Location-Type Substitutions). Say that a function σ from location-type variables to locations or location sets maps $\Gamma_{\ell,1}$ to $\Gamma_{\ell,2}$ (written $\vdash \sigma : \Gamma_{\ell,1} \Rightarrow \Gamma_{\ell,2}$) if and only if

$$\forall \alpha :: \kappa_\ell \in \Gamma_{\ell,1}. \Gamma_{\ell,2} \vdash \sigma(\alpha) :: \kappa_\ell.$$

Lemma 9 (Location Substitution Preserves Location Kinding). *If $\vdash \sigma : \Gamma_{\ell,1} \Rightarrow \Gamma_{\ell,2}$ and $\Gamma_{\ell,1} \vdash t :: \kappa_\ell$, then $\Gamma_{\ell,2} \vdash t[\sigma] :: \kappa_\ell$.*

Proof. By induction on the kinding derivation $\Gamma_{\ell,1} \vdash t :: \kappa_\ell$. □

Lemma 10 (Location Substitution Preserves Kinding). *If $\vdash \sigma : \Gamma_{\ell,1} \Rightarrow \Gamma_{\ell,2}$ and $\Gamma_{\ell,1}; \Gamma \vdash t :: \kappa$, then $\Gamma_{\ell,2}; \Gamma[\sigma] \vdash t[\sigma] :: \kappa[\sigma]$.*

Proof. By induction on the kinding derivation $\Gamma_{\ell,1}; \Gamma \vdash t :: \kappa$. □

Definition 2. For a location substitution σ and a set of locations Θ, say that σ does not mention Θ (written $\Theta \notin \sigma$) if and only if $L \neq \sigma(\alpha)$ and $L \notin \sigma(\alpha)$ for all location-type variables α and $L \in \Theta$.

Lemma 11 (Unmentioned Substitutions Preserve Equality). *If $\Theta \notin \sigma$ then $\ell = L \Leftrightarrow \ell[\sigma] = L$ for all $L \in \Theta$.*

Lemma 12 (Unmentioned Substitutions Preserve Containment). *If $\Theta \notin \sigma$ then $L \in \rho \Leftrightarrow L \in \rho[\sigma]$ for all $L \in \Theta$.*

Lemma 13 (Unmentioned Substitutions Preserve Containment in Named Locations). *If $\Theta \notin \sigma$ then $L \in \mathrm{NL}(\rho) \Leftrightarrow L \in \mathrm{NL}(\rho[\sigma])$ for all $L \in \Theta$.*

Lemma 14 (Unmentioned Substitutions Preserve Disjointness). *If $\Theta \notin \sigma$, then $\Theta \cap \rho = \varnothing$ if and only if $\Theta \cap \rho[\sigma] = \varnothing$.*

Lemma 15 (Unmentioned Substitutions Preserve Disjointness in Named Locations). *If* $\Theta \notin \sigma$, *then* $\Theta \cap \mathrm{NL}(\rho) = \varnothing$ *if and only if* $\Theta \cap \mathrm{NL}(\rho[\sigma]) = \varnothing$.

Lemma 16 (Context Projection and Location Substitution Commute). *If* σ *is a location substitution,* Δ_e *is a local context, and* ρ *is a location set, then* $(\Delta_e|_\rho)[\sigma] \subseteq \Delta_e[\sigma]|_{\rho[\sigma]}$.

Proof. By induction on Δ_e. If $\Delta_e = \cdot$, the claim is trivial. Otherwise suppose that $\Delta_e = \rho'.x : t_e, \Delta'_e$. If $\rho \subseteq \rho'$, then $\Delta_e|_\rho = x : t_e, \Delta'_e\big|_\rho$, and so

$$(\Delta_e|_\rho)[\sigma] = x : t_e[\sigma], (\Delta'_e\big|_\rho)[\sigma] \subseteq x : t_e[\sigma], \Delta'_e[\sigma]\big|_{\rho[\sigma]}$$

by induction. By Lemma 2, $\rho[\sigma] \subseteq \rho'[\sigma]$, so

$$\Delta_e[\sigma]|_{\rho[\sigma]} = (\rho'[\sigma].x : t_e[\sigma], \Delta'_e[\sigma])\big|_{\rho[\sigma]} = x : t_e[\sigma], \Delta'_e[\sigma]\big|_{\rho[\sigma]}$$

as desired. Otherwise suppose that $\rho \not\subseteq \rho'$. In this case,

$$(\Delta_e|_\rho)[\sigma] = (\Delta'_e\big|_\rho)[\sigma] \subseteq \Delta'_e[\sigma]\big|_{\rho[\sigma]}.$$

We could have either $\rho[\sigma] \subseteq \rho'[\sigma]$—in which case $\Delta_e[\sigma]|_{\rho[\sigma]} = x : t_e[\sigma], \Delta'_e[\sigma]\big|_{\rho[\sigma]}$ as before—or $\rho[\sigma] \not\subseteq \rho'[\sigma]$, wherein $\Delta_e[\sigma]|_{\rho[\sigma]} = \Delta'_e[\sigma]\big|_{\rho[\sigma]}$. In either instance, $(\Delta_e|_\rho)[\sigma] \subseteq \Delta_e[\sigma]|_{\rho[\sigma]}$, completing the proof. □

Lemma 17 (Location Substitution Preserves Typing). *If* $\vdash \sigma : \Gamma_{\ell,1} \Rightarrow \Gamma_{\ell,2}, \Gamma_{\ell,1}; \Gamma; \Delta_e; \Delta \vdash C : \tau \triangleright \rho$, *and* $\mathrm{SL}(C) \notin \sigma$, *then* $\Gamma_{\ell,2}; \Gamma[\sigma]; \Delta_e[\sigma]; \Delta[\sigma] \vdash C[\sigma] : \tau[\sigma] \triangleright \rho[\sigma]$.

Proof. By induction on the typing derivation $\Gamma_{\ell,1}; \Gamma; \Delta_e; \Delta \vdash C : \tau \triangleright \rho$. In the following we denote $\Theta_1 = \Gamma_{\ell,1}; \Gamma; \Delta_e; \Delta$ and $\Theta_2 = \Gamma_{\ell,2}; \Gamma[\sigma]; \Delta_e[\sigma]; \Delta[\sigma]$ for simplicity.

- (T-Var) As $X : \tau[\sigma] \in \Delta[\sigma]$, we have that $\Theta_2 \vdash X : \tau[\sigma] \triangleright \varnothing$ as desired.
- (T-Done) By Lemma 16 we have $(\Delta_e|_\rho)[\sigma] \subseteq \Delta_e[\sigma]|_{\rho[\sigma]}$. Therefore by weakening and location substitution of the local type system $\Gamma_{\ell,2}; \Gamma[\sigma]; \Delta_e[\sigma]|_{\rho[\sigma]} \Vdash e[\sigma] : t[\sigma]$ as desired. As well, because σ is a type substitution, e is a value if and only if $e[\sigma]$ is a value, meaning both $\rho[\sigma].e[\sigma]$ and $\rho.e$ have participant set $\rho[\sigma]$ and ρ, respectively, or both have $\varnothing$.
- (T-Fun) By induction, $\Theta_2, F : \tau_1[\sigma] \xrightarrow{\rho[\sigma]} \tau_2[\sigma], X : \tau_1[\sigma] \vdash C[\sigma] : \tau[\sigma] \triangleright \rho[\sigma]$, so $\Theta_2 \vdash \mathsf{fun}_{\rho'[\sigma]}\, F(X) := C[\sigma] : \tau_1[\sigma] \xrightarrow{\rho[\sigma]} \tau_2[\sigma] \triangleright \varnothing$.
- (T-App) As $\mathrm{SL}(C_1) \cup \mathrm{SL}(C_2) \notin \sigma$, we have that both $\mathrm{SL}(C_1) \notin \sigma$ and $\mathrm{SL}(C_2) \notin \sigma$. Thus by induction, $\Theta_2 \vdash C_1[\sigma] : \tau_1[\sigma] \xrightarrow{\rho[\sigma]} \tau_2[\sigma] \triangleright \rho_1[\sigma]$ and $\Theta_2 \vdash C_2[\sigma] : \tau_1[\sigma] \triangleright \rho_2[\sigma]$. Therefore $\Theta_2 \vdash C_1[\sigma]\ \$_{\rho'[\sigma]}\ C_2[\sigma] : \tau_2[\sigma] \triangleright \rho_1[\sigma] \cup \rho_2[\sigma] \cup \rho'[\sigma]$.
- (T-TFunLoc, T-TFun) Suppose $\Theta_1, F : \forall \alpha :: \kappa_\ell[\rho].\, \tau, \alpha :: \kappa_\ell \vdash C : \tau \triangleright \rho$. Then by induction, $\Theta_2, F : \forall \alpha :: \kappa_\ell[\rho[\sigma]].\, \tau[\sigma], \alpha :: \kappa_\ell \vdash C[\sigma] : \tau[\sigma] \triangleright \rho[\sigma]$. Therefore $\Theta_2 \vdash \mathsf{tfun}_{\rho'}\, F(\alpha :: \kappa_\ell) := C[\sigma] : \forall \alpha :: \kappa_\ell[\rho[\sigma]].\, \tau[\sigma] \triangleright \varnothing$. The case for T-TFun is similar.
- (T-TAppLoc, T-TApp) By induction $\Theta_2 \vdash C[\sigma] : \forall \alpha :: \kappa_\ell[\rho[\sigma]].\, \tau[\sigma] \triangleright \rho_1[\sigma]$, and $\Gamma_{\ell,2} \vdash t[\sigma] :: \kappa_\ell$ by Lemma 9. Therefore $\Theta_2 \vdash C[\sigma]\ \$_{\rho'[\sigma]}\ t[\sigma] : \tau[\alpha \mapsto t][\sigma] \triangleright \rho_1[\sigma] \cup \rho'[\sigma]$ as desired. The case for T-TApp is similar.
- (T-Pair) By induction, $\Theta_2 \vdash C_1[\sigma] : \tau_1[\sigma] \triangleright \rho_1[\sigma]$ and $\Theta_2 \vdash C_2[\sigma] : \tau_2[\sigma] \triangleright \rho_2[\sigma]$. Thus $\Theta_2 \vdash (C_1[\sigma], C_2[\sigma])_{\rho[\sigma]} : \tau_1[\sigma] \times \tau_2[\sigma] \triangleright \rho_1[\sigma] \cup \rho_2[\sigma]$ as desired. The arguments for the other algebraic data type constructors and eliminators are similar.
- (T-LetLoc, T-LetLocSet, T-LetLocal) By induction, $\Theta_2 \vdash C_1[\sigma] : \mathrm{loc}_{\rho_1[\sigma]}@\rho_3[\sigma] \triangleright \rho[\sigma]$ and $\Theta_2, \alpha :: *_{\mathsf{loc}} \vdash C_2[\sigma] : \tau[\sigma] \triangleright \rho'[\sigma]$. By preservation of $\subseteq$ under substitution, $\rho_1[\sigma] \subseteq \rho_2[\sigma] \subseteq \rho_3[\sigma]$. Thus $\Theta_2 \vdash \mathsf{let}\ \rho_2[\sigma].\alpha :: *_{\mathsf{loc}} := C_1[\sigma]\ \mathsf{in}\ C_2[\sigma] : \tau[\sigma] \triangleright \rho[\sigma] \cup (\rho'[\sigma] \setminus \alpha) \cup \rho_2[\sigma]$ as desired. The cases for T-LetLocSet, and T-LetLocal are analogous.

- (T-Send, T-Sync) By induction, $\Theta_2 \vdash C[\sigma] : t_e[\sigma]@\rho_1[\sigma] \rhd \rho[\sigma]$. As containment is preserved under substitution, $\ell_1[\sigma] \in \rho_1[\sigma]$. Therefore $\Theta_2 \vdash C[\sigma]$ $\{\ell[\sigma]\}\rightsquigarrow$ $\rho_2[\sigma] : t_e[\sigma]@(\rho_1[\sigma] \cup \rho_2[\sigma]) \rhd \rho[\sigma] \cup \{\ell[\sigma]\} \cup \rho_2[\sigma]$. The argument for T-Sync is similar.
- (T-Fork) By induction $\Theta_2, \alpha :: *_{\text{loc}}, \{\alpha, \ell[\sigma]\}.x : \text{loc}_\alpha \vdash C[\sigma] : \tau[\sigma] \rhd \rho[\sigma]$, so $\Theta_2 \vdash \text{let } (\alpha, x) := \ell[\sigma].\text{fork}() \text{ in } C[\sigma] : \tau[\sigma] \rhd \rho[\sigma]$ as desired.
- (T-Kill) By induction, $\Theta_2 \vdash C[\sigma] : \tau \rhd \rho$. As $L \notin \sigma$ and $L \notin \text{NL}(\tau)$, using Lemma 13 we have $L \notin \text{NL}(\tau[\sigma])$. Therefore $\Theta_2 \vdash \text{kill } L \text{ after } C[\sigma] : \tau[\sigma] \rhd \rho[\sigma] \cup \{L\}$.

□

Definition 3 (Well-formed Type Substitutions)**.** Say that a function σ from type variables to types maps Γ_1 to Γ_2 under Γ_ℓ (written $\Gamma_\ell \vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2$) if and only if

$$\forall \alpha :: \kappa \in \Gamma_1.\, \Gamma_\ell; \Gamma_2 \vdash \sigma(\alpha) :: \kappa.$$

Lemma 18 (Type Substitution Preserves Kinding)**.** *If* $\Gamma_\ell \vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2$, $\Gamma_\ell; \Gamma_1 \vdash t :: \kappa$, *and* $\Gamma_\ell \vdash \Gamma_2$, *then* $\Gamma_\ell; \Gamma_2 \vdash t[\sigma] :: \kappa[\sigma]$.

Proof. By induction on the kinding derivation $\Gamma_\ell, \Gamma_1 \vdash t :: \kappa$, and using the fact that the local kinding system is preserved under well-formed type substitutions. □

Lemma 19 (Context Projection and Type Substitution Commute)**.** *If* σ *is a type substitution,* Δ_e *is a local context, and* ρ *is a location set, then* $(\Delta_e|_\rho)[\sigma] = \Delta_e[\sigma]|_\rho$.

Proof. The proof is identical to Lemma 16, also noting that type substitution does not affect location sets so that both projected contexts contain the same variables. □

Lemma 20 (Type Substitution Preserves Typing)**.** *If* $\Gamma_\ell \vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2$, $\Gamma_\ell; \Gamma_1; \Delta_e; \Delta \vdash C : \tau \rhd \rho$, *and* $\Gamma_\ell \vdash \Gamma_2$, *then* $\Gamma_\ell; \Gamma_2; \Delta_e[\sigma]; \Delta[\sigma] \vdash C[\sigma] : \tau[\sigma] \rhd \rho$.

Proof. The argument proceeds similarly to Lemma 17, also using the facts that the local type system is preserved under well-formed type substitutions, and that locations and location sets are unaffected by type substitution. □

Definition 4 (Well-formed Local Substitutions)**.** Say that a function σ from local variables to local expressions maps $\Delta_{e,1}$ to $\Delta_{e,2}$ under $\Gamma_\ell; \Gamma$ (written $\Gamma_\ell; \Gamma \vdash \sigma : \Delta_{e,1} \Rightarrow \Delta_{e,2}$) if and only if

$$\forall \rho.x : t_e \in \Delta_{e,1}.\, \Gamma_\ell; \Gamma; \Delta_{e,2}\big|_\rho \Vdash \sigma(x) : t_e.$$

Lemma 21 (Local Substitution Preserves Typing)**.** *If* $\Gamma_\ell; \Gamma \vdash \sigma : \Delta_{e,1} \Rightarrow \Delta_{e,2}$, $\Gamma_\ell; \Gamma; \Delta_{e,1}; \Delta \vdash C : \tau \rhd \rho$, *and* $\Gamma_\ell; \Gamma \vdash \Delta_{e,2}$, *then* $\Gamma_\ell; \Gamma; \Delta_{e,2}; \Delta \vdash C[\sigma] : \tau \rhd \rho$.

Proof. By induction on the typing derivation $\Gamma_\ell; \Gamma; \Delta_{e,1}; \Delta \vdash C : \tau \rhd \rho$, and using the fact that the local type system is preserved under well-formed local substitutions. □

Definition 5 (Well-formed Substitutions)**.** Say that a function σ from program variables to choreographies maps Δ_1 to Δ_2 under $\Gamma_\ell; \Gamma; \Delta_e$ (written $\Gamma_\ell; \Gamma; \Delta_e \vdash \sigma : \Delta_1 \Rightarrow \Delta_2$) if and only if

$$\forall X : \tau \in \Delta_1.\, \Gamma_\ell; \Gamma; \Delta_e; \Delta_2 \vdash \sigma(X) : \tau \rhd \varnothing \wedge \text{SL}(\sigma(X)) = \varnothing.$$

Lemma 22 (Substitution Preserves Typing)**.** *If* $\Gamma_\ell; \Gamma; \Delta_e \vdash \sigma : \Delta_1 \Rightarrow \Delta_2$, $\Gamma_\ell; \Gamma; \Delta_e; \Delta_1 \vdash C : \tau \rhd \rho$, *and* $\Gamma_\ell; \Gamma \vdash \Delta_2$, *then* $\Gamma_\ell; \Gamma; \Delta_e; \Delta_2 \vdash C[\sigma] : \tau \rhd \rho$.

Proof. By induction on the typing derivation $\Gamma_\ell; \Gamma; \Delta_e; \Delta_1 \vdash C : \tau \rhd \rho$. The argument proceeds similarly to Lemma 17, and the only interesting cases are for variables. Indeed, if $X : \tau \in \Delta_1$, then as σ is well-formed, $\Theta_2 \vdash \sigma(X) : \tau \rhd \varnothing$. This suffices because the premise is that $\Theta_1 \vdash X : \tau \rhd \varnothing$. □

Lemma 23 (Participants of Values). *If* $\Theta \vdash V : \tau \triangleright \rho$ *and* $\mathrm{Val}(V)$ *or* $V = X$*, then* $\rho = \varnothing$ *and* $\mathrm{SL}(V) = \varnothing$.

Proof. By induction on the typing derivation $\Theta \vdash V : \tau \triangleright \rho$, noting that no introduction form adds more locations to ρ than are in its subterms. □

Corollary 1. *If* $\Theta, X : \tau_1 \vdash C : \tau_2 \triangleright \rho_2$, $\Theta \vdash V : \tau_1 \triangleright \rho_1$, *and* $\mathrm{Val}(V)$*, then* $\Theta \vdash C[X \mapsto V] : \tau_2 \triangleright \rho_2$.

