A New Architecture for Choreographic Programming Languages

A Dissertation Presented

by

Mako Bates

to

The Faculty of the Graduate College

of

The University of Vermont

In Partial Fullfillment of the Requirements
for the Degree of Doctor of Philosophy
Specializing in Computer Science

August, 2025

Defense Date: June 27, 2025
Dissertation Examination Committee:

Joseph P. Near, Ph.D., Advisor
Alice Patania, Ph.D., Chairperson
Christian Skalka, Ph.D.
Yuanyuan Feng, Ph.D.
Andrew K. Hirsch, Ph.D.
Holger Hoock, DPhil, Dean of the Graduate College

Abstract

Choreographic programming (CP) is a paradigm for implementing distributed systems that uses a single global program to define the actions and interactions of all participants. One characteristic of CP is that values are "located", *i.e.* associated or annotated with parties who own them, and non-owners of a located value cannot use it. Existing CP systems are generally either *select-&-merge* systems, which have a designated "select" operator for communicating knowledge of choice, or use alternative strategies that are known to be less efficient.

We make four contributions to the ongoing development of CP systems. First, we propose and formalize *conclaves* and *multiply-located values*; this combination of features enables efficient conditionals without redundant communication or a specialized operator. Second, we implement this "conclaves-&-MLVs" paradigm in Haskell as the MultiChor library. Third, we propose *census polymorphism*, a technique for abstracting over the number of participants in a choreography. Forth, we demonstrate the viability of a CP system MiniChor that uses only conclaves and has no designated syntax for located values.

MultiChor is available to end-users now, and contains solutions to key engineering problems. Based on anecdotal experiences using it and subsequent work on the theory-oriented fork MiniChor, we outline near-term avenues for future work on CP systems for industry use.

Citations

Material from this dissertation has been published in the following form:

Bates, M. and Kashiwa, S. and Jafri, S. and Shen, G. and Kuper, L. and Near, J. P.. (2025). Efficient, Portable, Census-Polymorphic Choreographic Programming. *PLDI25*.

Table of Contents

List of Figures

Chapter 1

Introduction and Literature Review

1.1 Introduction

Choreographic programming (CP) is a language paradigm for implementing distributed systems in which the programmer writes one unified program, called a choreography, that describes how the participants of the system interact from a third-person-omniscient perspective. (Carbone and Montesi 2013, Montesi 2014, Montesi 2023) A choreography can be translated into to a collection of executable programs for use in the real world, one for each participant; this process is called endpoint projection (EPP). The CP approach has benefits both for understandability of distributed system implementations, and for strong static guarantees about the deadlock-freedom of the resulting executable code (Carbone and Montesi 2013).

The study of CP is comparatively young; while some of the ideas have existed informally as far back as the 1970s, choreographic programming as it's understood today was first formalized in (Carbone and Montesi 2013). In this chapter we describe the central concepts of choreographic programming, its advantages and disadvantages, and past and ongoing work to push the boundaries of the kinds of systems it can implement.

1.1.1 Layout

Section 2.1 introduces *multiply-located values* (MLVs) and *conclaves*, fundamental ideas that enable everything that follows. These features combine to allow a compelling new strategy for KoC management as demonstrated in our formalism $\boldsymbol{\lambda_C}$ in Section 2.2. In particular, all well-typed $\boldsymbol{\lambda_C}$ choreographies are projectable and have

cromulent KoC by construction. In Section 2.3 we compare $\boldsymbol{\lambda_C}$ to representative systems that use other KoC management strategies.

Chapter 3 presents our implementation of the conclaves-&-MLVs paradigm in Haskell. The MultiChor library is already available on Hackage, Haskell's main package management system. MultiChor directly implements the main concepts of $\boldsymbol{\lambda_C}$ as a monadic eDSL which we describe in Section 3.2. Encoding the set-membership requirements in Haskell's Hindley–Milner-based type system presents particular chanleges which we discuss in Section 3.3. We also describes *census polymorphism*, a design pattern for choreographies that different CP systems may or may not support. MultiChor is the first CP implementation to support type-safe census polymorphism, by which we mean the ability to write choreographies that are parametric over their set of participants. Because MultiChor is fully embedded in and interoperable with Haskell, functional-programming patterns can be applied to the choreographic setting without further theoretical or infrastructural work. MultiChor's census polymorphic functions work by applying advanced type-level programming techniques native to modern Haskell to MultiChor's core API. We discuss census polymorphism in greater detail in Section 3.4. Section 3.5 presents a more involved case-study of what end-user MultiChor code might look like, and describes some qualitative challenges facing users of the current version.

Chapter 4 explores possibilities for the future development of MultiChor, in particular we show that located values are not a necessary primitive component of CP systems. We show this by developing MiniChor in Sections 4.1 to 4.3. MiniChor is a variant of MultiChor which can implement the same choreographies using only conclaves, communication, and local computation on normal ("naked") values. Section 4.4 relates the changes back to the subject of future developments in MultiChor.

Chapter 5 concludes this dissertation and describes promising avenues for future research.

1.1.2 Contributions

We enumerate four substantive contributions of this work.

Conclaves & MLVs In Chapter 2 we demonstrate a choreographic calculus $\boldsymbol{\lambda_C}$ that combines conclaves and multiply located values (MLVs) to enable a novel strategy for managing Knowledge of Choice (KoC). Prior works have featured structures similar to MLVs either in other computational contexts or as emergent patterns in choreographic systems, and prior works have also had conclaving behavior in their handling of

functions, but λ_C is the first work to combine them as built-in behaviors that enable KoC management. Prior strategies for KoC management required additional communication primitives and partial functions for EPP, or linearly-verbose conditional expressions and heavy predicate-transformers during static analysis.

An efficient library-level implementation Our Haskell library, MultiChor, is available now on Hackage. It implements the conclaves-&-MLVs CP paradigm as an eDSL in Haskell. Many existing systems for CP have required their own compilers (for stand-alone languages), custom compiler extensions to augment the syntax of a host language, or particular capabilities of a host language's macro system. A notable exception, and the direct ancestor of MultiChor in engineering terms, is HasChor (Shen et al. 2023). Like HasChor, MultiChor runs entirely within Haskell using the off-the-shelf GHC compiler. Unlike HasChor, MultiChor does not add spurious communication to choreographies as part of its KoC strategy, so it can implement many protocols more efficiently. In Chapter 3 we discuss the engineering challenges involved, our solutions to them, some example programs, and the qualitative experience of writing software in MultiChor.

Census polymorphism Although it is common to describe concurrent protocols parametrically with respect to the quantities of participants, we know of only one prior system capable of actually expressing choreographies in that way: an un-implemented extension to Procedural Choreographies described abstractly in (Cruz-Filipe and Montesi 2016). In Section 3.4 we discuss this property in greater detail, describing in general what a system must provide to make census polymorphism actually useful, and how specifically this can be implemented as a type-safe layer on top of the core MultiChor API. The principal motivation for census polymorphism is that choreographic code doesn't need to be updated every time the participant list changes. A variety of non-choreographic systems already support analogous capabilities (Le Brun et al. 2025, Weisenburger et al. 2018, Jongmans 2025, Rastogi et al. 2014, Sweet et al. 2023) in their respective contexts.