Lemma 24 (Location Substitution Preserves Spawned Thread Well-Scopedness). *If* $\vdash \sigma : \Gamma_{\ell,1} \Rightarrow \Gamma_{\ell,2}$, $\Gamma_{\ell,1}; \Gamma; \Delta_e; \Delta \vdash C$ loc-ok*, and* $\mathrm{SL}(C) \notin \sigma$ *then* $\Gamma_{\ell,2}; \Gamma[\sigma]; \Delta_e[\sigma]; \Delta[\sigma] \vdash C[\sigma]$ loc-ok.

Proof. By induction on the judgment $\Theta_1 \vdash C$ loc-ok.

- (S-Var) As $X[\sigma] = X$, we trivially we have that $\Theta_2 \vdash X$ loc-ok.
- (S-Fun, S-TFun) We handle the case for functions. By induction $\Theta_2 \vdash C[\sigma]$ loc-ok, and by Lemma 5, $\mathrm{SL}(C[\sigma]) = \mathrm{SL}(C) = \varnothing$, so $\Theta_2 \vdash \mathsf{fun}_{\rho[\sigma]}\, F(X) \coloneqq C[\sigma]$ loc-ok. The argument is identical for type functions.
- (S-App, S-TApp) We handle the case for function application. By induction, $\Theta_2 \vdash C_1[\sigma]$ loc-ok and $\Theta_2 \vdash C_2[\sigma]$ loc-ok. We show that $\mathrm{SL}(C_1[\sigma]) = \mathrm{SL}(C_1)$ and $\mathrm{NL}(\rho_2[\sigma]) \subseteq \mathrm{NL}(\rho_2) \cup \mathrm{NL}(\sigma)$ are disjoint. Indeed, $\mathrm{NL}(\rho_2)$ is already disjoint with $\mathrm{SL}(C_1)$ by assumption, and $\mathrm{NL}(\sigma)$ is disjoint with $\mathrm{SL}(C_1)$ because $\mathrm{SL}(C_1) \notin \sigma$. The same is true for $\mathrm{SL}(C_2[\sigma])$ and $\mathrm{NL}(\rho_1[\sigma])$. Therefore $\Theta_2 \vdash C_1\ \$_\rho\ C_2$ loc-ok. The argument is similar for most other data type introduction and elimination forms.
- (S-Inl, S-Inr, S-Fold, S-Send, S-Sync, S-Fork) We handle the case for inl. By induction, $\Theta_2 \vdash C[\sigma]$ loc-ok. We must show that $\mathrm{NL}(\rho[\sigma])$ and $\mathrm{SL}(C[\sigma]) = \mathrm{SL}(C)$ are disjoint. This follows immediately by using Lemma 15, and the assumptions that $\mathrm{NL}(\rho)$ and $\mathrm{SL}(C)$ are disjoint and that $\mathrm{SL}(C) \notin \sigma$. The arguments for the other cases are similar.
- (T-Kill) The assumptions are that $\Theta_1 \vdash C$ loc-ok, $L \notin \mathrm{SL}(C)$, and $\{L\} \cup \mathrm{SL}(C) \notin \sigma$. Then by induction, $\Theta_2 \vdash C[\sigma]$ loc-ok. As well, $L \notin \mathrm{SL}(C[\sigma]) = \mathrm{SL}(C)$, so $\Theta_2 \vdash$ kill L after $C[\sigma]$ loc-ok as desired.

□

Corollary 2. *If* $\Theta, \alpha :: \kappa_\ell \vdash C$ loc-ok, $\Theta \vdash t :: \kappa_\ell$*, and* $t \notin \mathrm{SL}(C)$*, then* $\Theta \vdash C[\alpha \mapsto t]$ loc-ok.

Lemma 25 (Type Substitution Preserves Spawned Thread Well-Scopedness). *If* $\Gamma_\ell \vdash \sigma : \Gamma_1 \Rightarrow \Gamma_2$ *and* $\Gamma_\ell; \Gamma_1; \Delta_e; \Delta \vdash C$ loc-ok*, then* $\Gamma_\ell; \Gamma_2; \Delta_e[\sigma]; \Delta[\sigma] \vdash C[\sigma]$ loc-ok.

Proof. The proof is straightforward by induction, noting that by Lemmas 6 and 20 the substitution will not change the participants of C nor $\mathrm{SL}(C)$. □

Lemma 26 (Local Substitution Preserves Spawned Thread Well-Scopedness). *If* $\Gamma_\ell; \Gamma \vdash \sigma : \Delta_{e,1} \Rightarrow \Delta_{e,2}$, $\Theta \vdash \Gamma_\ell; \Gamma; \Delta_{e,1}; \Delta$ loc-ok C*, and* $\Gamma_\ell; \Gamma \vdash \Delta_{e,2}$*, then* $\Gamma_\ell; \Gamma; \Delta_{e,2}; \Delta \vdash C[\rho|\sigma]$ loc-ok.

Proof. By induction on the judgment $\Theta \vdash C$ loc-ok. □

Lemma 27. *If* $\mathrm{SL}(C) = \varnothing$ *and* $\Theta \vdash C : \tau \triangleright \rho$ *then* $\Theta \vdash C$ loc-ok.

Proof. By induction, noting that $\mathrm{SL}(C') = \varnothing$ for all subterms C' of C. □

Lemma 28 (Substitution Preserves Spawned Thread Well-Scopedness). *If* $\Gamma_\ell; \Gamma; \Delta_e \vdash \sigma : \Delta_1 \Rightarrow \Delta_2$, $\Gamma_\ell; \Gamma; \Delta_e; \Delta_1 \vdash C$ loc-ok, $\Gamma_\ell; \Gamma \vdash \Delta_2$*, and* $\forall X : \tau_1 \in \Delta_1.\ \Gamma_\ell; \Gamma; \Delta_e; \Delta_2 \vdash \sigma(X)$ loc-ok*, then* $\Gamma_\ell; \Gamma; \Delta_e; \Delta_2 \vdash C[\sigma]$ loc-ok.

Proof. By induction on the judgment $\Theta \vdash C$ loc-ok.

- (S-Var) By assumption, $\Theta_2 \vdash \sigma(X)$ loc-ok.

- (S-Fun, S-TFun) We handle the case for functions. By induction $\Theta_2, F : \tau_1 \xrightarrow{\rho} \tau_2, X : \tau_1 \vdash C[\sigma]$ loc-ok, and by Lemma 8, $\mathrm{SL}(C[\sigma]) = \mathrm{SL}(C) = \varnothing$, so $\Theta_2 \vdash \mathsf{fun}_\rho\, F(X) := C[\sigma]$ loc-ok. The argument is identical for type functions.
- (S-App, S-TApp) We hande the case for function application. By induction, $\Theta_2 \vdash C_1[\sigma]$ loc-ok and $\Theta_2 \vdash C_2[\sigma]$ loc-ok. It follows by the assumptions that $\mathrm{SL}(C_1[\sigma]) = \mathrm{SL}(C_1)$ and $\mathrm{NL}(\rho_2)$ are disjoint. The same is true for $\mathrm{SL}(C_2[\sigma])$ and $\mathrm{NL}(\rho_1)$. Therefore $\Theta_2 \vdash C_1\ \$_\rho\ C_2$ loc-ok. The argument is similar for most other data type introduction and elimination forms.
- (S-Inl, S-Inr, S-Fold, S-Send, S-Sync, S-Fork) We handle the case for inl. By induction, $\Theta \vdash C[\sigma]$ loc-ok. We must show that $\mathrm{NL}(\rho)$ and $\mathrm{SL}(C[\sigma]) = \mathrm{SL}(C)$ are disjoint, which is already given. The arguments for the other cases are similar.
- (T-Kill) The assumptions are that $\Theta_1 \vdash C$ loc-ok, $L \notin \mathrm{SL}(C)$, and $\{L\} \cup \mathrm{SL}(C) \notin \sigma$. Then by induction, $\Theta_2 \vdash C[\sigma]$ loc-ok. As well, $L \notin \mathrm{SL}(C[\sigma]) = \mathrm{SL}(C)$, so $\Theta_2 \vdash$ kill L after $C[\sigma]$ loc-ok as desired.

□

Corollary 3. *If* $\Theta, X : \tau \vdash C$ loc-ok, $\Theta, X : \tau \vdash C : \tau' \rhd \rho'$, $\Theta \vdash V : \tau \rhd \rho$, *and* $\mathrm{Val}(V)$, *then* $\Theta \vdash C[X \mapsto V]$ loc-ok.

E.2 Type Soundness

Lemma 29. *If* $\Theta \vdash C : \tau \rhd \rho$, *then* $\mathrm{SL}(C) \subseteq \mathrm{NL}(\rho) \subseteq \mathrm{NL}(C)$.

Proof. By induction on the typing judgment $\Theta \vdash C : \tau \rhd \rho$. □

Theorem 1 (Sound Participant Sets). *If* $\Theta \vdash C : \tau \rhd \rho$ *and* $\langle C, \Omega\rangle \overset{R}{\Longrightarrow}_c \langle C', \Omega'\rangle$, *then* $\mathrm{rloc}(R) \subseteq \rho \setminus \top$.

Proof. By induction on the step $\langle C, \Theta\rangle \overset{R}{\Longrightarrow}_c \langle C', \Theta'\rangle$.

- (C-Done, C-App, C-TApp, C-UnfoldFold, C-FstPair, C-SndPair, C-CaseInl, C-CaseInr, C-LetV, C-TyLetV, C-SendV, C-Fork, C-Sync) Immediate.
- (C-Ctx, C-SyncI, C-CaseI, C-AppI, C-PairI, C-LetI) By induction.
- (C-TyLetI, C-ForkI) Follows because all locations in $\mathrm{rloc}(R)$ are concrete, so $\alpha \notin \mathrm{rloc}(R)$.

□

Lemma 30. *If* $\Theta \vdash C : \tau \rhd \rho$, *then* $\mathrm{NL}(\rho) \subseteq \mathrm{cloc}(C)$.

Proof. By induction on the typing derivation $\Theta \vdash C : \tau \rhd \rho$. The only interesting case is when $C = \rho.e$, wherein if $\mathrm{Val}(e)$ we have that $\mathrm{NL}(\varnothing) \subseteq \rho$, and otherwise $\mathrm{NL}(\rho) \subseteq \rho$. □

Lemma 31 (Single-Step Type Preservation). *If* $\Theta \vdash C : \tau \rhd \rho$, $\Theta \vdash C$ loc-ok, $\mathrm{NL}(\rho) \subseteq \Omega$, *and* $\langle C, \Omega\rangle \overset{R}{\Longrightarrow}_c \langle C', \Omega'\rangle$, *then there is some* ρ' *such that all of the following properties hold.*

(1) $\Theta \vdash C' : \tau \rhd \rho'$
(2) $\Theta \vdash C'$ loc-ok
(3) $\mathrm{NL}(\rho') \subseteq \Omega'$
(4) $\mathrm{SL}(C') \setminus \mathrm{SL}(C) = \Omega' \setminus \Omega$
(5) $\mathrm{SL}(C) \setminus \mathrm{SL}(C') = \Omega \setminus \Omega'$
(6) $\mathrm{NL}(\rho') \setminus \mathrm{NL}(\rho) \subseteq \Omega' \setminus \Omega$
(7) $\Omega \setminus \Omega' \subseteq \mathrm{NL}(\rho) \setminus \mathrm{NL}(\rho')$

Proof. By induction on the step $\langle C, \Omega\rangle \overset{R}{\Longrightarrow}_c \langle C', \Omega'\rangle$. Most cases are immediate by induction and using the various substitution lemmas.

- (C-Ctx) We handle the case for reductions in pairs and inl. The argument for the other cases, as well as the out-of-order cases, are similar.
  For a pair (C_1, C_2), the assumptions are that $\Theta \vdash C_1 : \tau_1 \triangleright \rho_1$, $\Theta \vdash C_2 : \tau_2 \triangleright \rho_2$, $\mathrm{NL}(\rho_1)$ and $\mathrm{SL}(C_2)$ are disjoint, and $\mathrm{NL}(\rho_2)$ and $\mathrm{SL}(C_1)$ are disjoint. First suppose the reduction is in the left-hand side. By induction, there is some ρ_1' where $\Theta \vdash C_1' : \tau_1 \triangleright \rho_1'$, $\Theta \vdash C_1'$ loc-ok, and conditions (3–7) hold.
  (1) By induction, $\Theta \vdash (C_1', C_2)_\rho : \tau_1 \times \tau_2 \triangleright \rho_1' \cup \rho_2$.
  (2) First, we show that $\mathrm{NL}(\rho_1')$ and $\mathrm{SL}(C_2)$ are disjoint. Suppose that $L \in \mathrm{NL}(\rho_1')$ and $L \in \mathrm{SL}(C_2)$. By the assumption that $\mathrm{NL}(\rho_1)$ and $\mathrm{SL}(C_2)$ are disjoint, we must have that $L \notin \mathrm{NL}(\rho_1)$. Therefore $L \in \mathrm{NL}(\rho_1') \setminus \mathrm{NL}(\rho_1)$, and hence $L \in \Omega' \setminus \Omega$ by the inductive hypothesis. But as $\mathrm{SL}(C_2) \subseteq \mathrm{NL}(\rho_2) \subseteq \Omega$, we have a contradiction, as desired.
  Now we show that $\mathrm{NL}(\rho_2)$ and $\mathrm{SL}(C_1')$ are disjoint. Suppose that $L \in \mathrm{NL}(\rho_2)$ and $L \in \mathrm{SL}(C_1')$. By the assumption that $\mathrm{NL}(\rho_2)$ and $\mathrm{SL}(C_1)$ are disjoint, we must have that $L \notin \mathrm{SL}(C_1)$. Therefore $L \in \mathrm{SL}(C_1') \setminus \mathrm{SL}(C_1)$, and hence $L \in \Omega' \setminus \Omega$ by the inductive hypothesis. But as $\mathrm{NL}(\rho_2) \subseteq \Omega$, we have a contradiction, as desired. This means that $\Theta \vdash (C_1', C_2)_\rho$ loc-ok.
  (3) We show that $\mathrm{NL}(\rho_1') \cup \mathrm{NL}(\rho_2) \subseteq \Omega'$. By induction $\mathrm{NL}(\rho_1') \subseteq \Omega'$, so we need only show that $\mathrm{NL}(\rho_2) \subseteq \Omega'$. To that end, let $L \in \mathrm{NL}(\rho_2)$, and suppose for contradiction that $L \notin \Omega'$. Then $L \in \Omega$ because $\mathrm{NL}(\rho_2) \subseteq \Omega$ by assumption, so $L \in \Omega \setminus \Omega'$. Then by the inductive hypothesis, $L \in \mathrm{SL}(C_1) \setminus \mathrm{SL}(C_1') \subseteq \mathrm{SL}(C_1)$. But by assumption $\mathrm{SL}(C_1)$ and $\mathrm{NL}(\rho_2)$ are disjoint, so we have a contradiction.
  (4) We need to show that $\mathrm{SL}((C_1', C_2)_\rho) \setminus \mathrm{SL}((C_1, C_2)_\rho) = \mathrm{SL}(C_1') \setminus (\mathrm{SL}(C_1) \cup \mathrm{SL}(C_2)) = \Omega' \setminus \Omega$. We can see that $\mathrm{SL}(C_1') \setminus (\mathrm{SL}(C_1) \cup \mathrm{SL}(C_2)) \subseteq \Omega' \setminus \Omega$ easily because $\mathrm{SL}(C_1') \setminus \mathrm{SL}(C_1) \subseteq \Omega' \setminus \Omega$ by the inductive hypothesis. For the other direction, if $L \in \Omega' \setminus \Omega$, then $L \in \mathrm{SL}(C_1') \setminus \mathrm{SL}(C_1)$, so we need only show that $L \notin \mathrm{SL}(C_2)$. This holds because if $L \in \mathrm{SL}(C_2) \subseteq \mathrm{NL}(C_2)$, then we would have that $L \in \Omega$, a contradiction.
  (5) We need to show that $\mathrm{SL}((C_1, C_2)_\rho) \setminus \mathrm{SL}((C_1', C_2)_\rho) = \mathrm{SL}(C_1) \setminus (\mathrm{SL}(C_1') \cup \mathrm{SL}(C_2)) = \Omega \setminus \Omega'$. We can see that $\mathrm{SL}(C_1) \setminus (\mathrm{SL}(C_1') \cup \mathrm{SL}(C_2)) \subseteq \Omega \setminus \Omega'$ easily because $\mathrm{SL}(C_1) \setminus \mathrm{SL}(C_1') \subseteq \Omega \setminus \Omega'$ by the inductive hypothesis. For the other direction, if $L \in \Omega \setminus \Omega'$, then $L \in \mathrm{SL}(C_1) \setminus \mathrm{SL}(C_1')$, so we need only show that $L \notin \mathrm{SL}(C_2)$. This holds because $L \in \mathrm{NL}(C_1) \setminus \mathrm{NL}(\rho_1')$ by (7) of the induction, so $L \in \mathrm{NL}(C_1)$. But then as $\mathrm{NL}(C_1)$ and $\mathrm{SL}(C_2')$ are disjoint, $L \notin \mathrm{SL}(C_2)$ as desired.
  (6) We need to show that $(\mathrm{NL}(\rho_1') \cup \mathrm{NL}(\rho_2)) \setminus (\mathrm{NL}(\rho_1) \cup \mathrm{NL}(\rho_2)) = \mathrm{NL}(\rho_1') \setminus (\mathrm{NL}(\rho_1) \cup \mathrm{NL}(\rho_2)) \subseteq \Omega' \setminus \Omega$. However, $\mathrm{NL}(\rho_1') \setminus \mathrm{NL}(\rho_1) \subseteq \Omega' \setminus \Omega$ by (6) of the inductive hypothesis, which is satisfactory.
  (7) We need to show that $\Omega \setminus \Omega' \subseteq (\mathrm{NL}(\rho_1) \cup \mathrm{NL}(\rho_2)) \setminus (\mathrm{NL}(\rho_1') \cup \mathrm{NL}(\rho_2)) = \mathrm{NL}(\rho_1) \setminus (\mathrm{NL}(\rho_1') \cup \mathrm{NL}(\rho_2))$. To that end, let $L \in \Omega \setminus \Omega'$. Clearly $L \in \mathrm{NL}(\rho_1)$ as $\mathrm{NL}(\rho_1) \subseteq \Omega$, so we must show that $L \notin \mathrm{NL}(\rho_1') \cup \mathrm{NL}(C_2)$. But by (7) of the inductive hypothesis, $L \notin \mathrm{NL}(\rho_1')$, so we must simply show that $L \notin \mathrm{NL}(\rho_2)$. By (5) of the inductive hypothesis, $L \in \mathrm{SL}(\rho_1) \setminus \mathrm{SL}(\rho_1')$, and hence $L \notin \mathrm{NL}(\rho_2)$ because $\mathrm{SL}(\rho_1)$ and $\mathrm{NL}(\rho_2)$ are disjoint.
  The argument for reductions on the right-hand side of the pair is symmetric.
  For $\mathsf{inl}_\rho\ C$, the assumptions are that $\Theta \vdash C_1 : \tau_1 \triangleright \rho_1$, $\mathrm{tloc}(\Theta; \tau_1) \cup \mathrm{tloc}(\Theta; \tau_2) \subseteq \rho$, $\Theta \vdash C_1$ loc-ok, $\mathrm{NL}(\rho)$ and $\mathrm{SL}(C_1)$ are disjoint, and $\mathrm{NL}(\rho_1) \cup \mathrm{NL}(\rho) \subseteq \Omega$. By induction, there is some ρ_1' where $\Theta \vdash C_1' : \tau_1 \triangleright \rho_1'$, $\Theta \vdash C_1'$ loc-ok, and conditions (3–7) hold.
  (1) Clearly $\Theta \vdash \mathsf{inl}_\rho\ C_1' : \tau_1 +_\rho \tau_2 \triangleright \rho_1'$.
  (2) We show that $\mathrm{NL}(\rho)$ and $\mathrm{SL}(C_1')$ are disjoint. Suppose that $L \in \mathrm{SL}(C_1')$ and $L \in \mathrm{NL}(\rho)$. We must have that $L \notin \mathrm{SL}(C_1)$, as $\mathrm{NL}(\rho)$ and $\mathrm{SL}(C_1)$ are disjoint by assumption. But

then $L \in \mathrm{SL}(C_1') \setminus \mathrm{SL}(C_1) = \Omega' \setminus \Omega$, and hence $L \notin \mathrm{NL}(\rho) \subseteq \Omega$, a contradiction as desired. Therefore $\Theta \vdash \mathsf{inl}_\rho\ C_1'\ \texttt{loc-ok}$.

(3) We need to show that $\mathrm{NL}(\rho_1') \subseteq \Omega'$, which is precisely (3) of the inductive hypothesis.

(4) We need to show that $\mathrm{SL}(\mathsf{inl}_\rho\ C_1') \setminus \mathrm{SL}(\mathsf{inl}_\rho\ C_1) = \mathrm{SL}(C_1') \setminus \mathrm{SL}(C_1) = \Omega' \setminus \Omega$, but this is precisely (4) of the inductive hypothesis.

(5) Symmetrically, $\mathrm{SL}(\mathsf{inl}_\rho\ C_1) \setminus \mathrm{SL}(\mathsf{inl}_\rho\ C_1') = \mathrm{SL}(C_1) \setminus \mathrm{SL}(C_1') = \Omega \setminus \Omega'$ by (5) of the inductive hypothesis.

(6) We need to show that $\mathrm{NL}(\rho_1') \setminus \mathrm{NL}(\rho_1) \subseteq \Omega' \setminus \Omega$, which is given directly by the inductive hypothesis.

(7) Finally, we need to show that $\mathrm{NL}(\rho_1) \setminus \mathrm{NL}(\rho_1') \subseteq \Omega \setminus \Omega'$, which is also provided by (7) of the inductive hypothesis.

- (C-Done) Follows by local type preservation, and as no locations are spawned or killed.
- (C-App) The assumptions are that $\Theta, F : \tau_1 \xrightarrow{\rho} \tau_2, X : \tau_1 \vdash C : \tau_2 \rhd \rho$, $\Theta, F : \tau_1 \xrightarrow{\rho} \tau_2, X : \tau_1 \vdash C\ \texttt{loc-ok}$, $\Theta \vdash V : \tau_1 \rhd \varnothing$, $\Theta \vdash V\ \texttt{loc-ok}$, and $\mathrm{tloc}(\Theta; \tau_1) \cup \mathrm{tloc}(\Theta; \tau_2) \cup \rho = \rho'$.
  (1) By Lemma 22, $\Theta \vdash C[F \mapsto f, X \mapsto V] : \tau_2 \rhd \rho$, where $f = \mathsf{fun}_\rho\ F(X) \coloneqq C$.
  (2) By Lemma 28, $\Theta \vdash C[F \mapsto f, X \mapsto V]\ \texttt{loc-ok}$.
  (3) By assumption, $\rho' \subseteq \Omega$, which immediately implies that $\rho \subseteq \Omega$.
  (4) Since $\mathrm{SL}(C) = \mathrm{SL}(f) = \mathrm{SL}(V) = \varnothing$, $\mathrm{SL}(C[F \mapsto f, X \mapsto V]) = \varnothing$, and $\Omega' = \Omega$, this condition is satisfied.
  (5) Follows identically to (4).
  (6) We should show that $\mathrm{NL}(\rho) \setminus \mathrm{NL}(\rho') \subseteq \Omega' \setminus \Omega = \varnothing$, which is true precisely because $\mathrm{NL}(\rho) \subseteq \mathrm{NL}(\rho')$.
  (7) As $\Omega \setminus \Omega' = \varnothing$, (7) is trivially true.
- (C-TApp) We handle the case when the function's type variable is a location. The assumptions are that $\Theta, F : \forall \alpha :: *_{\mathrm{loc}}[\rho].\ \tau, \alpha :: *_{\mathrm{loc}} \vdash C : \tau \rhd \rho$, $\Theta, F : \forall \alpha :: *_{\mathrm{loc}}[\rho].\ \tau, \alpha :: *_{\mathrm{loc}} \vdash C\ \texttt{loc-ok}$, $\Theta \vdash \ell :: *_{\mathrm{loc}}$, and $\rho[\alpha \mapsto \ell] \cup \mathrm{tloc}(\Theta; \tau[\alpha \mapsto \ell]) = \rho'$.
  (1) By Lemma 22, $\Theta, \alpha :: *_{\mathrm{loc}} \vdash C[F \mapsto f] : \tau \rhd \rho$, where $f = \mathsf{tfun}_\top\ F(\alpha) \coloneqq C$, and by Lemma 17, $\Theta \vdash C[F \mapsto f, \alpha \mapsto \ell] : \tau[\alpha \mapsto \ell] \rhd \rho[\alpha \mapsto \ell]$.
  (2) By Lemmas 24 and 28, noting that $\mathrm{SL}(C) = \mathrm{SL}(f) = \varnothing$, we have $\Theta \vdash C[F \mapsto f, \alpha \mapsto \ell]\ \texttt{loc-ok}$.
  (3) By assumption, $\rho' \subseteq \Omega$, which immediately implies that $\rho[\alpha \mapsto \ell] \subseteq \Omega$.
  (4) As $\mathrm{SL}(C[F \mapsto f, \alpha \mapsto \ell]) = \varnothing$, and $\Omega' = \Omega$, this condition is satisfied.
  (5) Follows symmetrically to (4).
  (6) We should show that $\mathrm{NL}(\rho[\alpha \mapsto \ell]) \setminus \mathrm{NL}(\rho') \subseteq \Omega' \setminus \Omega = \varnothing$, which is true precisely because $\mathrm{NL}(\rho[\alpha \mapsto \ell]) \subseteq \mathrm{NL}(\rho')$.
  (7) As $\Omega \setminus \Omega' = \varnothing$, (7) is trivially true.

  The case when the function's type variable is a location set, program type, or local type is analogous.
- (C-TyLetV) We handle the case when the type variable bound by the type-let is a location. The assumptions are that $\Theta \vdash \rho_3.\lceil L \rfloor : \mathrm{loc}_{\rho_1} @ \rho_3 \rhd \varnothing$, $\Theta \vdash \tau :: *_{\rho_t}$, $\Theta, \alpha :: *_{\mathrm{loc}} \vdash C_2 : \tau \rhd \rho$, $\Theta, \alpha :: *_{\mathrm{loc}} \vdash C_2\ \texttt{loc-ok}$, $\rho_1 \subseteq \rho_2 \subseteq \rho_3$, $\mathrm{NL}(\rho_2) \cap \mathrm{SL}(C_2) = \varnothing$ $\mathrm{NL}(\rho_2) \cup \mathrm{NL}(\rho) \subseteq \Omega$, and by soundness of the loc type, $L \in \rho_1$.
  (1) By Lemma 17, $\Theta \vdash C_2[\alpha \mapsto L] : \tau \rhd \rho[\alpha \mapsto L]$.
  (2) As $L \in \rho_1 \subseteq \rho_2$, we have that $L \notin \mathrm{SL}(C_2)$ by well-scopedness of the entire type-let expression. Therefore by Lemma 24, we have $\Theta \vdash C_2[\alpha \mapsto L]\ \texttt{loc-ok}$.
  (3) By assumption, $\mathrm{NL}(\rho) \subseteq \Omega$ and $L \in \mathrm{NL}(\rho_2) \subseteq \Omega$, therefore $\mathrm{NL}(\rho[\alpha \mapsto L]) \subseteq \mathrm{NL}(\rho) \cup \{L\} \subseteq \Omega' = \Omega$.

(4) As $\mathrm{SL}(C_2[\alpha \mapsto L]) \setminus \mathrm{SL}(C_2) = \mathrm{SL}(C_2) \setminus \mathrm{SL}(C_2) = \varnothing = \Omega' \setminus \Omega$, this condition is satisfied.
(5) Follows symmetrically to (4).
(6) We should show that $\mathrm{NL}(\rho[\alpha \mapsto L]) \setminus (\mathrm{NL}(\rho) \cup \mathrm{NL}(\rho_2)) \subseteq \Omega' \setminus \Omega = \varnothing$, which is true precisely because

$$\begin{aligned}\mathrm{NL}(\rho[\alpha \mapsto L]) \setminus (\mathrm{NL}(\rho) \cup \mathrm{NL}(\rho_2)) &\subseteq (\mathrm{NL}(\rho) \cup \{L\}) \setminus (\mathrm{NL}(\rho) \cup \mathrm{NL}(\rho_2)) \\ &\subseteq (\mathrm{NL}(\rho) \cup \mathrm{NL}(\rho_2)) \setminus (\mathrm{NL}(\rho) \cup \mathrm{NL}(\rho_2)) \\ &= \varnothing\end{aligned}$$

(7) As $\Omega \setminus \Omega' = \varnothing$, (7) is trivially true.

The case when the type variable is a location set follow similar reasoning.

- (C-SendV) The assumptions are that $\Theta|_{\rho_1} \Vdash v : t_e$, $\{L\} \cup \mathrm{NL}(\rho_2) \subseteq \Omega$, and $L \in \rho_1$. The new expression is well-typed because, as v is a value, $\Vdash v : t_e$, and so $\Theta|_{(\rho_1 \cup \rho_2)} \Vdash v : t_e$ by weakening of the local type system. The other conclusions are also straightforward because there are no locations spawned or killed, and the reduct $(\rho_1 \cup \rho_2).v$ is a value.
- (C-Fork) The assumptions are that $\Theta, \alpha :: *_{\mathsf{loc}}, \{L, \alpha\}.x : \mathrm{loc}_\alpha \vdash C : \tau \rhd \rho, \Theta, \alpha :: *_{\mathsf{loc}}, \{L, \alpha\}.x : \mathrm{loc}_\alpha \vdash C$ loc-ok, $\Theta \vdash \tau :: *_{\rho'}$, $\{L\} \cup \mathrm{NL}(\rho) \subseteq \Omega$, $L \notin \mathrm{SL}(C)$, and $L' \notin \Omega$.
  (1) As well-typedness is preserved under substitution and α is not free in τ, $\Theta \vdash C' : \tau \rhd \rho[\alpha \mapsto L]$, where $C' = C[\alpha \mapsto L', x \mapsto \lceil L' \rfloor]$, and hence $\Theta \vdash$ kill L' after $C' : \tau \rhd \{L'\} \cup \rho[\alpha \mapsto L]$.
  (2) As $L' \notin \Omega \supseteq \mathrm{SL}(C)$, we have that $\Theta \vdash C'$ loc-ok. As well, because $\mathrm{SL}(C') = \mathrm{SL}(C)$, we have that $\Theta \vdash$ kill L' after C' loc-ok as desired.
  (3) $\mathrm{NL}(\{L\} \cup \rho[\alpha \mapsto L]) \subseteq \{L\} \cup \mathrm{NL}(\rho) \subseteq \{L\} \cup \Omega = \Omega'$
  (4) $\mathrm{SL}(\text{kill } L' \text{ after } C') \setminus \mathrm{SL}(\text{let } (\alpha, x) := L.\text{fork}() \text{ in } C) = \{L'\} = \Omega' \setminus \Omega$
  (5) $\mathrm{SL}(\text{let } (\alpha, x) := L.\text{fork}() \text{ in } C) \setminus \mathrm{SL}(\text{kill } L' \text{ after } C') = \varnothing = \Omega \setminus \Omega'$
  (6) As $L \in \Omega$ but $L' \notin \Omega$, we must have $L' \neq L$. As well, because $\mathrm{NL}(\rho) \subseteq \Omega$, it follows that $L' \notin \mathrm{NL}(\rho)$. Therefore

$$\begin{aligned}(\{L'\} \cup \mathrm{NL}(\rho[\alpha \mapsto L'])) \setminus (\{L\} \cup \mathrm{NL}(\rho)) &= (\{L'\} \cup \mathrm{NL}(\rho)) \setminus (\{L\} \cup \mathrm{NL}(\rho)) \\ &= \{L'\} \setminus (\{L\} \cup \mathrm{NL}(\rho)) \\ &= \{L'\} = \Omega' \setminus \Omega\end{aligned}$$

  (7) This conclusion holds trivially because $\Omega \setminus \Omega' = \varnothing$.
- (C-Kill) The assumptions are that $\Theta \vdash V : \tau \rhd \varnothing$, $\Theta \vdash V$ loc-ok, $\mathrm{Val}(V)$, and $L \in \Omega$.
  (1) Clearly V is well-typed.
  (2) Clearly the spawned locations in V are well-scoped.
  (3) As there are no participants in V because it is a value, clearly $\varnothing \subseteq \Omega' = \Omega \setminus L$.
  (4) $\mathrm{SL}(V) \setminus \mathrm{SL}(\text{kill } L \text{ after } V) = \varnothing \setminus \{L\} = \varnothing = \Omega' \setminus \Omega$
  (5) $\mathrm{SL}(\text{kill } L \text{ after } V) \setminus \mathrm{SL}(V) = \{L\} = \Omega \setminus \Omega'$
  (6) $\varnothing \setminus \{L\} = \varnothing \subseteq \Omega' \setminus \Omega = \varnothing$
  (7) $\{L\} \setminus \varnothing = \{L\} \supseteq \Omega \setminus \Omega' = \{L\}$
- (C-KillI) The assumptions are that $\Theta \vdash C : \tau \rhd \rho$, $\Theta \vdash C$ loc-ok, $\{L\} \cup \mathrm{NL}(\rho) \subseteq \Omega$, and $L \notin \mathrm{cloc}(C)$.
  (1) Clearly C is well-typed.
  (2) Clearly the spawned locations in C are well-scoped.
  (3) By assumption, $\mathrm{NL}(\rho) \subseteq \Omega$. As well, by Lemma 30 we can say that $L \notin \mathrm{NL}(\rho)$, and so $\mathrm{NL}(\rho) \subseteq \Omega \setminus \{L\} = \Omega'$.
  (4) $\mathrm{SL}(C) \setminus \mathrm{SL}(\text{kill } L \text{ after } C) = \varnothing = \Omega' \setminus \Omega$
  (5) $\mathrm{SL}(\text{kill } L \text{ after } C) \setminus \mathrm{SL}(C) = \{L\} = \Omega \setminus \Omega'$

(6) $\mathrm{NL}(\rho) \setminus (\{L\} \cup \mathrm{NL}(\rho)) = \varnothing \subseteq \Omega' \setminus \Omega = \varnothing$
(7) $(\{L\} \cup \mathrm{NL}(\rho)) \setminus \mathrm{NL}(\rho) = \{L\} \supseteq \Omega \setminus \Omega' = \{L\}$

□

Theorem 6 (Type Preservation). *If* $\Theta \vdash C : \tau \rhd \rho$, $\Theta \vdash C$ `loc-ok`, $\mathrm{NL}(\rho) \subseteq \Omega$, *and* $\langle C, \Omega\rangle \Longrightarrow_c^* \langle C', \Omega'\rangle$, *then there is some* ρ' *such that all of the following properties hold.*

(1) $\Theta \vdash C' : \tau \rhd \rho'$
(2) $\Theta \vdash C'$ `loc-ok`
(3) $\mathrm{NL}(\rho') \subseteq \Omega'$
(4) $\mathrm{SL}(C') \setminus \mathrm{SL}(C) = \Omega' \setminus \Omega$
(5) $\mathrm{SL}(C) \setminus \mathrm{SL}(C') = \Omega \setminus \Omega'$
(6) $\mathrm{NL}(\rho') \setminus \mathrm{NL}(\rho) \subseteq \Omega' \setminus \Omega$
(7) $\Omega \setminus \Omega' \subseteq \mathrm{NL}(\rho) \setminus \mathrm{NL}(\rho')$

Proof. By induction on the length of the reduction sequence. If the reduction is of length 0, the conclusion is immediate. Otherwise suppose that $\langle C_1, \Omega_1\rangle \Longrightarrow_c^* \langle C_2, \Omega_2\rangle \Longrightarrow_c \langle C_3, \Omega_3\rangle$, where we can apply the inductive hypothesis to the first reduction sequence, and then apply Theorem 31 to the last step. (1–3) hold by the conclusion of Theorem 31.