Choreographies without located values We demonstration by example that (multiply or singly) located values need not be a primitive concept in choreographic programming. Specifically, Chapter 4 describes a transformation of MultiChor into MiniChor, an equally expressive CP system in which MLVs are an emergent special case of conclaved computation. Compared against the λ_C formalism, this shows that in an appropriate computational context and type system the conclaves-&-MLVs paradigm can be collapsed into a

just conclaves paradigm. As discussed in Section 4.4, other CP systems feature analogous dualities. To the best of our knowledge MiniChor is the first CP system with a single mode of ownership.

1.1.3 Basics of Choreographic Programming

Characteristic features of CP are

- the use of a single program to represent the behavior of multiple concurrent parties or threads,
- the freedom to sequence actions by various parties without preferring any one party's perspective over others, and
- a unified communication operator typically written $\rightsquigarrow$, `~>`, `comm`, or `com`.

A unified communication expression might look like `Alice{"hello"} ~> Bob`; this represents both Alice's implementation `send "hello" to Bob` and Bob's implementation `receive String from Alice`. Endpoint projection is the process of automatically deriving these "local" implementations from a choreography; see Section 1.2.2. Traditional computations (possibly annotated with locations) and choreographic expressions like `comm` can be sequenced and composed to make more involved choreographies. For example, we could write a simple call-and-response choreography like

$$
\begin{aligned}
&x := \mathsf{Client}\{\texttt{"Hello, my name is "} +\!\!+ \mathit{input}()\} \rightsquigarrow \mathsf{Server};\\
&\mathsf{Server}\{\mathit{log}(x)\};\\
&\_ := \mathsf{Server}\{\texttt{"Hi "} +\!\!+ \mathit{last}(\mathit{words}(x))\} \rightsquigarrow \mathsf{Client};
\end{aligned}
$$

to represent both a client's behavior,

$$
\begin{aligned}
&\mathsf{send}\ \texttt{"Hello, my name is "} +\!\!+ \mathit{input}()\ \mathsf{to\ Server};\\
&\_ := \mathsf{receive\ String\ from\ Server};
\end{aligned}
$$

and the corresponding server's behavior:

$$
\begin{aligned}
&x := \mathsf{receive\ String\ from\ Client};\\
&\mathit{log}(x);\\
&\mathsf{send}\ \texttt{"Hi "} +\!\!+ \mathit{last}(\mathit{words}(x))\ \mathsf{to\ Client};
\end{aligned}
$$

Compare the above pseudo-code with the MultiChor code in Figure 1.1.

```
1  callNResponse :: Choreo '["Client", "Server"] (CLI IO) ()
2  callNResponse = do
3      name <- _locally client (getInput @String "Enter your name:")
4      x <- (client, name) ~> server @@ nobody
5      locally_ server \un -> putOutput "Received message:" (un singleton x)
6      response <- purely server \un -> "Hi " ++ last(words(un singleton x))
7      _ <- (server, response) ~> client @@ nobody
8      return ()
9    where client :: Member "Client" '["Client", "Server"] = explicitMember
10         server  :: Member "Server"  '["Client", "Server"] = explicitMember
11
12 clientBehavior :: Network (CLI IO) ()
13 clientBehavior = do
14     name <- run (getInput @String "Enter your name:")
15     send name ["Server"]
16     _ <- recv @String "Server"
17     return ()
18
19 serverBehavior :: Network (CLI IO) ()
20 serverBehavior = do
21     x <- recv @String "Client"
22     run (putOutput "Received message:" (unwrap singleton x))
23     let response = ("Hi " ++ last(words(unwrap singleton x)))
24     send response ["Client"]
```

The choreography `callNResponse` is defined on lines 2–8; its type is declared on line 1. In MultiChor local computations must be sequenced with, rather than freely composed with, communication operations, *e.g.* on line 3 `name` is first computed (by the side-effect-full `getInput`) before it can be sent as a message on line 4. MultiChor does contain helper functions that would allow a user to express `callNResponse` in a natural three-line format, but we neglect to use them here in the interest of simplicity. Endpoint projection of `callNResponse` to `"Client"` would result in a first-person procedure like `clientBehavior` (line 13), and `"Server"` would similarly get `serverBehavior` (line 20). Those two definitions are somewhat simplified compared to the actual `Network` values MultiChor manipulates at runtime; the structure of the modules makes `unwrap` (lines 22 and 23) inaccessible to users, and in practice no human should ever need to inspect a `Network` value generated by `epp`.

Figure 1.1: A Simple Choreography and its Projections: a call & response.

1.1.4 A More Involved Example

To motivate choreographic programming, consider the three (non-choreographic, single-thread) programs in Figure 1.2, which are intended to run concurrently and pass messages back and forth between each other. The overall effect is a protocol in which the client makes a `Get` or `Put` request to a server (with a backup) that manages a key-value-store (KVS). Even this simplified example takes a moment for a reader to make sense

of; one must read the three programs, infer the correspondence between messages sent and received by the three parties, and judge for oneself if the communication protocol implemented is sensical. One might even judge that this simple protocol has a bug: if the request is a `Get`, the backup server will hang indefinitely!

```
kvs_client :: Request -> IO Response
kvs_clien = request = do
    request `send` primary     -- send request to the primary node
    response <- recv primary   -- receive the response
    return response
```

(a) The function to be called by the client process. They pass in their `Request` object and send it to the server. Then they receive a response from the server and return it.

```
-- handle a Get or Put request
handleRequest :: Request -> IORef State -> IO Response

kvs_primary :: IORef State -> IO ()
kvs_primary stateRef = do
    request <- recv client                      -- receive the request
    case request of                             -- branch on the request
      Get _ -> pure ()                          -- no-op
      Put _ _ -> do request `send` backup       -- send request to the backup node
                    ack <- recv backup          -- and get back an acknowlegement
                    pure ()
    response <- handleRequest request stateRef  -- process the request locally
    response `send` client                      -- send response to client
```

(b) The function to be called by the primary server. They pass in a reference to their mutable state, and receive a message of type `Request` from the client. In the case of a `Put` request, they forward it to the backup server and check for the backup's acknowledgement. In either case, they process the request against their own mutable state and send the response back to the client.

```
kvs_backup :: IORef State -> IO ()
kvs_backup stateRef = do
    request@(Put _) <- recv primary  -- receive the request
    success <- handleRequest request stateRef  -- process the request locally
    success `send` client  -- acknowlege to the primary server that we're done
```

(c) The function to be called by the backup server. They pass in a reference to their mutable state, and receive a `Put` message from the primary server. They process it against their mutable state and send back an acknowledgement message to indicate their success.

Figure 1.2: A Simple Concurrent Protocol: a key-value store with a backup server

In Section 1.2.1 we will mention some other techniques that have been used to facilitate writing large and complicated concurrent protocols, but here we jump directly to choreographic programming (CP). Figure 1.3 shows the same protocol as Figure 1.2, but implemented as a choreography. In this form there is no cognitive

overhead for matching `send` and `recv` operations, because matching pairs of them are combined into monolithic `comm` operations. The entire protocol can be read at once in a sensical order. (The order in which operations are presented in a choreography is not necessarily the order in which they will happen; the participants are not guaranteed to all start at the same physical time, or to operate at the same speeds.) Re-writing the example KVS system as a choreography does not immediately solve the issue of what the backup server should do in the event of a `Get` request, but it makes the problem detectable by static analysis. In fact, the choreography in Figure 1.3 cannot compile in any real CP system because `"backup"` 's behavior is ambiguous! Figure 1.4 shows two variations of how to realize the KVS behavior in Haskell using our MultiChor library.

```
kvs :: Located '["client"] Request ->
       Located '["primary"] (IORef State) ->
       Located '["backup"] (IORef State) ->
       Choreo Participants IO (Located '["client"] Response)
kvs request primaryStateRef backupStateRef = do
  request' <- (client, request) `comm` primary  -- send request to the primary node
  case request' of                              -- branch on the request
    Get _ -> pure ()
    Put _ _ -> do
      request'' <- (primary, request') `comm` backup  -- forward request to backup
      success <- backup `locally`                     -- the backup does local work:
                   (handleRequest <$> request'' <*> backupStateRef)
      ack <- (backup, success) `comm` primary
      pure ()
  response <- primary `locally`               -- the primary server does local work:
                (handleRequest <$> request' <*> primaryStateRef)
  result <- (primary, response) `comm` client @@ nobody  -- send response to client
  return result
```

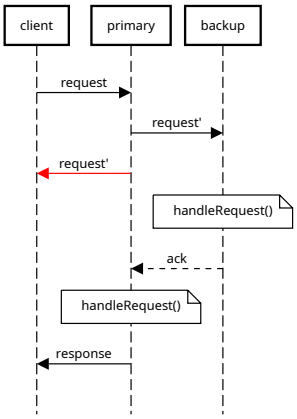
This pseudo-code choreography implements the protocol from Figure 1.2 as a single program. As written, it is not actually realizable because `backup` doesn't know whether to expect a message or not. Real CP systems have ways of detecting and fixing problems like this.

Figure 1.3: A (Less) Simple Choreography: a key-value store with a backup server

1.2 Background

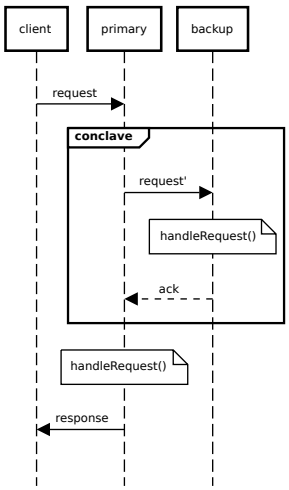
Choreographic programming is a paradigm that expresses a distributed system as a single, global program describing the behavior and interactions of all parties. The global view of the distributed system enables easier reasoning about the system's behavior; for example choreographic languages can ensure *deadlock*

```
1  -- sub-choreography: backup may or may not do anything...
2  handleBackup :: Located '["backup"] (IORef State)
3               -> Request
4               -> Choreo Participants IO ()
5  handleBackup state   (Get key)       = pure () -- no-op.
6  handleBackup state r@(Put key value) = do
7      success <- backup `locally` \un -> -- the backup's local work
8        handleRequest r (un backup state)
9      ack <- (backup, success) `comm` primary  -- send acknowlegement
10     pure ()
11
12 kvs :: Located '["client"] Request
13     -> Located '["primary"] (IORef State),
14     -> Located '["backup"] (IORef State)
15     -> Choreo Participants IO (Located '["client"] Response)
16 kvs request primaryStateRef backupStateRef = do
17   -- send request to the primary node:
18   request' <- (client, request) ~> primary @@ nobody
19   -- branch on the request:
20   broadcast (primary, request') >>= (handleBackup backupStateRef)
21   -- process request on the primary node:
22   response <- primary `locally` \un ->
23     handleRequest (un primary request') (un primary primaryStateRef)
24   -- send response to client:
25   (primary, response) ~> client @@ nobody
```



(a) A key-value store with a backup server, written in MultiChor. The backup server sends an acknowledgement message ack to the primary server if and only if `request` is a `Put`. The `broadcast` operator (line 19) ensures KoC so that the primary and backup servers are guaranteed to use the same case of `handleBackup`, but it results in redundant communication (shown in red in the sequence diagram).

```
1  type Servers = ["primary", "backup"]
2  servers :: Subset Servers Participants
3  servers = primary @@ backup @@ nobody
4
5  -- Change to handleBackup: now a sub-choreography for *servers only*
6  handleBackup :: Located '["backup"] (IORef State)
7               -> Request
8               -> Choreo Servers IO ()
9
10 -- Change to line 20: sub-choreography is *conclaved*
11 conclave servers $
12     broadcast (primary, request') >>= (handleBackup backupStateRef)
```



(b) In this variation, the `conclave` operator eliminates the redundant communication. The conclaved sub-choreography is indicated by a box in the sequence diagram. On line 3, `@@ nobody` is MultiChor idiom explained in Section 3.3.

Figure 1.4: Real Choreographies: a key-value store writing using MultiChor, two variations.

freedom (Carbone and Montesi 2013) and choreographies can be composed modularly like normal single-threaded protocols.

1.2.1 History and Adjacent Domains

Theoretical and pragmatic frameworks for thinking clearly about concurrent and communicating systems are many and varied, and include well known examples like Petri Nets that are over sixty years old (Petri 1962). A number of other systems from the following decades are important both as historical markers and because modern research builds directly on their ideas, their theory, structural patterns, and terminology. The actor model (Hewitt et al. 1973) posits the description of an "actor" and their behavior as the fundamental unit of a concurrent program, while being deliberately abstract about what an actor actually is. Process- or π-calculi are generally characterized by a designated syntax for parallelism (actions that may happen in any order or simultaneously), and typically have names or addresses associated with the participating processes and/or channels of communication between processes. (Baeten 2004) gives a good history of the field as it developed and diversified through the end of the 20^{th} century.

A basic goal in computer science is to provide guarantees that a program *can't* exbibit a behavior that has been deemed bad, *e.g.* run-time exceptions or deadlocking; a common way to accomplish this is by applying a type checker to the program. One successful system (or style of system) for typing concurrent protocols has been *multiparty session types* (MPST) (Honda et al. 2008). A user encodes a protocol of (possible sequences of) communication in a *global type* in the language of MPST. Projection of that global type to a named party ("endpoint") in the type yields a *local type* that can be ascribed to (and enforced against) an implementation (*e.g.* in a π-calculus) of that process's behavior[1]. In addition to the simple "what kind of thing might `x` be" sense of type safety, MPST also provided communication safety (*e.g.* a party never binds a received message to the wrong variable at runtime) and a form of deadlock freedom.