(4) We show that $\mathrm{SL}(C_3) \setminus \mathrm{SL}(C_1) \subseteq \Omega_3 \setminus \Omega_1$, with the opposite direction following a symmetric argument. Let $L \in \mathrm{SL}(C_3) \setminus \mathrm{SL}(C_1)$. In the case that $L \in \mathrm{SL}(C_2)$, we have that $L \in \mathrm{SL}(C_2) \setminus \mathrm{SL}(C_1)$, so $L \in \Omega_2 \setminus \Omega_1$ by (4) of the inductive hypothesis. It must be the case that $L \in \Omega_3$, for otherwise $L \in \Omega_2 \setminus \Omega_3$, which implies that $L \in \mathrm{SL}(C_2) \setminus \mathrm{SL}(C_3)$ by (5) of Theorem 31, a contradiction. Therefore $L \in \Omega_3 \setminus \Omega_1$ as desired. Now consider the case when $L \notin \mathrm{SL}(C_2)$. Then $L \in \mathrm{SL}(C_3) \setminus \mathrm{SL}(C_2)$, so $L \in \Omega_3 \setminus \Omega_2$ by (4) of the last step. It must be the case that $L \notin \Omega_1$, for otherwise $L \in \Omega_1 \setminus \Omega_2$, which implies that $L \in \mathrm{SL}(C_1) \setminus \mathrm{SL}(C_2)$ by (5) of the induction, a contradiction. Therefore $L \in \Omega_3 \setminus \Omega_1$ as desired.
(5) Symmetric to the argument for (4).
(6) Suppose that $L \in \mathrm{NL}(\rho_3) \setminus \mathrm{NL}(\rho_1)$. In the case that $L \in \mathrm{NL}(\rho_2)$, then $L \in \mathrm{NL}(\rho_2) \setminus \mathrm{NL}(\rho_1)$, so $L \in \Omega_2 \setminus \Omega_1$ by (6) of the induction. $L \in \mathrm{NL}(\rho_3)$, so $L \in \Omega_3$ by (2) of the last step, and hence $L \in \Omega_3 \setminus \Omega_1$ as desired. Otherwise in the case that $L \notin \mathrm{NL}(\rho_2)$, then $L \in \mathrm{NL}(\rho_3) \setminus \mathrm{NL}(\rho_2)$, so $L \in \Omega_3 \setminus \Omega_2$ by (6) of the last step. L cannot be in Ω_1, for otherwise $L \in \Omega_1 \setminus \Omega_2$, which by (7) of the inductive hypothesis would imply that $L \in \mathrm{NL}(\rho_1) \setminus \mathrm{NL}(\rho_2)$, a contradiction. Therefore $L \in \Omega_3 \setminus \Omega_1$ as (6) requires.
(7) Suppose that $L \in \Omega_1 \setminus \Omega_3$. If $L \in \Omega_2$, by (7) of the last step we have $L \in \mathrm{NL}(\rho_2) \setminus \mathrm{NL}(\rho_3)$. We must have that $L \in \mathrm{NL}(\rho_1)$, for otherwise $L \in \Omega_2 \setminus \Omega_1$ by (6) of the induction, a contradiction. Thus $L \in \mathrm{NL}(\rho_1)$, and $L \in \mathrm{NL}(\rho_1) \setminus \mathrm{NL}(\rho_3)$, as desired. Otherwise suppose that $L \notin \Omega_2$. In this case, $L \in \mathrm{NL}(\rho_1) \setminus \mathrm{NL}(\rho_2)$ by (7) of the induction. We must have that $L \notin \mathrm{NL}(\rho_3)$, for otherwise $L \in \Omega_3 \setminus \Omega_2$ by (6) of the last step, a contradiction. Therefore $L \in \mathrm{NL}(\rho_1) \setminus \mathrm{NL}(\rho_3)$ as (7) requires.

□

Theorem 7 (Type Preservation). *If* $\Theta \vdash C : \tau \rhd \rho$, $\Theta \vdash C$ `loc-ok`, $\mathrm{NL}(\rho) \subseteq \Omega$, *and* $\langle C, \Omega\rangle \Longrightarrow_c^* \langle C', \Omega'\rangle$, *then there is some* ρ' *such that* $\Theta \vdash C' : \tau \rhd \rho'$, $\Theta \vdash C'$ `loc-ok`, *and* $\mathrm{NL}(\rho') \subseteq \Omega'$.

Proof. An immediate corollary of Theorem 6. □

Theorem 8 (Progress). *If* $\vdash C : \tau \rhd \rho$ *then either* C *is a value, or there is some* C', Ω', *and* R *such that* $\langle C, \Omega\rangle \overset{R}{\Longrightarrow}_c \langle C', \Omega'\rangle$.

Proof. By induction on the typing derivation $\vdash C : \tau \rhd \rho$.

- (T-Var) This case is impossible as the context is empty.
- (T-Done) By local progress.
- (T-Fun) This choreography is already a value.
- (T-App) If either C_1 or C_2 can take a step, then take it. Otherwise if both C_1 and C_2 are values, then apply C-App.
- (T-TFunLoc, T-TFun) This choreography is already a value.
- (T-TAppLoc, T-TApp) If the function C_1 can take a step, then take it. Otherwise if C_1 is a type function, then apply C-TApp.
- (T-Pair) If either C_1 or C_2 can take a step, then take it. Otherwise if both C_1 and C_2 are values, then the pair is a value.
- (T-Inl, T-Inr, T-Fold) If the argument can take a step, then take it. Otherwise if it is a value, then the program is a value.
- (T-Fst, T-Snd, T-Unfold, T-Case, T-LocalCase) If the argument can take a step, then take it. Otherwise if it is a value, then apply the appropriate elimination rule.
- (T-LetLocal, T-LetLoc, T-LetLocSet) If the head can take a step, then take it. Otherwise if it is a value, then bind the variable as appropriate.
- (T-Send) If the argument can take a step, then take it. Otherwise if it is a value, then apply C-SendV.
- (T-Sync) Apply C-Sync.
- (T-Fork) Apply C-Fork. By the assumptions of our system and local language, there should always be another unused thread name, and a representation of that name.
- (T-Kill) Apply C-Kill.

□

Corollary 4 (Type Soundness). *If* $\vdash C : \tau \triangleright \rho$, $\vdash C$ `loc-ok`, *and* $\mathrm{NL}(\rho) \subseteq \Omega$, *then for any reachable configuration* $\langle C, \Omega\rangle \Longrightarrow_c^* \langle C', \Omega'\rangle$, *either* C' *is a value or there is some* C'', Ω'', *and* R *where* $\langle C', \Omega'\rangle \overset{R}{\Longrightarrow}_c \langle C'', \Omega''\rangle$.

Theorem 2 (Type Soundness). *If* $\vdash C : \tau \triangleright \rho$, *every location literal in* C *is in* Ω, *and* C *contains no* kill-after *expressions, then whenever* $\langle C, \Omega\rangle \Longrightarrow_c^* \langle C', \Omega'\rangle$, *either* C' *is a value, or* $\langle C', \Omega'\rangle$ *can step.*

Proof. A direct consequence of Corollary 4. □

E.3 Bisimulation Relation

Lemma 32 (Less-Than is Reflexive). *$E \preceq E$ for all network programs E.*

Lemma 33 (Less-Than is Transitive). *If $E_1 \preceq E_2$ and $E_2 \preceq E_3$ then $E_1 \preceq E_3$.*

Lemma 34 (Less-Than Relation Preserves Free Variables). *If $E_1 \preceq E_2$ then $\mathrm{fv}(E_1) \subseteq \mathrm{fv}(E_2)$.*

Lemma 35 (Merging Produces an Upper-Bound). *If $E_1 \sqcup E_2 = E$, then $E_1 \preceq E$ and $E_2 \preceq E$.*

Lemma 36 (Location Substitutions Preserve Less-Than). *For any location substitution σ, if $E_1 \preceq E_2$ then $E_1[\sigma] \preceq E_2[\sigma]$.*

Lemma 37 (Type Substitutions Preserve Less-Than). *For any type substitution σ, if $E_1 \preceq E_2$ then $E_1[\sigma] \preceq E_2[\sigma]$.*

Lemma 38 (Local Substitutions Preserve Less-Than). *For any local substitution σ, if $E_1 \preceq E_2$ then $E_1[\sigma] \preceq E_2[\sigma]$.*

Lemma 39 (Substitutions Preserve Less-Than). *For any pair σ_1, σ_2 of variable substitutions such that $\sigma_1(X) \preceq \sigma_2(X)$ for all X, if $E_1 \preceq E_2$, then $E_1[\sigma] \preceq E_2[\sigma]$.*

Corollary 5. *If $E_1 \preceq E_2$ and $V_1 \preceq V_2$, then $E_1[X \mapsto V_1] \preceq E_2[X \mapsto V_2]$.*

Definition 6 (Network Program Collapsing)**.** Let $\text{collapse}(E)$ be the structurally homomorphic function on network programs such that $\text{collapse}(E_1 \mathbin{;} E_2) = \text{collapse}(E_1) \fatsemi \text{collapse}(E_2)$. For instance, $\text{collapse}(\texttt{let}\ x \coloneqq E_1\ \texttt{in}\ E_2) = \texttt{let}\ x \coloneqq \text{collapse}(E_1)\ \texttt{in}\ \text{collapse}(E_2)$.

Lemma 40 (Collapsing Function is Less-Than)**.** $\text{collapse}(E) \preceq E$.

Proof. By induction on E. The only interesting case is when $E = E_1 \mathbin{;} E_2$, which holds by induction and as $\fatsemi$ preserves $\preceq$. □

Lemma 41 (Program Merging on Values)**.** *If $E_1 \sqcup E_2 = E$, then E_1 is a value $\Leftrightarrow$ E_2 is a value $\Leftrightarrow$ E is a value.*

Proof. By induction on E_1, and analyzing the possible cases of E_2. □

Lemma 42 (Collapsing Preserves Program Merging)**.** *If $E_1 \sqcup E_2 = E$ then* $\text{collapse}(E_1) \sqcup \text{collapse}(E_2) = \text{collapse}(E)$.

Proof. By induction on the definition of the merge function. The only interesting case is when $E_1 = E_{1,1} \mathbin{;} E_{1,2}$, $E_2 = E_{2,1} \mathbin{;} E_{2,2}$, and $E = (E_{1,1} \sqcup E_{2,1}) \mathbin{;} (E_{1,2} \sqcup E_{2,2})$. First suppose that $\text{collapse}(E_{1,1})$ is a value, in which case by Lemma 41 and the inductive hypothesis $\text{collapse}(E_{2,1})$ and $\text{collapse}(E_{1,1}) \sqcup \text{collapse}(E_{2,1})$ are also values. This implies that

$$\begin{aligned}
\text{collapse}(E_1) \sqcup \text{collapse}(E_2) &= (\text{collapse}(E_{1,1}) \fatsemi \text{collapse}(E_{1,2})) \sqcup (\text{collapse}(E_{2,1}) \fatsemi \text{collapse}(E_{2,2})) \\
&= \text{collapse}(E_{1,2}) \sqcup \text{collapse}(E_{2,2}) \\
&= (\text{collapse}(E_{1,1}) \sqcup \text{collapse}(E_{2,1})) \fatsemi (\text{collapse}(E_{1,2}) \sqcup \text{collapse}(E_{2,2})) \\
&= \text{collapse}(E_{1,1} \sqcup E_{2,1}) \fatsemi \text{collapse}(E_{1,2} \sqcup E_{2,2}) \\
&= \text{collapse}((E_{1,1} \sqcup E_{2,1}) \mathbin{;} (E_{1,2} \sqcup E_{2,2})) \\
&= \text{collapse}(E).
\end{aligned}$$

Now if $\text{collapse}(E_{1,1})$ is not a value, we similarly have that

$$\begin{aligned}
\text{collapse}(E_1) \sqcup \text{collapse}(E_2) &= (\text{collapse}(E_{1,1}) \fatsemi \text{collapse}(E_{1,2})) \sqcup (\text{collapse}(E_{2,1}) \fatsemi \text{collapse}(E_{2,2})) \\
&= (\text{collapse}(E_{1,1}) \mathbin{;} \text{collapse}(E_{1,2})) \sqcup (\text{collapse}(E_{2,1}) \mathbin{;} \text{collapse}(E_{2,2})) \\
&= (\text{collapse}(E_{1,1}) \sqcup \text{collapse}(E_{2,1})) \mathbin{;} (\text{collapse}(E_{1,2}) \sqcup \text{collapse}(E_{2,2})) \\
&= (\text{collapse}(E_{1,1}) \sqcup \text{collapse}(E_{2,1})) \fatsemi (\text{collapse}(E_{1,2}) \sqcup \text{collapse}(E_{2,2})) \\
&= \text{collapse}(E_{1,1} \sqcup E_{2,1}) \fatsemi \text{collapse}(E_{1,2} \sqcup E_{2,2}) \\
&= \text{collapse}((E_{1,1} \sqcup E_{2,1}) \mathbin{;} (E_{1,2} \sqcup E_{2,2})) \\
&= \text{collapse}(E).
\end{aligned}$$

□

Lemma 43 (Less-Than Relation Reflects Network-Program Merging)**.** *If $E_1' \preceq E_1$, $E_2' \preceq E_2$, $\text{collapse}(E_1') = E_1'$, $\text{collapse}(E_2') = E_2'$, and $E_1 \sqcup E_2 = E$, then there is some $E' \preceq E$ such that $E_1' \sqcup E_2' = E'$.*

Proof. By induction and case analysis of $\preceq$. The only interesting scenario is when the network programs are `allow-choice` expressions or sequencing operations.

First consider the case when

$$\begin{array}{l}\texttt{allow}\ \ell\ \texttt{choice}\\ \mid \mathbf{L} \Rightarrow E_1'\end{array} \preceq \begin{array}{l}\texttt{allow}\ \ell\ \texttt{choice}\\ \mid \mathbf{L} \Rightarrow E_1\\ \mid \mathbf{R} \Rightarrow E_3\end{array}$$

$$\begin{array}{l}\texttt{allow}\ \ell\ \texttt{choice}\\ \mid \mathbf{R} \Rightarrow E_2'\end{array} \preceq \begin{array}{l}\texttt{allow}\ \ell\ \texttt{choice}\\ \mid \mathbf{L} \Rightarrow E_4\\ \mid \mathbf{R} \Rightarrow E_2\end{array}$$

and

$$E = \begin{array}{l}\texttt{allow}\ \ell\ \texttt{choice}\\ \mid \mathbf{L} \Rightarrow E_1 \sqcup E_4\\ \mid \mathbf{R} \Rightarrow E_3 \sqcup E_2\end{array}$$

Then we have that

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

This suffices because by Lemma 35, $E_1' \preceq E_1 \preceq E_1 \sqcup E_4$, and $E_2' \preceq E_2 \preceq E_3 \sqcup E_2$. Now consider the case when

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

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

and

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

Then by induction, there is some $E_3 \preceq E_{1,1} \sqcup E_{2,1}$ where $E_{1,1}' \sqcup E_{1,2}' = E_3$, and some $E_4 \preceq E_{1,2} \sqcup E_{2,2}$ where $E_{2,1}' \sqcup E_{2,2}' = E_4$. Then the term

$$\begin{array}{ll}\texttt{allow}\ \ell\ \texttt{choice}\\ \mid \mathbf{L} \Rightarrow E_3 & \preceq E\\ \mid \mathbf{R} \Rightarrow E_4\end{array}$$

suffices. The other `allow-choice` cases are analogous to these two.

For sequencing operations, we note that the rule $E_1 \preceq V\ ;\ E_2$ does not apply because $\text{collapse}(V\ ;\ E_2) = E_2 \neq V\ ;\ E_2$, which violates the assumptions. Therefore the only possible scenario is $E_{1,1}'\ ;\ E_{1,2}' \preceq E_{1,1}\ ;\ E_{1,2}$ and $E_{2,1}'\ ;\ E_{2,2}' \preceq E_{2,1}\ ;\ E_{2,2}$, where $E = (E_{1,1} \sqcup E_{2,1})\ ;\ (E_{1,2} \sqcup E_{2,2})$. Then by induction, there is some $E_3 \preceq E_{1,1} \sqcup E_{2,1}$ where $E_{1,1}' \sqcup E_{1,2}' = E_3$, and some $E_4 \preceq E_{1,2} \sqcup E_{2,2}$ where $E_{2,1}' \sqcup E_{2,2}' = E_4$, so the term $E_3\ ;\ E_4$ suffices. □

Lemma 44 (Less-Than Reflects Values). *If* $E_1 \preceq E_2$ *and* $\text{Val}(E_2)$, *then* $\text{Val}(E_1)$.

Proof. By induction on the relation $E_1 \preceq E_2$. Note that the case when $E_1 \preceq V\ ;\ E_2$ is impossible, because the right-hand side is not a value. The other cases are straightforward, as all other rules (except `allow-choice` expressions, which are also not values) are homomorphic. □

Lemma 45 (Merging Preserves Steps). *If $E_1 \xrightarrow{l} E_1'$, $E_2 \xrightarrow{l} E_2'$, and $E_1 \sqcup E_2 = E$, then there is some E' and E'' such that $E' \preceq E''$, $E \xrightarrow{l} E''$, and $E_1' \sqcup E_2' = E'$.*

Proof. The interesting scenarios are for sequencing expressions and `allow-choice` expressions. Indeed, when $E_{1,1}\ ;\ E_{2,1} \xrightarrow{l} E_{1,1}'\ ;\ E_{2,1}$ and $E_{2,1}\ ;\ E_{2,2} \xrightarrow{l} E_{2,1}'\ ;\ E_{2,2}$, we can directly apply induction. Otherwise if $E_{2,1}\ ;\ E_{2,2} \xrightarrow{\iota} E_{2,2}$ because $\mathrm{Val}(E_{2,1})$, then by Lemma 41 $E_{1,1}$ is a value, and hence this is the only step the left-hand side can take, so the result is immediate. The same is symmetrically true if $E_{1,1}$ is a value.

For `allow-choice` expressions, note that the label l of each step is identical. This means that both E_1 and E_2 receive the same direction d, and because they are required to have that case defined, must both have at least this case defined. □

Lemma 46 (Simulating Less-Than is a Subrelation of Less-Than). *If $E_1 \precsim E_2$ then $E_1 \preceq E_2$.*

Lemma 47 (Simulating Less-Than is Reflexive). *$E \precsim E$ for all network programs E.*

Lemma 48 (Simulating Less-Than is Transitive). *If $E_1 \precsim E_2$ and $E_2 \precsim E_3$ then $E_1 \precsim E_3$.*

Lemma 49 (Location Substitutions Preserve Simulating Less-Than). *For any location substitution σ, if $E_1 \precsim E_2$ then $E_1[\sigma] \precsim E_2[\sigma]$.*

Lemma 50 (Type Substitutions Preserve Simulating Less-Than). *For any type substitution σ, if $E_1 \precsim E_2$ then $E_1[\sigma] \precsim E_2[\sigma]$.*

Lemma 51 (Local Substitutions Preserve Simulating Less-Than). *For any local substitution σ, if $E_1 \precsim E_2$ then $E_1[\sigma] \precsim E_2[\sigma]$.*

Lemma 52 (Substitutions Preserve Simulating Less-Than). *For any pair σ_1, σ_2 of variable substitutions such that $\sigma_1(X) \preceq \sigma_2(X)$ for all X, if $E_1 \precsim E_2$, then $E_1[\sigma] \preceq E_2[\sigma]$.*

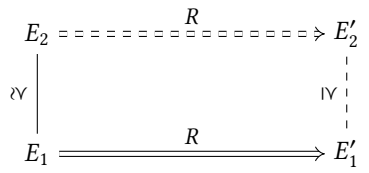
Corollary 6. *If $E_1 \precsim E_2$ and $V_1 \preceq V_2$, then $E_1[X \mapsto V_1] \preceq E_2[X \mapsto V_2]$.*

Lemma 53 (Simulating Less-Than Preserves and Reflects Values). *If $E_1 \precsim E_2$, then* $\mathrm{Val}(E_1) \Leftrightarrow \mathrm{Val}(E_2)$.