Research extending MPST's expressiveness (the space of desirable programs that can be judged well-formed) and safety (the space of undesired programs that can be statically rejected) is ongoing. For example (Stutz and D'Osualdo 2025) describe Automata-based Multiparty Protocols, a MPST-like framework that, with various caveats, is more expressive than MPST and in which endpoint projection is total. (Le Brun et al. 2025), presented at the same conference, directly extend the syntax and semantics of MPST with

[1] As in many CP systems, the endpoint projection function in MPST is incomplete.

first-class process names and open-ended receiving channels; in addition to the utility of these features *per se*, this lets their system model mutual-exclusion and race conditions. One way to think about choreographic programming is as the extension of the core ideas of MPST to include *both* the communication plan and the implementation of the computations each party is doing.

Although in this present work we use the noun "choreography" to refer to actual programs written in CP systems, the word was already in use before the invention of choreographic programming *per se*. The broader sense of the word is any unified 3rd-person description of or plan for interaction between two or more participating systems. Pseudo-code diagrams featuring a unified communication operator "→" were being used to describe cryptographic protocols at least as early as 1978 (Needham and Schroeder 1978). As an example, (W3C 2005) presented the *"Web Services Choreography Description Language"* in which a user could specify the interaction, the sequence of communications, between parties. Like MPST (introduced a few years later), this was a specification language; implementations compatible with a given specification needed to be written separately.

Another technique for multiparty programming (broadly, any technique for representing the behavior of a system of concurrent participants in as a single program) is *multitier programming*. A multitier program has a "first person" perspective that privileges one party or role; this perspective shifts to other roles, with implicit or explicit communication mediating these shifts. ScalaLoci is an example of a multitier language with an embedded implementation in Scala and a formalism demonstrating its soundness (Weisenburger et al. 2018). A similar computational model named "first person choreographies", implemented as a Rascal proof-of-concept named 1CPLT, omits ScalaLoci's reactive programming aspects (Jongmans 2025). Of particular interest to our current work, both these languages can represent applications with variable quantities of participants. The network behavior of a system built in these languages is less straightforward than that of a similar-looking choreography in two ways: critical contextual information (roughly corresponding to Knowledge of Choice) is communicated implicitly, and the exact communications that must or could happen (which depend on the variable quantities of participants) are determined dynamically. Nonetheless, (Giallorenzo et al. 2021) were able to show a degree of equivalence between minimalist models of multitier and choreographic programming.

1.2.2 Endpoint Projection

CP systems necessarily include a means by which a given computer can execute the behavior of a role in the choreography. Typically, this takes the form of a function from choreographies to programs in a "local" language which the target computer can execute. Such a function is parameterized by the target role, and is called *Endpoint Projection* (EPP); the roles are "endpoints", *i.e.* surfaces from which and into which messages pass, and the choreography is "projected" in the sense that the given endpoint's view of it is extracted.

As an example, EPP of the choreography in Figure 1.4*(b)* to each of the three participants would yield respective programs very similar to the ones in Figure 1.2, except that the primary server would always send the request to the backup server (and therefor the backup server would know how to proceed).

In a choreographic program, many (in some systems, all) values will be *located*: they will have explicit or implicit metadata indicating their owner. EPP of a located value to its owner results in the value itself (typically with the ownership annotation removed), and EPP to anyone else results in a special placeholder symbol, *e.g.* $\perp$. The appearance of $\perp$ in a party's projection is not an error, but attempting to do any semantic evaluation on $\perp$ would be, so an important correctness property for choreographies is that parties never do that.

1.2.3 Knowledge of Choice

Choreographies with conditionals (`if`-expressions or anything that could be used for conditional control-flow) introduce a challenge for endpoint projection: *some parties might not know which branch to take!* This challenge is referred to as the *knowledge of choice* (KoC) (Castagna et al. 2011) problem. All choreographic programming languages include a strategy for KoC that ensures that relevant parties have enough information to play their part in the program.

The pseudo-code in Figure 1.3 shows a simple instance of this exact problem: `backup` doesn't know whether to expect a message or not, because that decision depends on a value (`request'`) that `backup` doesn't have. Figure 1.4*(a)* implements the KoC strategy used by HasChor (Shen et al. 2023): the branch guard is broadcast to everyone. HasChor's authors knew this to be an inefficient (but expedient) solution. As we show in Figure 1.4*(b)*, MultiChor can do better.

A more traditional approach, which we refer to as *select-&-merge*, is to include in one's language a special operator just for communicating KoC. This operator is called "`select`"; it sends a statically-declared flag to the recipient to select which of that party's possible behaviors should be activated. For example, Figure 1.5 shows how our KVS choreography might be expressed in a select-&-merge system. Under both the centralized semantics and type analysis, `select` is a no-op! During EPP, `select` results in "offer" and "choose" actions at the receiver and sender, respectively. Because `select` doesn't affect typing, in Figure 1.5 it would not be possible to encode the need for the lines 9 and 12 in Haskell's type system. Instead, such CP systems enforce KoC requirements in their "merge" operator, which is applied during EPP. Any party besides the owner of a branch guard will replace the entire conditional expression with the merge of their projections of the branches. The merge of two "offer" expressions is an offer of the union of the possible continuations; merges of any other combinations of expressions are only defined when the two expressions are the same. Thus, if lines 9 and 12 were omitted from Figure 1.5, the error would be detected during EPP to `backup` when the system tried to merge `{}` with

```
{ request'' <- recv primary;
  success <- handleRequest request'' backupStateRef;
  send success primary  }
```

A substantial body of research has explored the soundness select-&-merge and its fundamentals, implementation, and extensions (Carbone and Montesi 2013, Cruz-Filipe and Montesi 2020, Giallorenzo et al. 2024, Montesi and Peressotti 2017). That said, HasChor, MultiChor, ChoRus, and ChoreoTS all do EPP at runtime(Bates et al. 2025), so using EPP to detect KoC problems would not be a satisfactory solution for them.

Yet another KoC strategy was proposed by (Jongmans and van den Bos 2022). Their approach requires writing distinct guards for every participant in a conditional expression; they show how to use predicate transformers to check that such distributed decisions are unanimous. This system is verbose, but at least as expressive as the one we describe in this work.

1.2.4 The "Census" typing context

Although it is a hallmark of CP that a user may write actions for various parties in any given place in the program without demarcations of who "control" is passing to, it is not necessarily the case that every party

```
kvs :: Located '["client"] Request ->
       Located '["primary"] (IORef State) ->
       Located '["backup"] (IORef State) ->
       Choreo Participants IO (Located '["client"] Response)
kvs request primaryStateRef backupStateRef = do
  request' <- (client, request) `comm` primary  -- send request to the primary node
  case request' of                              -- branch on the request
    Get _ -> do
      select primary backup LEFT
      pure ()
    Put _ _ -> do
      select primary backup RIGHT
      request'' <- (primary, request') `comm` backup  -- forward request to backup
      success <- backup `locally`                     -- the backup does local work:
                   (handleRequest <$> request'' <*> backupStateRef)
      ack <- (backup, success) `comm` primary
      pure ()
  response <- primary `locally`               -- the primary server does local work:
                (handleRequest <$> request' <*> primaryStateRef)
  result <- (primary, response) `comm` client @@ nobody  -- send response to client
  return result
```

This pseudo-code choreography implements the protocol from Figure 1.4*(b)* using "select-&-merge" syntax. Attempting to engineer a real select-&-merge CP eDSL in the style of MultiChor would result in a system that could not detect KoC errors until runtime.

Figure 1.5: A Select-&-Merge Choreography: a key-value store with a backup server

that exists is eligible to take action at every place in the choreography. Some earlier works, *e.g.* (Cruz-Filipe et al. 2022), have tracked these sets of participants in their type systems, and used that typing context to control participation inside of function bodies. Such a typing context plays a more active role in this present work, so we coin the term *"census"* for a typing context that controls which parties are "present" to participate in any given part of a choreography. A party not listed in a census typically should not evaluate that section of the choreography; exactly how that's enforced or implemented will depend on the system in question.

1.2.5 Additional Literature

Research and development of CP systems seems to have accelerated since approximately 2022. Pirouette (Hirsch and Garg 2022), Chorλ (Cruz-Filipe et al. 2022), and PolyChorλ (Graversen et al. 2024) are (higher order) functional languages for writing select-&-merge choreographies; PolChorλ additionally introduces polymorphism over identities of parties. Choral (Giallorenzo et al. 2024) is a choreographic language implementing the select-&-merge paradigm and targeting industrial use; it runs on the JVM and can easily import local Java code. Dyno (Zakhour et al. 2023) is a CP system for Android development; it's key feature

is the ability to resolve the location and ownership of data and computation dynamically, while still providing static safety guarantees.

Excitingly, CP libraries for a variety of general purpose languages have recently appeared on the scene. These include UniChorn (Chakraborty 2024), Chorex (Wiersdorf and Greenman 2024), and Klor (Lugović and Jongmans 2024). All three are under development; here we report on their state toward the end of 2024. UniChorn is a port of HasChor into the Unison programming language. To implement EPP, it uses the Unison feature of *abilities*, better known in the literature as algebraic effects (Plotkin and Power 2003, Plotkin and Pretnar 2013). This implementation approach, which was also recently proposed by (Shen and Kuper 2024), can be thought of as a generalization of the free(r)-monad approach. As a direct port of HasChor, UniChorn does not support conclaves, MLVs, or census polymorphism. Chorex is a CP system for Elixir, and Klor is a CP system for Clojure; both Chorex and Klor leverage the powerful macro systems of their respective host languages to carry out EPP. Chorex uses the select-&-merge KoC strategy, and unprojectable choreographies can be detected at macro expansion time. Klor, on the other hand, is effectively an conclaves-&-MLVs system, but their API differs from the implementations we present here, and the authors have not yet shown what safety guarantees it does or does not offer. Neither Chorex nor Klor support census polymorphism.

Wysteria (Rastogi et al. 2014) and Symphony (Sweet et al. 2023) are domain-specific languages for *secure multiparty computation*. Programs in these languages can exhibit census polymorphism, but they have homomorphic encryption baked into their semantics for communication, and they are not intended for general-purpose choreographic programming. Wysteria has a `par` language construct, used for evaluating an expression at a set of locations, that is somewhat similar in spirit to our conclaves. However, applying the conclave concept to choreographic programming, and to the choreographic knowledge-of-choice problem in particular, is to our knowledge a novel contribution of this paper. Symphony does not support conditionals, and therefore KoC propagation is a moot point for them.

Another possible antecedent to census polymorphism as described in Section 3.4 is (Cruz-Filipe and Montesi 2016), which extends the syntax of Procedural Choreographies (PC) (Cruz-Filipe and Montesi 2017) to support lists of processes as arguments to procedure calls. Exact comparison between the extended PC system and MultiChor is difficult for a few reasons: First, PC is an advanced select-&-merge system which tracks mutable network topology instead of a census. Second, in PC's computational model processes serve double-duty as "participants" who do things and as "variables" that store data; this is a sensible choice

for their target usage context: parallelized object-oriented programming in which communication is cheap. Understanding the limits of their syntax extensions in the context of their computational model is difficult because the commensurate typing rule extensions not explicit. Third, PC does not have an implementation. We do not assert here whether either system's expressivity subsumes that of the other.

Bibliography

Baeten, J. (2004). *A brief history of process algebra*. Computer science reports. Technische Universiteit Eindhoven.

Bates, M., S. Kashiwa, S. Jafri, G. Shen, L. Kuper, and J. P. Near (2025, June). Efficient, portable, census-polymorphic choreographic programming. *Proc. ACM Program. Lang. 9*(PLDI). Archive: `https://arxiv.org/abs/2412.02107`.

Carbone, M. and F. Montesi (2013). Deadlock-freedom-by-design: multiparty asynchronous global programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, New York, NY, USA, pp. 263–274. Association for Computing Machinery.

Castagna, G., M. Dezani-Ciancaglini, and L. Padovani (2011). On global types and multi-party sessions. In R. Bruni and J. Dingel (Eds.), *Formal Techniques for Distributed Systems*, Berlin, Heidelberg, pp. 1–28. Springer Berlin Heidelberg.

Chakraborty, K. (2024). Unichorn. `https://share.unison-lang.org/@kaychaks/unichorn/`.

Cruz-Filipe, L., E. Graversen, L. Lugović, F. Montesi, and M. Peressotti (2022, September). *Theoretical Aspects of Computing*, Volume 13572 of *Lecture Notes in Computer Science*, Chapter Functional choreographic programming, pp. 212–237. Tbilisi, Georgia: Springer. Archive: `https://arxiv.org/abs/2111.03701`.

Cruz-Filipe, L. and F. Montesi (2016). Choreographies in practice. In E. Albert and I. Lanese (Eds.), *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, Volume 9688 of *Lecture Notes in Computer Science*, pp. 114–123. Springer.

Cruz-Filipe, L. and F. Montesi (2017). Procedural choreographic programming. In A. Bouajjani and A. Silva (Eds.), *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, Volume 10321 of *Lecture Notes in Computer Science*, pp. 92–107. Springer.

Cruz-Filipe, L. and F. Montesi (2020). A core model for choreographic programming. *Theor. Comput. Sci. 802*, 38–66.

Giallorenzo, S., F. Montesi, and M. Peressotti (2024, jan). Choral: Object-oriented choreographic programming. *ACM Trans. Program. Lang. Syst. 46*(1).

Giallorenzo, S., F. Montesi, M. Peressotti, D. Richter, G. Salvaneschi, and P. Weisenburger (2021). Multiparty languages: The choreographic and multitier cases (pearl). In A. Møller and M. Sridharan (Eds.), *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, Volume 194 of *LIPIcs*, pp. 22:1–22:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Graversen, E., A. K. Hirsch, and F. Montesi (2024). Alice or bob?: Process polymorphism in choreographies. *Journal of Functional Programming 34*, e1.

Hewitt, C., P. Bishop, and R. Steiger (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, San Francisco, CA, USA, pp. 235–245. Morgan Kaufmann Publishers Inc.

Hirsch, A. K. and D. Garg (2022, January). Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang. 6*(POPL).

Honda, K., N. Yoshida, and M. Carbone (2008). Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, New York, NY, USA, pp. 273–284. Association for Computing Machinery.

Jongmans, S.-S. (2025). First-person choreographic programming with continuation-passing communications. In V. Vafeiadis (Ed.), *Programming Languages and Systems*, Volume 2 of *Lecture Notes in Computer Science*, pp. 62–90. Springer.