Lemma 54 (Simulating Less-Than is Reachable from Less-Than). *If $E_1 \preceq E_2$ then there is some E_2' such that $E_1 \precsim E_2'$ and $L \triangleright E_2 \xrightarrow{\iota}{}^{*} E_2'$ for any location L. That is, E_2 can reach E_2' through a series of internal steps.*

Proof. By induction on the definition of $\preceq$. For the case when $E_1 \preceq V\ ;\ E_2$, we can first step $V\ ;\ E_2 \xrightarrow{\iota} E_2$, and then by induction we can step $E_2 \xrightarrow{\iota}{}^{*} E_2'$, where $E_1 \precsim E_2'$, which is satisfactory. For those cases with a single head in the expression, apply the inductive hypothesis to the head of the expression, which is satisfactory. Lastly, for those cases with multiple evaluation positions (function applications and pairs), first apply the inductive hypothesis to the left expression. If it yields a non-value expression, this should suffice. Otherwise if it yields a value, also apply the inductive hypothesis to the right expression. □

Lemma 55 (Lifting Property). *If $E_1 \xrightarrow{R} E_1'$ and $E_1 \precsim E_2$, then there is some E_2' such that $E_1' \preceq E_2'$, and $E_2 \xrightarrow{R} E_2'$. That is, the following diagram holds:*



Proof. By induction on the definition of $\lesssim$. Each case follows by either applying the inductive hypothesis, or by recalling that substitution (of each sort) preserves the relation, or produces terms which are related by $\preceq$. □

E.4 Endpoint Projection

The following lemmas relate EPP to the substitution operations and the type system. Notably, we show that EPP is preserved under each of the sorts of variable substitution, with some specific conditions on the substitution depending on the sort.

Lemma 56 (Values Project to Values). *If* $\mathrm{Val}(V)$ *then* $\mathrm{Val}(\llbracket V \rrbracket_L)$ *for any* L.

Proof. By induction on V. □

Lemma 57 (EPP Reduces Free Variables). $\mathrm{fv}(\llbracket C \rrbracket_L) \subseteq \mathrm{fv}(C)$.

Proof. By induction on C. □

Lemma 58 (Location Substitution Preserves EPP). *If* $\llbracket C \rrbracket_L = E$ *and* $L \notin \sigma$, *then* $\llbracket C[\sigma] \rrbracket_L = E[\sigma]$.

Proof. By induction on C. Each case follows directly by induction, noting that Lemmas 11 and 12 guarantee that the same sub-case of EPP will be selected by $\llbracket C \rrbracket_L$ and $\llbracket C[\sigma] \rrbracket_L$. □

Lemma 59. *If* $\llbracket C \rrbracket_\alpha = E$ *and* $\mathrm{NL}(C) \notin \sigma$ *then* $\llbracket C[\sigma] \rrbracket_{\sigma(\alpha)} = E[\sigma]$.

Proof. Similar to Lemma 58. By induction on C, noting that the same sub-case of EPP will be selected by $\llbracket C \rrbracket_\alpha$ and $\llbracket C[\sigma] \rrbracket_{\sigma(\alpha)}$ because like location constants, variables are equal only to themselves, and because any value α may resolve to does not appear in C by assumption, and so may only appear in $C[\sigma]$ in places where α appears in C. For example, in the case of $C = \rho.e$, if $\alpha \in \rho$, and hence $\alpha \in \mathrm{fv}(\rho)$, we have that $\llbracket \rho.e \rrbracket_\alpha[\sigma] = \mathtt{ret}(e)[\sigma] = \mathtt{ret}(e[\sigma])$. Then because $\sigma(\alpha) \in \rho[\sigma]$, we have that $\llbracket \rho[\sigma].e[\sigma] \rrbracket_{\sigma(\alpha)} = \mathtt{ret}(e[\sigma])$ as expected. Otherwise if $\alpha \notin \rho$, and hence $\alpha \notin \mathrm{fv}(\rho)$, we have that $\llbracket \rho.e \rrbracket_\alpha[\sigma] = ()[\sigma] = ()$. Then because both $\alpha \notin \mathrm{fv}(\rho)$ and $\sigma(\alpha) \notin \mathrm{NL}(\rho.e) = \mathrm{NL}(\rho)$, we have that $\sigma(\alpha) \notin \rho[\sigma]$, and so $\llbracket \rho[\sigma].e[\sigma] \rrbracket_{\sigma(\alpha)} = ()$ as expected. □

Corollary 7. *If* $\llbracket C \rrbracket_\alpha = E$ *and* $L \notin \mathrm{NL}(C)$ *then* $\llbracket C[\alpha \mapsto L] \rrbracket_L = E[\alpha \mapsto L]$.

Lemma 60 (Type Substitution Preserves EPP). *If* $\llbracket C \rrbracket_L = E$, *then for any type variable substitution* σ *we have that* $\llbracket C[\sigma] \rrbracket_L = E[\sigma]$.

Proof. By induction on C. All cases follow directly by induction, noting that no location or location set in C will be affected by the substitution, so the same sub-case of EPP is selected. □

Lemma 61 (EPP is Fully Collapsed). *If* $\llbracket C \rrbracket_L = E$ *then* $\mathrm{collapse}(E) = E$.

Proof. By induction on C. Note that in the definition of EPP, the collapsing sequencing function $\fatsemi$ is always used instead of the primitive ; for sequencing two programs. Therefore each case follows directly by induction, and specifically because of the logic that if $\mathrm{collapse}(E_1) = E_1$ and $\mathrm{collapse}(E_2) = E_2$, then $\mathrm{collapse}(E_1 \fatsemi E_2) = \mathrm{collapse}(E_1) \fatsemi \mathrm{collapse}(E_2) = E_1 \fatsemi E_2$. □

Lemma 62 (Member Local Substitution Preserves EPP). *If $[\![C]\!]_L = E$ and $L \in \rho$, then for any local variable substitution σ we have that $[\![C[\rho|\sigma]]\!]_L = \text{collapse}(E[\sigma])$.*

Proof. By induction on C, noting that no location or location sets in C will be affected by the substitution. The interesting cases are for $\rho'.e$ and when ; can appear in the projection of C, such as $C = \text{let } \rho'.x := C_1 \text{ in } C_2$.

- If $\rho.e$ and $L \in \rho'$, we have that
$$[\![(\rho'.e)[\rho|\sigma]]\!]_L = [\![\rho'.e[\sigma]]\!]_L = \texttt{ret}(e[\sigma]) = \text{collapse}(\texttt{ret}(e[\sigma])).$$
Otherwise if $L \notin \rho'$ then $[\![\rho'.e[\sigma]]\!]_L = () = \text{collapse}(())$.
- For $\text{let } \rho'.x := C_1 \text{ in } C_2$ and $L \in \rho'$, we have that
$$\begin{aligned}[\![(\text{let } \rho'.x := C_1 \text{ in } C_2)[\rho|\sigma]]\!]_L &= [\![\text{let } \rho'.x := C_1[\rho|\sigma] \text{ in } C_2[\rho|\sigma]]\!]_L \\ &= \texttt{let } x := [\![C_1[\rho|\sigma]]\!]_L \texttt{ in } [\![C_2[\rho|\sigma]]\!]_L \\ &= \text{collapse}(\texttt{let } x := [\![C_1[\rho|\sigma]]\!]_L \texttt{ in } [\![C_2[\rho|\sigma]]\!]_L) \\ &= \text{collapse}(\texttt{let } x := [\![C_1]\!]_L\,[\sigma] \texttt{ in } [\![C_2]\!]_L\,[\sigma]) \\ &= \text{collapse}((\texttt{let } x := [\![C_1]\!]_L \texttt{ in } [\![C_2]\!]_L)[\sigma]).\end{aligned}$$
Otherwise if $L \notin \rho'$ then using Lemma 61 we see that
$$\begin{aligned}[\![\text{let } \rho'.x := C_1[\rho|\sigma] \text{ in } C_2[\rho|\sigma]]\!]_L &= [\![C_1[\rho|\sigma]]\!]_L \fatsemi [\![C_2[\rho|\sigma]]\!]_L \\ &= \text{collapse}([\![C_1]\!]_L\,[\sigma]) \fatsemi \text{collapse}([\![C_1]\!]_L\,[\sigma]) \\ &= \text{collapse}(([\![C_1]\!]_L \fatsemi [\![C_2]\!]_L)[\sigma]).\end{aligned}$$
- For $\text{case}_{\rho'}\ C \text{ of } (\text{inl } X \Rightarrow C_1)\ (\text{inr } Y \Rightarrow C_2)$, if $L \in \rho'$ the argument is straightforward by induction. Now consider the case when $L \notin \rho'$. We have that
$$\begin{aligned}\left[\!\!\left[\left(\begin{array}{l}\text{case}_{\rho'}\ C \text{ of} \\ \mid \text{inl } X \Rightarrow C_1 \\ \mid \text{inr } Y \Rightarrow C_2\end{array}\right)[\rho|\sigma]\right]\!\!\right]_L &= \left[\!\!\left[\begin{array}{l}\text{case}_{\rho'}\ C[\rho|\sigma] \text{ of} \\ \mid \text{inl } X \Rightarrow C_1[\rho|\sigma] \\ \mid \text{inr } Y \Rightarrow C_2[\rho|\sigma]\end{array}\right]\!\!\right]_L \\ &= [\![C[\rho|\sigma]]\!]_L \fatsemi [\![C_1[\rho|\sigma]]\!]_L \sqcup [\![C_2[\rho|\sigma]]\!]_L \\ &= \text{collapse}([\![C]\!]_L\,[\sigma]) \fatsemi \text{collapse}([\![C_1]\!]_L\,[\sigma]) \sqcup \text{collapse}([\![C_2]\!]_L\,[\sigma]) \\ &= \text{collapse}(([\![C]\!]_L \fatsemi [\![C_1]\!]_L \sqcup [\![C_2]\!]_L)[\sigma]),\end{aligned}$$
where the final equality uses Lemma 42.

□

Lemma 63 (Non-Member Local Substitution Preserves EPP). *If $[\![C]\!]_L = E$ and $L \notin \rho$, then for any local variable substitution σ we have that $[\![C[\rho|\sigma]]\!]_L = E$.*

Proof. The proof is nearly identical to Lemma 62. □

Corollary 8 (Local Substitution Preserves EPP). *If $[\![C]\!]_L = E$, then for any local variable substitution σ there is some $E' \preceq E$ such that $[\![C[\rho|\sigma]]\!]_L = E'$.*

Proof. If $L \in \rho$ then by Lemmas 40 and 62, $E' = \text{collapse}(E[\sigma])$ suffices. Otherwise if $L \notin \rho$, then by Lemma 63 and reflexivity of $\preceq$, $E' = E$ suffices. □

Definition 7. For a choreographic variable substitution σ_1 and a network-program variable substitution σ_2, say that $[\![\sigma_1]\!]_L = \sigma_2$ if and only if $[\![\sigma_1(X)]\!]_L = \sigma_2(X)$ for all program variables X.

Lemma 64 (Substitution Preserves EPP). *If $[\![C]\!]_L = E$ and $[\![\sigma_1]\!]_L = \sigma_2$, then $[\![C[\sigma_1]]\!]_L = \text{collapse}(E[\sigma_2])$.*

Proof. By induction on C.

- If $C = X$, then $[\![X[\sigma_1]]\!]_L = [\![\sigma_1(X)]\!]_L = \sigma_2(X)$ by the assumption and by Lemma 61.
- Let $C = \mathsf{let}\ \rho.x := C_1\ \mathsf{in}\ C_2$. If $L \in \rho$, the conclusion follows immediately by induction. Otherwise if $L \notin \rho$, then

$$\begin{aligned}
[\![(\mathsf{let}\ \rho.x := C_1\ \mathsf{in}\ C_2)[\sigma_1]]\!]_L &= [\![\mathsf{let}\ \rho.x := C_1[\sigma_1]\ \mathsf{in}\ C_2[\sigma_1]]\!]_L \\
&= [\![C_1[\sigma_1]]\!]_L \fatsemi [\![C_2[\sigma_1]]\!]_L \\
&= \mathrm{collapse}([\![C_1]\!]_L\ [\sigma_2]) \fatsemi \mathrm{collapse}([\![C_2]\!]_L\ [\sigma_2]) \\
&= \mathrm{collapse}(([\![C_1]\!]_L \fatsemi [\![C_2]\!]_L)[\sigma_2]) \\
&= \mathrm{collapse}([\![\mathsf{let}\ \rho.x := C_1\ \mathsf{in}\ C_2]\!]_L\ [\sigma_2]).
\end{aligned}$$

- Let $C = \mathsf{case}_\rho\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inr}\ Y \Rightarrow C_2)$. If $L \in \rho$, the conclusion follows immediately by induction. Otherwise if $L \notin \rho$ then, noting that $X \notin \mathrm{fv}([\![C_1]\!]_L)$ and $Y \notin \mathrm{fv}([\![C_2]\!]_L)$, we have that

$$\begin{aligned}
\left[\!\!\left[\left(\begin{array}{l}\mathsf{case}_\rho\ C\ \mathsf{of}\\ \mid \mathsf{inl}\ X \Rightarrow C_1\\ \mid \mathsf{inr}\ Y \Rightarrow C_2\end{array}\right)[\sigma_1]\right]\!\!\right]_L &= \left[\!\!\left[\begin{array}{l}\mathsf{case}_\rho\ C[\sigma_1]\ \mathsf{of}\\ \mid \mathsf{inl}\ X \Rightarrow C_1[X \mapsto X, Y \mapsto \sigma_1(Y)]\\ \mid \mathsf{inr}\ Y \Rightarrow C_2[X \mapsto X, Y \mapsto \sigma_1(Y)]\end{array}\right]\!\!\right]_L \\
&= [\![C[\sigma_1]]\!]_L \fatsemi [\![C_1[X \mapsto X, Y \mapsto \sigma_1(Y)]]\!]_L \sqcup [\![C_2[X \mapsto X, Y \mapsto \sigma_1(Y)]]\!]_L \\
&= \mathrm{collapse}([\![C]\!]_L\ [\sigma_2]) \fatsemi \mathrm{collapse}([\![C_1]\!]_L\ [\sigma_2]) \sqcup \mathrm{collapse}([\![C_2]\!]_L\ [\sigma_2]) \\
&= \mathrm{collapse}(([\![C]\!]_L \fatsemi [\![C_1]\!]_L \sqcup [\![C_2]\!]_L)[\sigma_2]).
\end{aligned}$$

  by applying Lemma 42.
- The other cases follow similar logic to those above.

□

Corollary 9. $[\![C[X \mapsto V]]\!]_L \preceq [\![C]\!]_L[X \mapsto [\![V]\!]_L]$.

Lemma 65 (Projection of Non-Participants). *If* $\Theta \vdash C : \tau \triangleright \rho$, $L \notin \rho$, *and* $[\![C]\!]_L = E$, *then* $\mathrm{Val}(E)$.

Proof. By induction on the typing derivation $\Theta \vdash C : \tau \triangleright \rho$. Most cases are straightforward or follow similar logic to a case shown below.

- (T-Var) We have $[\![X]\!]_L = X$, which is a value.
- (T-Done) If the MLV is a value, we have either $[\![\rho.v]\!]_L = \mathtt{ret}(v)$ or $[\![\rho.v]\!]_L = ()$, which is a value in either case. Otherwise if the MLV is not a value then $L \notin \rho$, so $[\![\rho.e]\!]_L = ()$.
- (T-Fun) If $[\![\mathsf{fun}_\rho\ F(X) := C]\!]_L$ is defined, then it is either $\mathtt{fun}$ or $()$, both of which are values.
- (T-App) Let $\Theta \vdash C_1 : \tau_1 \xrightarrow{\rho} \tau_2 \triangleright \rho_1$, $\Theta \vdash C_2 : \tau_2 \triangleright \rho_2$, and $\rho' = \mathrm{tloc}(\Theta; \tau_1) \cup \mathrm{tloc}(\Theta; \tau_2) \cup \rho$. By assumption $L \notin \rho' \cup \rho_1 \cup \rho_2$, so we can apply induction to C_1 and C_2 to see that $[\![C_1\ \$_{\rho'}\ C_2]\!]_L = [\![C_1]\!]_L \fatsemi [\![C_2]\!]_L \fatsemi () = ()$ as both $[\![C_1]\!]_L$ and $[\![C_2]\!]_L$ are values.
- (T-TFunLoc) If $[\![\mathsf{tfun}_\rho\ F(\alpha) := C]\!]_L$ is defined, then it is either $\mathtt{tfun}$ or $()$, both of which are values.
- (T-TAppLoc) Let $\Theta \vdash C_1 : \forall \alpha :: *_{\mathrm{loc}}\ [\rho].\ \tau \triangleright \rho_1$, $\Theta \vdash \ell :: *_{\mathrm{loc}}$, and $\rho' = \mathrm{tloc}(\Theta; \tau_1[\alpha \mapsto \ell]) \cup \rho[\alpha \mapsto \ell]$. By assumption $L \notin \rho' \cup \rho_1$, so we can apply induction to C_1 to see that $[\![C_1\ \$_{\rho'}\ \ell]\!]_L = [\![C_1]\!]_L \fatsemi () = ()$.
- (T-Pair) Let $\Theta \vdash C_1 : \tau_1 \triangleright \rho_1$ and $\Theta \vdash C_2 : \tau_1 \triangleright \rho_2$. By induction, both $[\![C_1]\!]_L$ and $[\![C_2]\!]_L$ are values. Therefore because either $[\![(C_1, C_2)_\rho]\!]_L = [\![C_1]\!]_L \fatsemi [\![C_2]\!]_L = [\![C_2]\!]_L$ if $L \notin \rho$ or otherwise $[\![(C_1, C_2)_\rho]\!]_L = ([\![C_1]\!]_L, [\![C_2]\!]_L)$, in either case the projection is a value. The argument for other introduction forms are identical.

- (T-Case) Let $\Theta \vdash C : \tau_1 +_{\rho'} \tau_2 \rhd \rho$, $\Theta, X : \tau_1 \vdash C_1 : \tau \rhd \rho_1$, and $\Theta, Y : \tau_2 \vdash C_2 : \tau \rhd \rho_2$. As $L \notin \rho \cup \rho_1 \cup \rho_2 \cup \rho'$, we can apply induction to all of C, C_1, and C_2. Therefore the projection is $[\![\mathsf{case}_{\rho'}\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inr}\ Y \Rightarrow C_2)]\!]_L = [\![C]\!]_L \fatsemi [\![C_1]\!]_L \sqcup [\![C_2]\!]_L = [\![C_1]\!]_L \sqcup [\![C_2]\!]_L$. By the assumption that the projection exists, it must be that $X \notin \mathrm{fv}([\![C_1]\!]_L)$, $Y \notin \mathrm{fv}([\![C_2]\!]_L)$, and the merge $[\![C_1]\!]_L \sqcup [\![C_2]\!]_L$ exists. Using Lemma 41, we find that $[\![C_1]\!]_L \sqcup [\![C_2]\!]_L$ is also a value. The argument for other elimination forms are identical.
- (T-LetLocal, T-LetLoc, T-LetLocSet) Let $\Theta \vdash C_1 : t_e @ \rho \rhd \rho_1$ and $\Theta, \rho'.x : t_e \vdash C_2 : \tau \rhd \rho_2$. The assumption is that $L \notin \rho' \cup \rho_1 \cup \rho_2$, so by induction $[\![\mathsf{let}\ \rho'.x : t_e := C_1\ \mathsf{in}\ C_2]\!]_L = [\![C_1]\!]_L \fatsemi [\![C_2]\!]_L = [\![C_2]\!]_L$, which is a value or variable. The same argument applies to the type-let expression.
- (T-Fork) Let $\Theta, \alpha :: *_{\mathsf{loc}}, \{\ell, \alpha\}.x : \mathrm{loc}_\alpha \vdash C : \tau \rhd \rho$ and $\Theta \vdash \tau :: *_{\rho_t}$. If $L \notin \{\ell\} \cup (\rho \setminus \{\alpha\})$, then $[\![\mathsf{let}\ (\alpha, x) := \ell.\mathsf{fork}()\ \mathsf{in}\ C]\!]_L = [\![C]\!]_L$. By assumption, $[\![C]\!]_L$ must be defined. As well, since $L \neq \alpha$, we have that $L \notin \rho$, so we can apply induction to C as desired.
- (T-Kill) Let $\Theta \vdash C : \tau \rhd \rho$, and $L \notin \rho \cup \{L'\}$. Then $[\![\mathsf{kill}\ L'\ \mathsf{after}\ C]\!]_L = [\![C]\!]_L$ is a value or variable by induction.

□

Lemma 66 (Projection of Non-Owners). *If* $\Theta \vdash C : \tau \rhd \rho$, $L \notin \rho \cup \mathrm{tloc}(\Theta; \tau)$, *and* $[\![C]\!]_L = E$, *then* $E = ()$ *or* $E = X$.

Proof. By induction on the typing derivation $\Theta \vdash C : \tau \rhd \rho$. Most cases are straightforward or follow similar logic to a case shown below.

- (T-Var) $[\![X]\!]_L = X$.
- (T-Done) If $L \notin \rho$ then $[\![\rho.e]\!]_L = ()$.
- (T-Fun) If $[\![\mathsf{fun}_\rho\ F(X) := C]\!]_L$ is defined and $L \notin \rho$, then it projects to $()$.
- (T-App) Let $\Theta \vdash C_1 : \tau_1 \xrightarrow{\rho} \tau_2 \rhd \rho_1$, $\Theta \vdash C_2 : \tau_2 \rhd \rho_2$, and $\rho' = \mathrm{tloc}(\Theta; \tau_1) \cup \mathrm{tloc}(\Theta; \tau_2) \cup \rho$. By assumption $L \notin \rho' \cup \rho_1 \cup \rho_2$, so as $L \notin \mathrm{tloc}(\Theta; \tau_2)$ and $L \notin \mathrm{tloc}(\Theta; \tau_1 \xrightarrow{\rho} \tau_2) = \mathrm{tloc}(\Theta; \tau_1) \cup \mathrm{tloc}(\Theta; \tau_2) \cup \rho$ we can apply induction to C_1 and C_2 to see that $[\![C_1\ \$_{\rho'}\ C_2]\!]_L = [\![C_1]\!]_L \fatsemi [\![C_2]\!]_L \fatsemi () = ()$.
- (T-TFunLoc) This case is vacuous as we can never have $L \notin \mathrm{tloc}(\Theta; \forall \alpha :: \kappa_\ell[\rho].\ \tau) = \top$.
- (T-TFun) If $L \notin \mathrm{tloc}(\Theta; \forall \alpha :: \kappa[\rho].\ \tau) = \rho \cup \mathrm{tloc}(\Theta, \alpha :: \kappa; \tau) = \rho'$, then $[\![\mathsf{tfun}_{\rho'}\ F(\alpha) := C]\!]_L = ()$.
- (T-TAppLoc, T-TApp) Let $\Theta \vdash C_1 : \forall \alpha :: \kappa_\ell[\rho].\ \tau \rhd \rho_1, \Theta \vdash \ell :: *_{\mathsf{loc}}$, and $\rho' = \mathrm{tloc}(\Theta; \tau[\alpha \mapsto \ell]) \cup \rho[\alpha \mapsto \ell]$. By assumption, $L \notin \rho' \cup \rho_1$. Then by Lemma 65, $[\![C_1]\!]_L$ must be a value. Therefore $[\![C_1\ \$_{\rho'}\ \ell]\!]_L = [\![C_1]\!]_L \fatsemi () = ()$. The argument for T-TApp is analogous.
- (T-Pair) Let $\Theta \vdash C_1 : \tau_1 \rhd \rho_1$ and $\Theta \vdash C_2 : \tau_2 \rhd \rho_2$. As $L \notin \mathrm{tloc}(\Theta; \tau_1 \times \tau_2) = \mathrm{tloc}(\Theta; \tau_1) \cup \mathrm{tloc}(\Theta; \tau_2) = \rho$, by induction, both C_1 and C_2 project to $()$ or a variable. Therefore $[\![(C_1, C_2)_\rho]\!]_L = [\![C_1]\!]_L \fatsemi [\![C_2]\!]_L$ is a value or variable. The argument for the other introduction forms is similar.
- (T-Case) Let $\Theta \vdash C : \tau_1 +_{\rho'} \tau_2 \rhd \rho$, $\Theta, X : \tau_1 \vdash C_1 : \tau \rhd \rho_1 \Theta_2$, and $\Theta, Y : \tau_2 \vdash C_2 : \tau \rhd \rho_2$. By assumption $L \notin \rho \cup \rho_1 \cup \rho_2 \cup \rho' \cup \mathrm{tloc}(\Theta; \tau)$, so we can apply induction to C_1 and C_2. By Lemma 65, $[\![C]\!]_L$ must be a value. Therefore the projection is $[\![\mathsf{case}_{\rho'}\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inr}\ Y \Rightarrow C_2)]\!]_L = [\![C]\!]_L \fatsemi [\![C_1]\!]_L \sqcup [\![C_2]\!]_L = [\![C_1]\!]_L \sqcup [\![C_2]\!]_L$. By the assumption that the projection exists, it must be that $X \notin \mathrm{fv}([\![C_1]\!]_L)$, $Y \notin \mathrm{fv}([\![C_2]\!]_L)$, and the merge $[\![C_1]\!]_L \sqcup [\![C_2]\!]_L$ exists. If either $[\![C_1]\!]_L$ or $[\![C_2]\!]_L$ equals $()$, they both must be, and hence their merge equals $()$. If either is a variable, they both must be the same variable by the fact that the merge exists, and hence their merge is a variable. The argument for the other elimination forms is similar.

- (T-LetLocal, T-LetLoc, T-LetLocSet) Let $\Theta \vdash C_1 : t_e@\rho \triangleright \rho_1$ and $\Theta, \rho'.x : t_e \vdash C_2 : \tau \triangleright \rho_2$. The assumption is that $L \notin \rho' \cup \rho_1 \cup \rho_2 \cup \mathrm{tloc}(\Theta; \tau)$, so by induction $[\![C_2]\!]_L$ is $()$ or a variable, and by Lemma 65 $[\![C_1]\!]_L$ must be a value. Therefore $[\![\text{let } \rho'.x : t_e \coloneqq C_1 \text{ in } C_2]\!]_L = [\![C_1]\!]_L \fatsemi [\![C_2]\!]_L = [\![C_2]\!]_L$. The same argument applies to the type-let expression.
- (T-Fork) Let $\Theta, \alpha :: *_{\mathrm{loc}}, \{\ell, \alpha\}.x : \mathrm{loc}_\alpha \vdash C : \tau \triangleright \rho$ and $\Theta \vdash \tau :: *_{\rho_\ell}$. If $L \notin \{\ell\} \cup (\rho \setminus \{\alpha\}) \cup \mathrm{tloc}(\Theta; \tau)$, then $[\![\text{let } (\alpha, x) \coloneqq \ell.\mathsf{fork}() \text{ in } C]\!]_L = [\![C]\!]_L$. By assumption, $[\![C]\!]_L$ must be defined. As well, since $L \neq \alpha$, we have that $L \notin \rho \cup \mathrm{tloc}(\Theta; \tau)$, so we can apply induction to C as desired.
- (T-Kill) Let $\Theta \vdash C : \tau \triangleright \rho$, and $L \notin \rho \cup \{L'\} \cup \mathrm{tloc}(\Theta; \tau)$. Then $[\![\text{kill } L' \text{ after } C]\!]_L = [\![C]\!]_L$ which satisfies the requirement by induction.

□

Lemma 67 (Projection of Non-Participant Values). *If* $\Theta \vdash V : \tau \triangleright \rho$, $\mathrm{Val}(V)$ *and* $L \notin \mathrm{tloc}(\Theta; \tau)$, *then* $[\![V]\!]_L = ()$.

Proof. By induction on the typing derivation $\Theta \vdash V : \tau \triangleright \rho$, similarly to Lemma 65. Note, however, that this Lemma is different than Lemma 65 because that there we pre-suppose that $[\![C]\!]_L$ exists, whereas here we do not. □

Lemma 68. *If* $L \notin \mathrm{cloc}(C)$ *and* $[\![C]\!]_L = E$, *then* $\mathrm{Val}(E)$.

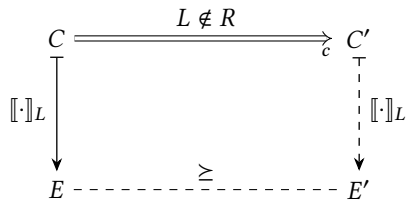
Proof. By induction on C. □

E.5 Completeness, Soundness, and Deadlock-Freedom

Lemma 69 (Labels Uniquely Determine Active Locations). *If* $L \triangleright \langle E_1, \Omega \rangle \overset{l}{\Longrightarrow} \langle E_1', \Omega_1' \rangle$ *and* $L \triangleright \langle E_2, \Omega \rangle \overset{l}{\Longrightarrow} \langle E_2', \Omega_2' \rangle$ *then* $\Omega_1' = \Omega_2'$.

Proof. By induction on the step, noting that the only times that Ω changes is when $l = \mathsf{fork}(L', E)$. But in this case, as the labels on the steps are identical, the same location must be added to Ω in both steps. □

Lemma 70 (Non-Participant Local Completeness). *If* $\Theta \vdash C : \tau \triangleright \rho$, $\langle C, \Omega \rangle \overset{R}{\Longrightarrow}_c \langle C', \Omega' \rangle$, $L \in \Omega \setminus \mathrm{rloc}(R)$, *and* $[\![C]\!]_L = E$, *then there is some* $E' \preceq E$ *such that* $[\![C']\!]_L = E'$. *That is, the following diagram holds.*



Proof. By induction on the step $\langle C, \Omega \rangle \overset{R}{\Longrightarrow}_c \langle C', \Omega' \rangle$.

- (C-Ctx) Straightforward by induction.
- (C-Done) Both sides of the step project to $()$.
- (C-App) Let $\Theta, F : \tau_1 \overset{\rho_1}{\longrightarrow} \tau_2, X : \tau_1 \vdash C : \tau_2 \triangleright \rho_1$, $\mathrm{tloc}(\Theta; \tau_1) \cup \mathrm{tloc}(\Theta; \tau_2) \cup \rho_1 = \rho$, and $\Theta \vdash V : \tau_1 \triangleright \rho_2$. By Lemma 67, the left side of the step projects to

$$[\![f \$_\rho V]\!]_L = () \fatsemi [\![V]\!]_L \fatsemi () = () \fatsemi () \fatsemi () = (),$$

where $f = \mathsf{fun}_\rho\, F(X) := C$ and $L \notin \rho$. By Lemma 22, $\Theta \vdash C[F \mapsto f, X \mapsto V] : \tau_2 \rhd \rho_1$. Therefore as $[\![C]\!]_L$ must exist by the definition of EPP on fun, by Lemma 65 $[\![C[F \mapsto f, X \mapsto V]]\!]_L$ is either a unit $()$ or variable Z, both of which are satisfactory because $Z \preceq ()$ and $() \preceq ()$.

- (C-TApp) We handle the case when the function's type variable is a location. The assumptions are that $\Theta, F : \forall \alpha :: *_{\mathsf{loc}}[\rho_1].\, \tau, \alpha :: *_{\mathsf{loc}} \vdash C : \tau \rhd \rho_1$, $\Theta \vdash \ell :: *_{\mathsf{loc}}$, and $L \notin \rho = \mathsf{tloc}(\Theta; \tau[\alpha \mapsto \ell]) \cup \rho_1[\alpha \mapsto \ell]$. The left side of the step projects to $[\![f\ \$_\rho\ \ell]\!]_L = () \fatsemi () = ()$, where $f = \mathsf{tfun}_\rho\, F(\alpha) := C$. By Lemma 22, $\Theta, \alpha :: *_{\mathsf{loc}} \vdash C[F \mapsto f] : \tau \rhd \rho_1$, and by Lemma 17, $\Theta \vdash C[F \mapsto f, \alpha \mapsto \ell] : \tau[\alpha \mapsto \ell] \rhd \rho_1[\alpha \mapsto \ell]$. Therefore as $[\![C]\!]_L$ must exist by the definition of EPP on tfun, by Lemma 65 $[\![C[F \mapsto f, \alpha \mapsto \ell]]\!]_L$ is either a unit or variable. The arguments when the function's type variable is a location set, program type, or local type are analogous.
- (C-UnfoldFold) By Lemma 67, the left side projects to $[\![\mathsf{unfold}_\rho\ (\mathsf{fold}_\rho\ V)]\!]_L = [\![V]\!]_L \fatsemi () = ()$. The right side also projects to $[\![V]\!]_L = ()$.
- (C-FstPair, C-SndPair) By Lemma 67, the left side projects to $[\![\mathsf{fst}_\rho\ (V_1, V_2)_\rho]\!]_L = [\![V_1]\!]_L \fatsemi [\![V_2]\!]_L \fatsemi () = ()$. The right side also projects to $[\![V_1]\!]_L = ()$. The case for C-SndPair is symmetric.
- (C-CaseInl, C-CaseInr) By Lemma 67, the left side projects to

$$[\![\mathsf{case}_\rho\ (\mathsf{inl}_\rho\ V)\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inr}\ Y \Rightarrow C_2)]\!]_L = [\![V]\!]_L \fatsemi [\![C_1]\!]_L \sqcup [\![C_2]\!]_L = [\![C_1]\!]_L \sqcup [\![C_2]\!]_L\,.$$

  As $X \notin \mathrm{fv}([\![C_1]\!]_L)$ by the above projection and by Lemma 64, the right side projects to

$$[\![C_1[X \mapsto V]]\!]_L \preceq [\![C_1]\!]_L\,[X \mapsto [\![V]\!]_L] = [\![C_1]\!]_L \preceq [\![C_1]\!]_L \sqcup [\![C_2]\!]_L\,.$$

  The case for C-CaseInr is symmetric.
- (C-LetV) The left side projects to $[\![\mathsf{let}\ \rho_1.x := \rho_2.v\ \mathsf{in}\ C]\!]_L = [\![\rho_2.v]\!]_L \fatsemi [\![C]\!]_L = [\![C]\!]_L$. By Lemma 63, the right side projects to $[\![C[\rho_1 | x \mapsto v]]\!]_L = [\![C]\!]_L$.
- (C-TyLetV) The left side projects to

$$[\![\mathsf{let}\ \rho_2.\alpha :: *_{\mathsf{loc}} := \rho_3.\lceil L' \rfloor\ \mathsf{in}\ C]\!]_L = [\![\rho_3.\lceil L' \rfloor]\!]_L \fatsemi [\![C]\!]_L = [\![C]\!]_L\,.$$

  By soundness of the local loc type, we must have that $L' \in \rho_1 \subseteq \rho_2 \subseteq \rho_3$, and hence $L \neq L'$. Therefore by Lemma 58, and as $\alpha \notin \mathrm{fv}([\![C]\!]_L)$, the right side projects to $[\![C[\alpha \mapsto L']]\!]_L = [\![C]\!]_L[\alpha \mapsto L'] = [\![C]\!]_L$ which suffices. The cases when the type variable is a location set is analogous.
- (C-SendV) The left side projects to $[\![\rho_1.v\ \{L'\}\!\rightsquigarrow\ \rho_2]\!]_L = [\![\rho_1.v]\!]_L$, and the right side projects to $[\![(\rho_1 \cup \rho_2).v]\!]_L$. Because $L \notin \rho_2$, whether or not $L \in \rho_1$ we have that projections are identical.
- (C-Sync) The left and right side both project to $[\![L'[d] \rightsquigarrow \rho\ ;\ C]\!]_L = [\![C]\!]_L$.
- (C-Fork) Let L'' be the newly spawned location. As $L \in \Omega$ but $L'' \notin \Omega$, we have that $L \neq L''$. Therefore by Lemmas 58 and 63, and as $\alpha, x \notin \mathrm{fv}([\![C]\!]_L)$, we have that

$$\begin{aligned}[\![\mathsf{let}\ (\alpha, x) := L'.\mathsf{fork}()\ \mathsf{in}\ C]\!]_L &= [\![C]\!]_L \\ &= [\![C]\!]_L\,[\alpha \mapsto L'', x \mapsto \lceil L'' \rfloor] \\ &= [\![C[\alpha \mapsto L'', \{L, L'\}.x \mapsto \lceil L'' \rfloor]]\!]_L\,,\end{aligned}$$

  which suffices.
- (C-Kill) For $L \neq L'$, we directly have that $[\![\mathsf{kill}\ L'\ \mathsf{after}\ V]\!]_L = [\![V]\!]_L$, so the conclusion is satisfied by reflexivity of $\preceq$.
- (C-KillI) Suppose the step is $\langle \mathsf{kill}\ L'\ \mathsf{after}\ C, \Omega\rangle \xrightarrow{R}_c \langle C, \Omega \setminus \{L'\}\rangle$ with $L' \notin \mathrm{cloc}(C)$ and $L \neq L'$. Then $[\![\mathsf{kill}\ L'\ \mathsf{after}\ C]\!]_L = [\![C]\!]_L$, so the conclusion similarly follows.