Jongmans, S.-S. and P. van den Bos (2022). *A Predicate Transformer for Choreographies (Full Version)*. Number 01 in OUNL-CS (Technical Reports). Open Universiteit Nederland.

Le Brun, M. A., S. Fowler, and O. Dardha (2025). Multiparty session types with a bang! In V. Vafeiadis (Ed.), *Programming Languages and Systems*, Cham, pp. 125–153. Springer Nature Switzerland.

Lugović, L. and S.-S. Jongmans (2024). Klor: Choreographies in clojure. `https://github.com/lovrosdu/klor`.

Montesi, F. (2014). *Choreographic Programming*. Ph. D. thesis, Denmark.

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

Montesi, F. and M. Peressotti (2017). Choreographies meet communication failures. *CoRR abs/1712.05465*.

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

Petri, C. A. (1962). *Kommunikation mit Automaten*. Ph. D. thesis, Bonn. 128 Seiten.

Plotkin, G. and J. Power (2003). Algebraic operations and generic effects. *Applied categorical structures 11*, 69–94.

Plotkin, G. D. and M. Pretnar (2013, December). Handling algebraic effects. *Logical Methods in Computer Science Volume 9, Issue 4*.

Rastogi, A., M. A. Hammer, and M. Hicks (2014). Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy*, pp. 655–670.

Shen, G., S. Kashiwa, and L. Kuper (2023, aug). Haschor: Functional choreographic programming for all (functional pearl). *Proc. ACM Program. Lang. 7*(ICFP).

Shen, G. and L. Kuper (2024). Toward verified library-level choreographic programming with algebraic effects. `https://arxiv.org/abs/2407.06509`.

Stutz, F. and E. D'Osualdo (2025). An automata-theoretic basis for specification and type checking of multiparty protocols. In V. Vafeiadis (Ed.), *Programming Languages and Systems*, Cham, pp. 314–346. Springer Nature Switzerland.

Sweet, I., D. Darais, D. Heath, W. Harris, R. Estes, and M. Hicks (2023, February). Symphony: Expressive secure multiparty computation with coordination. *The Art, Science, and Engineering of Programming 7*(3).

W3C (2005). WS Choreography Description Language. `http://www.w3.org/TR/ws-cdl-10/`.

Weisenburger, P., M. Köhler, and G. Salvaneschi (2018, October). Distributed system development with scalaloci. *Proc. ACM Program. Lang. 2*(OOPSLA).

Wiersdorf, A. and B. Greenman (2024). Chorex: Choreographic programming in elixir. `https://github.com/utahplt/chorex`.

Zakhour, G., P. Weisenburger, and G. Salvaneschi (2023, October). Type-safe dynamic placement with first-class placed values. *Proc. ACM Program. Lang. 7*(OOPSLA2).

Chapter 2

A New Core Choreographic Calculus

2.1 Introduction

In this chapter we introduce and formalize the *conclaves-&-MLVs* paradigm for choreographic programming. The motivation is to match both the communication efficiency of select-&-merge systems like Chorλ and the portability and simplicity of broadcast-based systems like HasChor.

The HasChor library is inefficient in two ways. The first is particular to its broadcast-based KoC strategy: in order for any parities to branch on a value, the value must be broadcast to *all* parties. This can be overcome by adding censuses and *conclaves* to the system. The second shortcoming is shared in common with many select-&-merge systems like Pirouette: Sequential conditional clauses on the same branch-guard (*aka* "scrutinee") require KoC to be *re*communicated. We address this issue by extending the notion of located values into *multiply located values* (MLVs). MLVs allow multiple parties to branch together on a shared guard; in addition to recyclability, this alleviates the need for a designated `select` operator. In this section we present $\boldsymbol{\lambda_C}$, a formal model of a higher-order functional CP language that uses conclaves, census tracking, and MLVs to guarantee proper KoC management entirely by type-checking and without compromising efficiency or expressivity.

2.1.1 Multiply-located values

Previous choreography languages have featured *located values*, values annotated with (or implicitly assigned to) their owning party such that EPP to the owner results in the value itself and EPP to any other party results in a special "missing" value (*e.g.* $\bot$). *Multiply located values* are exactly the same except they are annotated with a non-empty *set* of parties. EPP of a multiply-located value for any of the owning parties results in the same value, and projection to any other party yields $\bot$. Prior works have objects with multiple owners as emergent structures in a language (*e.g.* choreographic processes (Giallorenzo et al. 2024), distributed choice types (Cruz-Filipe et al. 2023)), but these project to each owner's distinct view of the structure.

Creation of an MLV within an choreographic runtime follows from the fact that if Alice sends Bob a number, both Alice and Bob know what number was sent. Representing this in the language can be done a few different ways:

- A `share` operator that updates the type of an MLV-typed variable to include the recipient(s) in the ownership set. This is a poor fit for a functional programming language because it mutates the type of a variable, and additional machinery would be needed to make it work with nested conclaves (see Section 2.1.2).
- A `comm` operator that returns a value owned by the original owners and the recipient(s). This is not as straightforward as it sounds; if the communication happens inside a conditional, some of the original owners may not know that the communication happened. Intersecting with the current census is a sound solution, but may be difficult to embed in the type system of existing host languages.
- A `multicast` operator (by whatever name) that returns a value owned by exactly the specified recipients. In practice, users will often list the sender (and possibly any subset of the original owners) among the recipients; an ideal implementation would omit the actual communication to recipients who already have the data.
- A `broadcast` operator that always returns an MLV owned by the entire current census. This is actually equivalent to the above `multicast` operator; the choice is purely ergonomic.

Multiply-located values can also enable concise expression of programs in which multiple parties compute the same thing in parallel, a common occurrence when communication is more expensive than computation.

For example, the λ_C expression $5@\{p,q,r\} + 3@\{p,q,r\}$ represents an addition performed in parallel by p, q, and r.

2.1.2 Managing KoC with Conclaves and MLVs

As discussed in Section 1.2.4, prior systems have tracked censuses as attributes of choreographic functions. It's not usually required that every member of a census actually do anything in the choreography in question, so (intuitively) if a choreography has some given census **C**, then it should be possible to embed that choreography inside a larger one with census **D**, provided $\mathbf{C} \subseteq \mathbf{D}$. We call such sub-choreographies with sub-censuses *conclaves*. Any system with censuses likely has some version of conclaves, but in prior systems like Chorλ they were implicit side-effects of function application; they did not affect the KoC strategy (Cruz-Filipe et al. 2022). The ChoRus system introduced a designated conclave operator to explicitly limit the census within its argument, and showed how this could be used to avoid HasChor's overly broad `broadcasts` [1]. An `conclave` operator is a good design choice for an embedded DSL, but is not necessary in an abstract or bespoke language where the action of conclaving can be built into relevant constructs like functions and conditionals.

The combination of censuses with conclaving and MLVs constitutes a novel KoC strategy, on par with state-of-the-art select-&-merge systems like Chorλ in terms of communication efficiency and more amenable to implementation as an eDSL. The specific strategy is to require conditionals (control-flow branches) to conclave to the owners of their guard value. This can be accomplished by broadcasting inside the conclave, or by passing in a MLV as the guard. The advantage of using MLVs (aside from generally making things more concise) is that they let conclaves return shared knowledge to for reuse in later conditionals.

2.2 A Formal Conclaves-&-MLVs Language

We present the λ_C CP system. The syntax of λ_C and our overarching computational model and proof-approach are loosely based on Chorλ (Cruz-Filipe et al. 2022). λ_C is a higher-order choreographic lambda calculus; we omit recursion and polymorphism because they are orthogonal to our goals here. Specifically, we will show

[1] As discussed in (Bates et al. 2025), ChoRus has since been upgraded to incorporate the innovations discussed in this work. (Kashiwa et al. 2023) is an unpublished pre-print describing their system as it existed without MLVs or census polymorphism. It uses the now-deprecated term *"enclave"* instead of *"conclave"*.

that multiply-located values and conclaving operations are sufficient for a sound CP language without further KoC management. In Sections 2.2.1 to 2.2.5 we describe the syntax, type system, and semantics of λ_C. As in other choreographic languages, the primary semantics describes the intended *meaning* of choreographies and can be used to reason about their behavior, but is not the "ground truth" of concurrent execution. Sections 2.2.6 to 2.2.8 describe the languages of distributed processes, λ_L and λ_N, and define endpoint projection for λ_C. In Section 2.2.9, we prove that the behavior of a choreography's projection in λ_N matches that of the original λ_C choreography, and that λ_C's type system ensures deadlock-freedom. In Section 2.3 we provide some example choreographies in (a plain-text rendering of) λ_C. For example, Figure 2.11 implements the KVS example from Section 1.1.

2.2.1 Syntax

The syntax of λ_C is in Figure 2.1. Location information sufficient for typing, semantics, and EPP is explicit in the expression forms. We distinguish between "pairs" ($\mathsf{Pair}\, V_1 V_2$, of type $(d_1 \times d_2)@p^+$) and "tuples" ((V_1, V_2), of type (T_1, T_2)) so that we can have a distinguishable concept of "data" as "stuff that can be sent"; we do not believe this to have any theoretic significance.

The superscript-marked identifier p^+ is a semantic token representing a set of parties; an unmarked p is a completely distinct token representing a single party. Note the use of a superscript "+" to denote sets of parties instead of a hat or boldface; this denotes that these lists may never be empty.[2] The typing and semantic rules will enforce this invariant as needed. When referring to a census, or when a set of parties should be understood as a "context" rather than an "attribute", we write Θ rather than p^+; this is entirely to clarify intent and the distinction has no formal meaning.

2.2.2 The Mask Operator

Here we introduce the $\triangleright$ operator, the purpose of which is to allow Theorem 2 (semantic stepping preserves types) to hold without adding sub-typing or polymorphism to λ_C. $\triangleright$ is a partial function defined in Figure 2.2; the left-hand argument is either a type (in which case it returns a type) or a value (in which case it returns a value). The effect of $\triangleright$ is very similar to EPP, except that it projects to a set of parties instead of just

[2] Later, we'll use an "$*$" to denote a possibly-empty set or list, and a "?" to denote "zero or one".

M	$::=$	V	Values.
	$\mid$	$M\,M$	Function application.
	$\mid$	$\mathsf{case}_{p^+}\ M\ \mathsf{of}\ \mathsf{Inl}\ x \Rightarrow M; \mathsf{Inr}\ x \Rightarrow M$	Branching on a disjoint-sum value.
V	$::=$	x	Variables.
	$\mid$	$(\lambda x : T\,.\,M)@p^+$	Function literals annotated with participants.
	$\mid$	$()@p^+$	Multiply-located unit.
	$\mid$	$\mathsf{Inl}\,V$	Injection to a disjoint-sum.
	$\mid$	$\mathsf{Inr}\,V$	
	$\mid$	$\mathsf{Pair}\,V\,V$	Construction of data pairs (products).
	$\mid$	$(V, \ldots, V)$	Construction of heterogeneous tuples.
	$\mid$	fst_{p^+}	Projection of data pairs.
	$\mid$	snd_{p^+}	
	$\mid$	$\mathsf{lookup}^n_{p^+}$	Projection of tuples.
	$\mid$	$\mathsf{com}_{p:p^+}$	Send to one or more recipients.
d	$::=$	$()$	We provide a simple algebra of "data" types,
	$\mid$	$d + d$	which can encode booleans or other finite types
	$\mid$	$d \times d$	and could be extended with natural numbers if desired.
T	$::=$	$d@p^+$	A complete multiply-located data type.
	$\mid$	$(T \to T)@p^+$	Functions are located at their participants.
	$\mid$	$(T, \ldots, T)$	A fixed-length heterogeneous tuple.

Figure 2.1: The complete syntax of the $\boldsymbol{\lambda_C}$ language.

one, and instead of introducing a $\perp$ symbol it is simply undefined in some cases. Because it is used during type-checking, errors related to it are caught at that time.

Consider an expression using a "masking identity" function: $(\lambda x : ()@\{p\}\,.\,x)@\{p\}\,()@\{p,q\}$, where the lambda is an identity function *application of which* turns a multiply-located unit value into one located at just p. Clearly, the lambda should type as $(()@\{p\} \to ()@\{p\})@\{p\}$; and so the whole application expression should type as $()@\{p\}$. Masking in the typing rules lets this work as expected, and similar masking in the semantic rules ensures type preservation.

$$\textsc{MTData}\ \frac{p^+ \cap \Theta \neq \varnothing}{d@p^+ \triangleright \Theta \ \triangleq\ d@(p^+ \cap \Theta)} \qquad \textsc{MTFunction}\ \frac{p^+ \subseteq \Theta}{(T \to T')@p^+ \triangleright \Theta \ \triangleq\ (T \to T')@p^+}$$

$$\textsc{MTVector}\ \frac{T_1' = T_1 \triangleright \Theta, \quad \ldots \quad T_n' = T_n \triangleright \Theta}{(T_1, \ldots, T_n) \triangleright \Theta \ \triangleq\ (T_1', \ldots, T_n')}$$

$$\textsc{MVLambda}\ \frac{p^+ \subseteq \Theta}{(\lambda x : T\,.\,M)@p^+ \triangleright \Theta \ \triangleq\ (\lambda x : T\,.\,M)@p^+} \qquad \textsc{MVUnit}\ \frac{p^+ \cap \Theta \neq \varnothing}{()@p^+ \triangleright \Theta \ \triangleq\ ()@(p^+ \cap \Theta)}$$

$$\textsc{MVInL}\ \frac{V' = V \triangleright \Theta}{\mathsf{Inl}\ V \triangleright \Theta \ \triangleq\ \mathsf{Inl}\ V'} \qquad \textsc{MVInR}\ \frac{\ldots}{\ldots} \qquad \textsc{MVProj1}\ \frac{p^+ \subseteq \Theta}{\mathsf{fst}_{p^+} \triangleright \Theta \ \triangleq\ \mathsf{fst}_{p^+}} \qquad \textsc{MVProj2}\ \frac{\ldots}{\ldots}$$