- (C-CaseI) First consider the case when $L \notin \rho$. We can apply the inductive hypothesis to C_1 and C_2 to see that

$$\begin{aligned}\left[\!\left[\mathsf{case}_\rho\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inr}\ Y \Rightarrow C_2)\right]\!\right]_L &= [\![C]\!]_L \fatsemi [\![C_1]\!]_L \sqcup [\![C_2]\!]_L \\ &\succeq [\![C]\!]_L \fatsemi [\![C_1']\!]_L \sqcup [\![C_2']\!]_L \\ &= \left[\!\left[\mathsf{case}_\rho\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1')\ (\mathsf{inr}\ Y \Rightarrow C_2')\right]\!\right]_L,\end{aligned}$$

  where the inequality holds because of Lemmas 43 and 61. The second equality holds because $X \notin \mathrm{fv}([\![C_1']\!]_L)$ and $Y \notin \mathrm{fv}([\![C_2']\!]_L)$ by Lemma 34 and the assumption that the original choreography projects. In the alternate case that $L \in \rho$ the logic is straightforward:

$$\begin{aligned}\left[\!\left[\mathsf{case}_\rho\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inr}\ Y \Rightarrow C_2)\right]\!\right]_L &= \mathsf{case}\ [\![C]\!]_L\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1)\ (\mathsf{inr}\ Y \Rightarrow C_2) \\ &\succeq \mathsf{case}\ [\![C]\!]_L\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1')\ (\mathsf{inr}\ Y \Rightarrow C_2') \\ &= \left[\!\left[\mathsf{case}_\rho\ C\ \mathsf{of}\ (\mathsf{inl}\ X \Rightarrow C_1')\ (\mathsf{inr}\ Y \Rightarrow C_2')\right]\!\right]_L.\end{aligned}$$

  The other out-of-order steps follow similar logic.

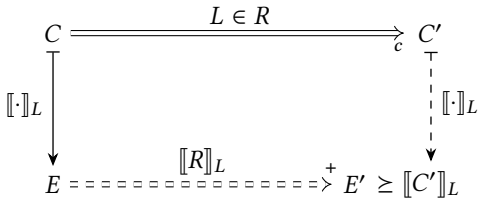
□

Corollary 10. *If* $\Theta \vdash C : \tau \triangleright \rho$, $\langle C, \Omega\rangle \overset{R}{\Longrightarrow}_c \langle C', \Omega'\rangle$, $L \in \Omega \setminus \mathrm{rloc}(R)$, *and* $[\![C]\!]_L^{\Uparrow} = E$, *then there is some* $E' \preceq E$ *such that* $[\![C']\!]_L^{\Uparrow} = E'$.

Proof. If $L \notin \mathrm{SL}(C)$, then this follows immediately by Lemma 70. Otherwise if $L \in \mathrm{SL}(C)$, then it either follows by setting $E' = E$ in the case that the reduction does not occur in the scope of the kill expression that L is executing, and otherwise if it does, the result also follows by applying Lemma 70 to that subexpression. □

Definition 8. For Ω a set of locations, let $\Omega|_L$ be the subset of Ω representing the children of L. That is, $L' \in \Omega|_L$ if and only if $L' \in \Omega$ and L has spawned L'.

Lemma 71 (Participant Local Completeness). *If* $\Theta \vdash C : \tau \triangleright \rho$, $\langle C, \Omega\rangle \overset{R}{\Longrightarrow}_c \langle C', \Omega'\rangle$, $L \in \Omega \cup \mathrm{rloc}(R)$, *$R$ is not a* kill *step, and* $[\![C]\!]_L = E$, *there is some* E_1' *and* E_2' *such that* $E_1' \preceq E_2'$, $[\![C']\!]_L = E_1'$, *and* $L \triangleright \langle E, \Omega|_L\rangle \overset{[\![R]\!]_L}{\Longrightarrow}{}^{+} \langle E_2', \Omega'|_L\rangle$. *That is, the following diagram holds.*



Proof. By induction on the step $\langle C, \Omega\rangle \overset{R}{\Longrightarrow}_c \langle C', \Omega'\rangle$.

- (C-Ctx) Straightforward by induction.
- (C-Done) Apply N-Ret.
- (C-App) The left side of the step projects to

$$\left[\!\left[f\ \$_\rho\ V\right]\!\right]_L = (\mathsf{fun}\ F(X) \coloneqq [\![C]\!]_L)\ [\![V]\!]_L.$$

  We can apply N-App to step to $[\![C]\!]_L[F \mapsto [\![f]\!]_L, X \mapsto [\![V]\!]_L]$, which is satisfactory by Lemma 64.

- (C-TApp) We consider the case when the type variable is a location. The left side of the step projects to

$$[\![f\ \$_\rho\ \ell]\!]_L = (\texttt{tfun}\, F(\alpha) \coloneqq \texttt{AmI}\ \alpha\ \texttt{then}\ [\![C[\alpha \mapsto L]]\!]_L\ \texttt{else}\ [\![C]\!]_L)\ \ell.$$

  We first apply N-TApp to step to

$$\texttt{AmI}\ \ell\ \texttt{then}\ [\![C[\alpha \mapsto L]]\!]_L \left[F \mapsto [\![f]\!]_L\right]\ \texttt{else}\ [\![C]\!]_L \left[F \mapsto [\![f]\!]_L, \alpha \mapsto \ell\right].$$

  If $L = \ell$, we apply N-IAmIn to then step to

$$[\![C[\alpha \mapsto L]]\!]_L \left[F \mapsto [\![f]\!]_L\right] \geq [\![C[F \mapsto f, \alpha \mapsto L]]\!]_L = [\![C']\!]_L\,.$$

  Otherwise suppose $L \neq \ell$. We apply N-IAmNotIn to step to $[\![C]\!]_L[F \mapsto [\![f]\!]_L, \alpha \mapsto \ell]$, and by Lemmas 58 and 64 have that

$$[\![C']\!]_L = [\![C[F \mapsto f, \alpha \mapsto \ell]]\!]_L = [\![C[F \mapsto f]]\!]_L\,[\alpha \mapsto \ell] \preceq [\![C]\!]_L \left[F \mapsto [\![f]\!]_L, \alpha \mapsto \ell\right]$$

  as required. The argument when the type variable is a location set, program type, or local type are analogous.
- (C-CaseInl, C-CaseInr) The left side projects to

$$\left[\!\!\left[\begin{array}{l}\mathsf{case}_\rho\ (\mathsf{inl}_\rho\ V)\ \mathsf{of}\\ \mid \mathsf{inl}\ X \Rightarrow C_1\\ \mid \mathsf{inr}\ Y \Rightarrow C_2\end{array}\right]\!\!\right]_L = \begin{array}{l}\texttt{case inl}\ [\![V]\!]_L\ \texttt{of}\\ \mid \texttt{inl}\ X \Rightarrow [\![C_1]\!]_L\\ \mid \texttt{inr}\ Y \Rightarrow [\![C_2]\!]_L\end{array}$$

  We apply N-CaseInl to step to $[\![C_1]\!]_L[X \mapsto V]$, which is satisfactory by Lemma 64. The argument for C-CaseInr is symmetric.
- (C-LetV) The left side projects to $[\![\mathsf{let}\ \rho_1.x \coloneqq \rho_2.v\ \mathsf{in}\ C]\!]_L = \texttt{let}\ x \coloneqq \texttt{ret}(v)\ \texttt{in}\ [\![C]\!]_L$ because $L \in \rho_1 \subseteq \rho_2$. We apply N-Let to step to $[\![C]\!]_L[\rho_1 | x \mapsto v]$, which is satisfactory by Corollary 8.
- (C-TyLetV) We consider the case when the type variable is a location. The left side of the step projects to

$$[\![\mathsf{let}\ \rho_2.\alpha \coloneqq \rho_3.\lceil L' \rfloor\ \mathsf{in}\ C]\!]_L = \texttt{let}\ \alpha \coloneqq \texttt{ret}(\lceil L' \rfloor)\ \texttt{in}\ \texttt{AmI}\ \alpha\ \texttt{then}\ [\![C[\alpha \mapsto L]]\!]_L\ \texttt{else}\ [\![C]\!]_L$$

  because $L \in \rho_2 \subseteq \rho_3$. We first apply N-TyLet to step to

$$\texttt{AmI}\ L'\ \texttt{then}\ [\![C[\alpha \mapsto L]]\!]_L\ \texttt{else}\ [\![C]\!]_L\,[\alpha \mapsto L'].$$

  If $L = L'$, we apply N-IAmIn to then step to

$$[\![C[\alpha \mapsto L]]\!]_L = [\![C[\alpha \mapsto L']]\!]_L = [\![C']\!]_L\,.$$

  Otherwise if $L \neq L'$ we apply N-IAmNotIn to step to $[\![C]\!]_L[\alpha \mapsto L']$, and by Lemma 58 we have that

$$[\![C']\!]_L = [\![C[\alpha \mapsto L']]\!]_L = [\![C]\!]_L\,[\alpha \mapsto L']$$

  as required. The argument when the type variable is a location set or local type are analogous.
- (C-Fork) Let L' be the newly spawned location. As $L \in \Omega$ but $L' \notin \Omega$, we have that $L \neq L'$. Thus the left side of the step projects to

$$[\![\mathsf{let}\ (\alpha, x) \coloneqq L.\mathsf{fork}()\ \mathsf{in}\ C]\!]_L = \texttt{let}\ (\alpha, x) \coloneqq \texttt{fork}([\![C]\!]_\alpha)\ \texttt{in}\ [\![C]\!]_L.$$

  By applying N-Fork we can step to $[\![C]\!]_L[\alpha \mapsto L', x \mapsto \lceil L' \rfloor]$, and by Lemma 58 and Corollary 8

$$[\![C']\!]_L = [\![C[\alpha \mapsto L', \{L, L'\}.x \mapsto \lceil L' \rfloor]]\!]_L \preceq [\![C]\!]_L\,[\alpha \mapsto L', x \mapsto \lceil L' \rfloor]$$

  as required.

- (C-CaseI) We can apply the inductive hypothesis to C_1 and C_2 to find some E_1 and E_2 such that $[\![C_1']\!]_L \preceq E_1$, $[\![C_2']\!]_L \preceq E_2$, $\langle [\![C_1]\!]_L, \Omega\rangle \xrightarrow{[\![R]\!]_L}{}^{+} \langle E_1, \Omega'\rangle$, and $\langle [\![C_2]\!]_L, \Omega\rangle \xrightarrow{[\![R]\!]_L}{}^{+} \langle E_2, \Omega'\rangle$, where the Ω' are equivalent by Lemma 43. Because $\rho \cap \mathrm{rloc}(R) = \varnothing$ and $L \in \mathrm{rloc}(R)$, we must have that $L \notin \rho$. Similarly because $\mathrm{cloc}(C) \cap \mathrm{rloc}(R) = \varnothing$, we have that $L \notin \mathrm{cloc}(C)$. Thus by Lemma 68, $[\![C]\!]_L$ is a value. Then the projection of the left-hand side is

$$\left[\!\!\left[\begin{array}{l} \mathsf{case}_\rho\ C\ \mathsf{of} \\ \mid \mathsf{inl}\ X \Rightarrow C_1 \\ \mid \mathsf{inr}\ Y \Rightarrow C_2 \end{array} \right]\!\!\right]_L = [\![C]\!]_L \fatsemi [\![C_1]\!]_L \sqcup [\![C_2]\!]_L = [\![C_1]\!]_L \sqcup [\![C_2]\!]_L\,,$$

and the projection of the right-hand side is

$$\left[\!\!\left[\begin{array}{l} \mathsf{case}_\rho\ C\ \mathsf{of} \\ \mid \mathsf{inl}\ X \Rightarrow C_1' \\ \mid \mathsf{inr}\ Y \Rightarrow C_2' \end{array} \right]\!\!\right]_L = [\![C]\!]_L \fatsemi [\![C_1']\!]_L \sqcup [\![C_2']\!]_L = [\![C_1']\!]_L \sqcup [\![C_2']\!]_L\,.$$

Using Lemma 45 allows the required steps to be made on the right-hand side. The other out-of-order steps follow similar logic.

□

Corollary 11. *If* $\Theta \vdash C : \tau \triangleright \rho$, $\langle C, \Omega\rangle \xrightarrow{R}_c \langle C', \Omega'\rangle$, $L \in \Omega \cup \mathrm{rloc}(R)$, *$R$ is not a* kill *step, and* $[\![C]\!]_L^{\Uparrow} = E$, *there is some* E_1' *and* E_2' *such that* $E_1' \preceq E_2'$, $[\![C']\!]_L^{\Uparrow} = E_1'$, *and* $L \triangleright \langle E, \Omega|_L\rangle \xrightarrow{[\![R]\!]_L}{}^{+} \langle E_2', \Omega'|_L\rangle$.

Proof. If $L \notin \mathrm{SL}(C)$, then this follows immediately by Lemma 71. Otherwise if $L \in \mathrm{SL}(C)$, then the reduction must occur in the scope of the kill expression that L is executing, and the result also follows by applying Lemma 71 to that subexpression. □

Lemma 72 (Kill-Step Local Completeness). *If* $\Theta \vdash C : \tau \triangleright \rho$, $\langle C, \Omega\rangle \xrightarrow{\mathsf{kill}(L)}_c \langle C', \Omega'\rangle$, $L \in \Omega$, *and* $[\![C]\!]_L^{\Uparrow} = E$, *then* $E \xrightarrow{\mathsf{exit}} ()$.

Proof. By induction on the step, similarly to Lemma 71. For the step C-Kill where $\langle\, \mathsf{kill}\ L\ \mathsf{after}\ V, \Omega\rangle \xrightarrow{R}_c \langle V, \Omega \setminus \{L\}\rangle$, the projection for L simply steps as $[\![V]\!]_L \fatsemi \mathtt{exit} = \mathtt{exit} \xrightarrow{\mathsf{exit}} ()$ because by Lemma 56, $[\![V]\!]_L$ is a value. If instead the step C-KillI occurs and $\langle \mathsf{kill}\ L\ \mathsf{after}\ C, \Omega\rangle \xrightarrow{R}_c \langle C, \Omega \setminus \{L\}\rangle$, the projection for L steps as $[\![C]\!]_L \fatsemi \mathtt{exit} = \mathtt{exit} \xrightarrow{\mathsf{exit}} ()$ because by Lemma 68, $[\![C]\!]_L$ is a value. □

Definition 9 (System Label Extraction). The label extraction function $\lfloor l_S \rfloor_L$ is a partial function which maps system labels to network program labels as follows:

$$\lfloor \iota_{L_1} \rfloor_L = \begin{cases} \iota & \text{if } L = L_1 \\ \text{undefined} & \text{otherwise} \end{cases} \qquad \lfloor L_1.m \rightsquigarrow \rho_2 \rfloor_L = \begin{cases} m \rightsquigarrow \rho_2 & \text{if } L = L_1 \\ L_1.m \rightsquigarrow & \text{if } L \neq L_1 \text{ and } L \in \rho_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\lfloor L_1.\mathtt{fork}(L_2, E) \rfloor_L = \begin{cases} \mathtt{fork}(L_2, E) & \text{if } L = L_1 \\ \text{undefined} & \text{otherwise} \end{cases} \qquad \lfloor \mathtt{kill}(L_1) \rfloor_L = \begin{cases} \mathtt{exit} & \text{if } L = L_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Lemma 73 (Single System Step Combining). *For all system labels* l_S *that are not* `fork` *or* `kill`, *if*

(1) for all locations L *such that* $\lfloor l_S \rfloor_L = l$, *both* $L \in \Omega$ *and* $L \triangleright \langle \Pi(L), \Omega|_L\rangle \xrightarrow{l} \langle \Pi'(L), \Omega|_L\rangle$,

(2) for all locations L such that $\lfloor l_S \rfloor_L$ = undefined, either $L \notin \Omega$ or $\Pi(L) = \Pi'(L)$,

then $\Pi \xRightarrow{l_S}_S \Pi'$, *where* $\Omega = \mathrm{dom}(\Pi) = \mathrm{dom}(\Pi')$.

Proof. By case analysis of the label l_S, noting that $\lfloor l_S \rfloor_L$ is defined precisely for those locations that will participate in the step. □

Definition 10 (catMaybes)**.** Let $\mathrm{catMaybes} : \mathrm{list}(\mathrm{maybe}(t)) \to \mathrm{list}(t)$ be the function which selects all defined entries in a list. For instance,

$$\mathrm{catMaybes}([1\,,\ \mathrm{undefined}\,,\ 2\,,\ 3\,,\ \mathrm{undefined}]) = [1\,,\ 2\,,\ 3].$$

Corollary 12 (System Step Combining)**.** *For all sequences of system labels $l_{S,1}\,,\ l_{S,2}\,,\ \ldots\,,\ l_{S,n}$ which are not* `fork` *or* `kill`*, if*

(1) $L \triangleright \langle \Pi(L), \Omega|_L \rangle \xRightarrow{\mathrm{catMaybes}([\lfloor l_{S,1} \rfloor_L\ ,\ \lfloor l_{S,2} \rfloor_L\ ,\ \ldots\ ,\ \lfloor l_{S,n} \rfloor_L])}{}^{*} \langle \Pi'(L), \Omega|_L \rangle$ *for all* $L \in \mathrm{dom}(\Pi) = \Omega$,

(2) $L \in \Omega$ *for all L such that at least one of the* $\lfloor l_{S,i} \rfloor_L$ *is defined,*

then $\Pi \xRightarrow{l_{S,1}\ ,\ l_{S,2}\ ,\ \ldots\ ,\ l_{S,n}}{}^{*}_S \Pi'$.

Proof. By induction on the length of the reduction, applying Lemma 73 repeatedly. □

Lemma 74 (Redex Projection Commutes)**.** *For all locations L and redices R,*

$$\mathrm{catMaybes}([\lfloor l_S \rfloor_L \mid l_S \in [\![R]\!]_{\mathcal{L}}]) = [\![R]\!]_L\,.$$

That is, the subsequence of system labels in $[\![R]\!]_{\mathcal{L}}$ which involve a location L is given precisely by the single-location projection $[\![R]\!]_L$.

Proof. By induction on R. □

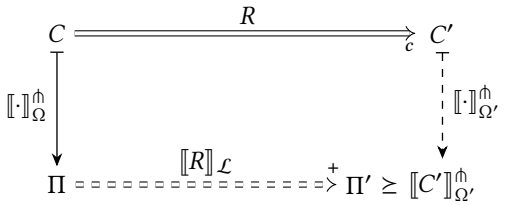
Lemma 75. *If* $\langle C, \Omega \rangle \xRightarrow{L.\mathtt{fork}(L', C'')}_c \langle C', \Omega' \rangle$, $\Theta \vdash C : \tau \triangleright \rho$, $\Theta \vdash C$ `loc-ok`, $\mathrm{NL}(\rho) \subseteq \Omega$, *and* $[\![C]\!]_L = E$, *then there is some E'' such that* $[\![C'']\!]_{L'} \preceq E''$ *and* $[\![C']\!]^{\Uparrow}_{L'} = [\![C'']\!]_{L'}\ \mathtt{;}\ \mathtt{exit}$.

Proof. By induction on the step. The out-of-order steps and steps in an evaluation context follow by induction, noting that $L' \notin \Omega$, and hence $L' \notin \mathrm{SL}(C)$, so no other `kill` expressions than the one created by this step may contain L'. For a C-Fork step $\mathtt{let}\ (\alpha, x) := L.\mathtt{fork}()\ \mathtt{in}\ C'' \Longrightarrow_c \mathtt{kill}\ L'\ \mathtt{after}\ C''[\alpha \mapsto L', x \mapsto \lceil L' \rfloor]$ with $L' \notin \mathrm{NL}(C'')$, the assumption that the left-hand side projects for L means that $[\![C'']\!]_\alpha$ must exist. Then by Lemmas 59 and 62, we have that

$$\begin{aligned}[\![C''[\alpha \mapsto L', x \mapsto \lceil L' \rfloor]]\!]_{L'} &\preceq [\![C''[\alpha \mapsto L']]\!]_{L'}\,[x \mapsto \lceil L' \rfloor] \\ &= [\![C'']\!]_\alpha\,[\alpha \mapsto L', x \mapsto \lceil L' \rfloor]\end{aligned}$$

which, as desired, is defined, and $\succeq$ the required projections. □

Lemma 76 (Single-Step Completeness)**.** *If* $\Theta \vdash C : \tau \triangleright \rho$, $\Theta \vdash C$ `loc-ok`, $\mathrm{NL}(\rho) \subseteq \Omega$, $[\![C]\!]^{\Uparrow}_\Omega = \Pi$, *and* $\langle C, \Omega \rangle \xRightarrow{R}_c \langle C', \Omega' \rangle$, *then there is some* Π'_1 *and* Π'_2 *such that* $\Pi'_1 \preceq \Pi'_2$, $[\![C']\!]^{\Uparrow}_{\Omega'} = \Pi'_1$, *and* $\Pi \xRightarrow{[\![R]\!]_{\mathcal{L}}}{}^{+}_S \Pi'_2$. *That is, the following diagram holds.*



Proof. First the case for steps which are not fork or kill. By Corollary 12 and Lemma 74, it suffices to prove that

$$L \triangleright \langle [\![C]\!]_L^{\Uparrow}, \Omega|_L \rangle \xrightarrow{[\![R]\!]_L}{}^{*} \langle \Pi_2'(L), \Omega|_L \rangle$$

and $[\![C']\!]_L^{\Uparrow} \preceq \Pi_2'(L)$ for each location $L \in \Omega$. If $L \notin \mathrm{rloc}(R)$, this holds by Corollary 10, noting that $[\![R]\!]_L$ is empty. Otherwise if $L \in \mathrm{rloc}(R)$, this is precisely Corollary 11. To apply Corollary 12 we also need to show that $\mathrm{rloc}(R) \subseteq \Omega$. This follows by soundness of participants (Theorem 1), and as $\mathrm{NL}(\rho) \subseteq \Omega$ by assumption. That is, $L \in \mathrm{rloc}(R) \subseteq \mathrm{NL}(\rho) \subseteq \Omega$. As well, there is always at least one location in the system that makes a step.

For the case of a $L.\mathsf{fork}(L', C'')$ step, for each location already in the system—in Ω—we can either apply Corollary 10 for $L'' \neq L, L'$, or Corollary 11 for L. The new thread L' does not need to make a step, but we must show that $[\![C']\!]_{L'}^{\Uparrow} \preceq [\![C'']\!]_{L'}$; exit, and that this projection exists—this is precisely Lemma 75.

For a $\mathsf{kill}(L)$ step, for each location $L' \neq L$, we apply Corollary 10. For L, since they were removed from Ω and the system, we do not need to worry about their projection, and can simply apply Lemma 72 to allow the system to perform a $\mathtt{kill}(L)$ step. □

Lemma 77 (System Single-Step Lifting). *If* $\Pi_1 \overset{l}{\Longrightarrow}_S \Pi_1'$ *and* $\Pi_1 \precsim \Pi_2$ *then there is some* Π_2' *such that* $\Pi_2 \overset{l}{\Longrightarrow}_S \Pi_2'$ *and* $\Pi_1' \precsim \Pi_2'$.

Proof. Follows via a case analysis of the step and using Lemma 55, noting that by definition if $\Pi_1 \precsim \Pi_2$ then $\mathrm{dom}(\Pi_1) = \mathrm{dom}(\Pi_2)$, so for the `fork` and `kill` steps, the respective systems after the steps will still have the same domains, and the network program of any spawned thread will be identical in Π_1' and Π_2'. □

Corollary 13. *If* $\Pi_1 \preceq \Pi_2$ *then there is some* Π_2' *such that* $\Pi_1 \precsim \Pi_2'$ *and* $\Pi_2 \Longrightarrow_S^{*} \Pi_2'$.

Proof. Follows by Lemmas 54 and 73, noting that the reduction sequence taken by each location are all ι steps, and hence can all happen independently. □

Lemma 78 (System Lifting Property). *If* $\Pi_1 \Longrightarrow_S^n \Pi_1'$ *and* $\Pi_1 \preceq \Pi_2$ *then there is some* Π_2' *and* $k \geq n$ *such that* $\Pi_2 \Longrightarrow_S^k \Pi_2'$ *and* $\Pi_1' \preceq \Pi_2'$.

Proof. By induction on the length n of the initial reduction sequence. If $n = 0$ the conclusion is trivial by choosing $k = 0$ and $\Pi_2' = \Pi_2$. Otherwise suppose the reduction is $\Pi_1 \Longrightarrow_S^n \Pi_1' \Longrightarrow_S \Pi_1''$. By induction, there is some Π_2' and $k \geq n$ where $\Pi_2 \Longrightarrow_S^k \Pi_2'$ and $\Pi_1' \preceq \Pi_2'$. By Corollary 13, we can step $\Pi_2' \Longrightarrow_S^{*} \Pi_2''$ where $\Pi_1' \precsim \Pi_2''$. By Lemma 77, we can take a step $\Pi_2'' \Longrightarrow_S \Pi_2'''$ to some Π_2''' where $\Pi_1'' \preceq \Pi_2'''$. Then the reduction sequence $\Pi_2 \Longrightarrow_S^k \Pi_2' \Longrightarrow_S^{*} \Pi_2'' \Longrightarrow_S \Pi_2'''$ is precisely as is required. □

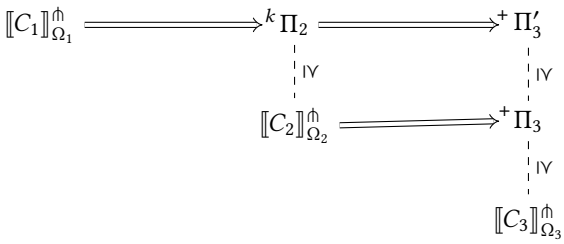
Theorem 9 (Completeness). *If* $\Theta \vdash C : \tau \triangleright \rho$, $\Theta \vdash C$ loc-ok, $\mathrm{NL}(\rho) \subseteq \Omega$, $\langle C, \Omega \rangle \Longrightarrow_c^n \langle C', \Omega' \rangle$, *and* $[\![C]\!]_{\Omega}^{\Uparrow} = \Pi$, *then there is some* $k \geq n$, Π_1', *and* Π_2' *such that* $\Pi_1' \precsim \Pi_2'$, $[\![C']\!]_{\Omega'}^{\Uparrow} = \Pi_1'$, *and* $\Pi \Longrightarrow_S^k \Pi_2'$.

Proof. By induction on the number of steps n. The case when $n = 0$ is trivial. For $n > 0$, we have a reduction sequence of the form

$$\langle C_1, \Omega_1 \rangle \Longrightarrow_c^n \langle C_2, \Omega_2 \rangle \Longrightarrow_c \langle C_3, \Omega_3 \rangle.$$

By the inductive hypothesis, there is some $k \geq n$ and Π_2 where $[\![C_2]\!]_{\Omega_2}^{\Uparrow} \preceq \Pi_2$ and $[\![C_1]\!]_{\Omega_1}^{\Uparrow} \Longrightarrow_S^k \Pi_2$. By Type Preservation (Theorem 7), C_2 is typed as $\Theta \vdash C_2 : \tau \triangleright \rho_2$ for some ρ_2, $\Theta \vdash C_2$ loc-ok, and $\mathrm{NL}(\rho_2) \subseteq \Omega_2$. Thus we can apply Lemma 76 to C_2 to find some Π_3 such that $[\![C_3]\!]_{\Omega_3}^{\Uparrow} \preceq \Pi_3$

and $[\![C_2]\!]^{\Uparrow}_{\Omega_2} \Longrightarrow_S^+ \Pi_3$. By Lemma 78, there is some $\Pi_3' \succeq \Pi_3$ such that $\Pi_2 \Longrightarrow_S^+ \Pi_3'$. Then Π_3' is satisfactory, as $[\![C_1]\!]^{\Uparrow}_{\Omega_1} \Longrightarrow_S^k \Pi_2 \Longrightarrow_S^+ \Pi_3'$ and $[\![C_3]\!]^{\Uparrow}_{\Omega_3} \preceq \Pi_3 \preceq \Pi_3'$. The argument is summarized by the following diagram.



□

Theorem 3 (Completeness). *If $\Theta \vdash C : \tau \rhd \rho$, every location literal in C is in Ω, and C contains no* kill-after *expressions, then whenever $\langle C, \Omega\rangle \Longrightarrow_c^* \langle C', \Omega'\rangle$, there is some Π' such that $[\![C]\!]^{\Uparrow}_{\Omega} \Longrightarrow_S^* \Pi'$ and $[\![C']\!]^{\Uparrow}_{\Omega'} \preceq \Pi'$.*

Proof. Follows directly from Theorem 9. □

Lemma 79 (Network Program Diamond Lemma). *If $L \rhd \langle E_1, \Omega_1\rangle \overset{l_1}{\Longrightarrow} \langle E_2, \Omega_2\rangle$ and $L \rhd \langle E_1, \Omega_1\rangle \overset{l_2}{\Longrightarrow} \langle E_3, \Omega_3\rangle$, where $E_2 \neq E_3$, then either both steps are message receives from the same sender, or we can find some E_4 and Ω_4 such that $L \rhd \langle E_2, \Omega_2\rangle \overset{l_2}{\Longrightarrow} \langle E_4, \Omega_4\rangle$ and $L \rhd \langle E_3, \Omega_3\rangle \overset{l_1}{\Longrightarrow} \langle E_4, \Omega_4\rangle$,*

Proof. By induction on the first step, and case analysis of the second step. The only interesting cases are when both reductions are ι steps to reduce the same local program. If this is the case, because we assume the diamond lemma (Property (2)) for local programs, we can reduce the local programs—and hence network programs—to a common reduct. □

Lemma 80 (System Diamond Lemma). *If $\Pi_1 \Longrightarrow_S \Pi_2$ and $\Pi_1 \Longrightarrow_S \Pi_3$, where $\Pi_2 \neq \Pi_3$, then there is some Π_4 where $\Pi_2 \Longrightarrow_S \Pi_4$ and $\Pi_3 \Longrightarrow_S \Pi_4$.*

Proof. By case analysis of the steps. If both steps are different cases (i.e., an ι step and message receive), or when both steps are ι, `kill`, or `fork`, we can apply Lemma 79. Note for `fork` steps that there is non-determinism in the choice of spawned thread name. This is not an issue, as we can simply equate two systems modulo a permutation given by the choice of spawned thread name.

Now consider the case when both steps are message sends of the form $L_1.m_1 \rightsquigarrow \rho_1$ and $L_2.m_2 \rightsquigarrow \rho_2$. We must have that $L_1 \neq L_2$, for otherwise as there is exactly one message the sender can send, we would have $m_1 = m_2$ and $\rho_1 = \rho_2$, which violates the assumption that $\Pi_2 \neq \Pi_3$. But in this case we can simply apply Lemma 79 as the senders are distinct. □

Lemma 81. *If $\Pi_1 \Longrightarrow_S \Pi_2$ and $\Pi_1 \Longrightarrow_S^n \Pi_3$, then either $\Pi_2 \Longrightarrow_S^{n-1} \Pi_3$, or there is some Π_4 such that $\Pi_2 \Longrightarrow_S^n \Pi_4$ and $\Pi_3 \Longrightarrow_S \Pi_4$*

Proof. By induction on n. If the second reduction sequence is of length 0, we trivially satisfy the second case as $\Pi_3 = \Pi_1$. Otherwise suppose the reduction sequence is of the form $\Pi_1 \Longrightarrow_S^n \Pi_3 \Longrightarrow_S \Pi_4$. We can apply induction to the pair Π_2 and Π_3. In the first case, where $\Pi_2 \Longrightarrow_S^{n-1} \Pi_3$, the first case also applies for the larger reduction sequence with the witness $\Pi_2 \Longrightarrow_S^{n-1} \Pi_3 \Longrightarrow_S \Pi_4$ of length n. Otherwise suppose that there is some Π_5 where $\Pi_2 \Longrightarrow_S^n \Pi_5$ and $\Pi_3 \Longrightarrow_S \Pi_5$.

First suppose that $\Pi_4 \neq \Pi_5$. Then by Lemma 80, we can find some Π_6 such that $\Pi_4 \Longrightarrow_S \Pi_6$ and $\Pi_5 \Longrightarrow_S \Pi_6$, which is satisfactory with the witness reduction sequence $\Pi_2 \Longrightarrow_S^n \Pi_5 \Longrightarrow_S \Pi_6$ of length $n+1$.

Otherwise let $\Pi_4 = \Pi_5$. But then we have a reduction $\Pi_2 \Longrightarrow_S^n \Pi_4$, which is satisfactory. □

Lemma 82 (System Confluence). *If $\Pi_1 \Longrightarrow_S^m \Pi_2$ and $\Pi_1 \Longrightarrow_S^n \Pi_3$, then there is some Π_4, m', and n' where $\Pi_2 \Longrightarrow_S^{n'} \Pi_4$, $\Pi_3 \Longrightarrow_S^{m'} \Pi_4$, $m' \leq m$, $n' \leq n$, and $m' \geq m - n$.*

Proof. By induction on n. If $n = 0$, the conclusion follows by choosing $\Pi_4 = \Pi_2$. Otherwise suppose the second reduction sequence is of the form $\Pi_1 \Longrightarrow_S^n \Pi_3 \Longrightarrow_S \Pi_4$. First we apply the inductive hypothesis to find some Π_5, m', and n' with the required properties. Now we apply Lemma 81 to the two reductions $\Pi_3 \Longrightarrow_S \Pi_4$ and $\Pi_3 \Longrightarrow_S^{m'} \Pi_5$.

First the case where $\Pi_4 \Longrightarrow_S^{m'-1} \Pi_5$. The reduction sequences $\Pi_2 \Longrightarrow_S^{n'} \Pi_5$ and $\Pi_4 \Longrightarrow_S^{m'-1} \Pi_5$ are satisfactory because $m' - 1 < m' \leq m$, $n' \leq n$, and $m' \geq m - n \Rightarrow m' - 1 \geq m - (n+1)$.

Now consider the case when there is some Π_6 where $\Pi_4 \Longrightarrow_S^{m'} \Pi_6$ and $\Pi_5 \Longrightarrow_S \Pi_6$. Then the reduction sequences $\Pi_2 \Longrightarrow_S^{n'} \Pi_5 \Longrightarrow_S \Pi_6$ and $\Pi_4 \Longrightarrow_S^{m'} \Pi_6$ are satisfactory because $m' \leq m$, $n' \leq n \Rightarrow n' + 1 \leq n + 1$, and $m' \geq m - n \Rightarrow m' \geq m - (n+1)$. □

Theorem 4 (Soundness). *If $\vdash C : \tau \triangleright \rho$, every location literal in C is in Ω, C contains no* kill-after *expressions, and $[\![C]\!]_\Omega^\Uparrow \Longrightarrow_S^* \Pi$ where Π is final, then $\langle C, \Omega\rangle \Longrightarrow_c^* \langle V, \Omega'\rangle$ where $[\![V]\!]_{\Omega'}^\Uparrow \preceq \Pi$.*

Proof. First we claim that C must terminate. Indeed, if it did not, then by Corollary 4, it will loop. But then by completeness (Theorem 9) and confluence (Lemma 82), Π must be able to take a step, contradicting the fact that it is final.

Now suppose that that C is executed as $\langle C, \Omega\rangle \Longrightarrow_c^* \langle V, \Omega'\rangle$. Then by completeness, there is some $\Pi' \succeq [\![V]\!]_{\Omega'}^\Uparrow$ such that $[\![C]\!]_\Omega^\Uparrow \Longrightarrow_S^* \Pi'$. By confluence, there is then some Π'' where $\Pi \Longrightarrow_S^* \Pi''$ and $\Pi' \Longrightarrow_S^* \Pi''$. However, both Π and Π' are final, and hence equivalent, so that V is satisfactory. □

Theorem 10 (Deadlock Freedom). *If $\vdash C : \tau \triangleright \rho$, $\vdash C$ loc-ok, $\mathrm{NL}(\rho) \subseteq \Omega$, $[\![C]\!]_\Omega^\Uparrow = \Pi$, and $\Pi \Longrightarrow_S^* \Pi'$, then either Π' is a value for every location, or there is some Π'' such that $\Pi' \Longrightarrow_S \Pi''$.*

Proof. By Corollary 4, C either terminates or loops forever. First the case if C terminates, where the argument is similar to Theorem 4. If $\langle C, \Omega\rangle$ evaluates to $\langle V, \Omega'\rangle$, then by Completeness there is some $\Pi_V \succeq [\![V]\!]_{\Omega'}^\Uparrow$ such that $\Pi \Longrightarrow_S^* \Pi_V$. By Lemma 54, we can step $\Pi_V \Longrightarrow_S^* \Pi'_V$ where $\Pi'_V \succsim [\![V]\!]_{\Omega'}^\Uparrow$. By Confluence (Lemma 82), we can step $\Pi' \Longrightarrow_S^* \Pi'_V$. Then by Lemmas 53 and 56, Π'_V is a value. Lastly, there is either at least one step in the reduction sequence $\Pi' \Longrightarrow_S^* \Pi'_V$ satisfying the conclusion, or there are no steps, in which case Π' is itself a value, satisfying the theorem in either case.

Now the case when C loops forever. Suppose that the reduction sequence $\Pi \Longrightarrow_S^n \Pi'$ is of length n. Then we can find some C'' and Ω'' where $\langle C, \Omega\rangle \Longrightarrow_c^{1+n} \langle C'', \Omega''\rangle$. By Completeness (Theorem 9), we can step $\Pi \Longrightarrow_S^k \Pi''$ where $k \geq 1 + n$ and $[\![C'']\!]_{\Omega''}^\Uparrow \preceq \Pi''$. By Confluence (Lemma 82), Π'' can then make at least $k - n \geq 1 + n - n = 1$ steps, satisfying the theorem. □

Theorem 5 (Deadlock Freedom). *If $\vdash C : \tau \triangleright \rho$, every location literal in C is in Ω, and C contains no* kill-after *expressions, then whenever $[\![C]\!]_\Omega^\Uparrow \Longrightarrow_c^* \Pi$, either Π is final or it can step.*

Proof. Follows directly from Theorem 10. □