$$\textsc{MVPair}\ \frac{V_1' = V_1 \triangleright \Theta \quad V_2' = V_2 \triangleright \Theta}{\mathsf{Pair}\ V_1\ V_2 \triangleright \Theta \ \triangleq\ \mathsf{Pair}\ V_1'\ V_2'} \qquad \textsc{MVVector}\ \frac{V_1' = V_1 \triangleright \Theta \quad \ldots \quad V_n' = V_n \triangleright \Theta}{(V_1, \ldots, V_n) \triangleright \Theta \ \triangleq\ (V_1', \ldots, V_n')}$$

$$\textsc{MVProjN}\ \frac{p^+ \subseteq \Theta}{\mathsf{lookup}^n_{p^+} \triangleright \Theta \ \triangleq\ \mathsf{lookup}^n_{p^+}} \qquad \textsc{MVCom}\ \frac{s \in \Theta \quad r^+ \subseteq \Theta}{\mathsf{com}_{s;r^+} \triangleright \Theta \ \triangleq\ \mathsf{com}_{s;r^+}} \qquad \textsc{MVVar}\ \frac{}{x \triangleright \Theta \ \triangleq\ x}$$

Figure 2.2: Definition of the $\triangleright$ operator.

2.2.3 Typing Rules

The typing rules for $\boldsymbol{\lambda_C}$ are in Figure 2.3. A judgment $\Theta; \Gamma \vdash M : T$ says that M has type T in the context of a non-empty census Θ and a (possibly empty) list of variable bindings $\Gamma = (x_1 : T_1), \ldots (x_n : T_n)$. In TLambda and TProjN we write preconditions $\mathtt{noop}^{\triangleright p^+}(T)$ meaning $T = T \triangleright p^+$, *i.e.* masking to those parties is a "no-op".

Examine TCase as the most involved example. The actual judgment says that in the context of Θ and Γ, the case expression types as T. The first two preconditions say that the guard expression N must type in the parent context as some type T_N, which masks to the explicit party-set p^+ as a sum-type $(d_l + d_r)@p^+$. The

$$\textsc{TLambda}\ \frac{p^+;\Gamma,(x:T)\vdash M:T' \quad p^+\subseteq\Theta \quad \mathtt{noop}^{\triangleright p^+}(T)}{\Theta;\Gamma\vdash(\lambda x:T\,.\,M)@p^+:(T\to T')@p^+} \qquad \textsc{TVar}\ \frac{x:T\in\Gamma \quad T'=T\triangleright\Theta}{\Theta;\Gamma\vdash x:T'}$$

$$\textsc{TApp}\ \frac{\Theta;\Gamma\vdash M:(T_a\to T_r)@p^+ \quad \Theta;\Gamma\vdash N:T'_a \quad T'_a\triangleright p^+=T_a}{\Theta;\Gamma\vdash MN:T_r}$$

$$\textsc{TCase}\ \frac{\begin{array}{c}\Theta;\Gamma\vdash N:T_N \quad (d_l+d_r)@p^+=T_N\triangleright p^+\\ p^+;\Gamma,(x_l:d_l@p^+)\vdash M_l:T \quad p^+;\Gamma,(x_r:d_r@p^+)\vdash M_r:T \quad p^+\subseteq\Theta\end{array}}{\Theta;\Gamma\vdash \mathsf{case}_{p^+}\ N\ \mathsf{of}\ \mathsf{Inl}\,x_l\Rightarrow M_l;\mathsf{Inr}\,x_r\Rightarrow M_r:T}$$

$$\textsc{TUnit}\ \frac{p^+\subseteq\Theta}{\Theta;\Gamma\vdash()@p^+:()@p^+} \qquad \textsc{TPair}\ \frac{\Theta;\Gamma\vdash V_1:d_1@p_1^+ \quad \Theta;\Gamma\vdash V_2:d_2@p_2^+ \quad p_1^+\cap p_2^+\neq\varnothing}{\Theta;\Gamma\vdash \mathsf{Pair}\,V_1V_2:(d_1\times d_2)@(p_1^+\cap p_2^+)}$$

$$\textsc{TVec}\ \frac{\Theta;\Gamma\vdash V_1:T_1 \quad \ldots \quad \Theta;\Gamma\vdash V_n:T_n}{\Theta;\Gamma\vdash(V_1,\ldots,V_n):(T_1,\ldots T_n)} \qquad \textsc{TInl}\ \frac{\Theta;\Gamma\vdash V:d@p^+}{\Theta;\Gamma\vdash \mathsf{Inl}\,V:(d+d')@p^+} \qquad \textsc{TInr}\ \frac{\ldots}{\ldots}$$

$$\textsc{TProjN}\ \frac{p^+\subseteq\Theta \quad \mathtt{noop}^{\triangleright p^+}((T_1,\ldots,T_n))}{\Theta;\Gamma\vdash \mathsf{lookup}^i_{p^+}:((T_1,\ldots,T_i,\ldots,T_n)\to T_i)@p^+} \qquad \textsc{TProj2}\ \frac{\ldots}{\ldots}$$

$$\textsc{TProj1}\ \frac{p^+\subseteq\Theta}{\Theta;\Gamma\vdash \mathsf{fst}_{p^+}:((d_1\times d_2)@p^+\to d_1@p^+)@p^+}$$

$$\textsc{TCom}\ \frac{s\in s^+ \quad s^+\cup r^+\subseteq\Theta}{\Theta;\Gamma\vdash \mathsf{com}_{s;r^+}:(d@s^+\to d@r^+)@(\{s\}\cup r^+)}$$

Figure 2.3: $\boldsymbol{\lambda_C}$ typing rules.

only rule by which it can do that is MTDATA, so we can deduce that $T_N = (d_l + d_r)@q^+$, where q^+ is some unspecified superset of p^+. The third and forth preconditions require the bodies of the expression to conclave correctly: M_l and M_r must both type as T with the reduced census p^+ (and with the respective x_l and x_r bound to the right and left data types at p^+). The final precondition says that p^+ is a subset of Θ, *i.e.* everyone who's supposed to be branching is actually present to do so.

The other rules are mostly normal, with similar masking of types and conclaving of censuses as needed. In TVAR, the census masks the type bindings in Γ. In isolation, some expressions such as $\mathsf{Inr}()@\{p\}$ or the projection operators are flexible about their exact types; additional annotations could give them monomorphic typing, if that was desirable.

2.2.4 Masked Substitution

For $\triangleright$ to fulfil its purpose during semantic evaluation, it may need to be applied arbitrarily many times with different party-sets inside the new expressions, and it may not even be defined for all such party-sets. Conceptually, this just recapitulates the masking performed in TVAR. To formalize these subtleties, in Figure 2.4 we specialize the normal variable-substitution notation $M[x := V]$ to perform location-aware substitution. In Appendix A.1 we prove Theorem 1, which shows that this specialized substitution operation satisfies the usual concept of substitution. (Our various definitons and proofs about them in this work all assume Barendregt's variable convention. Roughly, this says that bound variables are unique. (Urban et al. 2007) provide a more detailed discussion.)

Theorem 1 (Substitution)**.** *If* $\Theta; \Gamma, (x : T_x) \vdash M : T$ *and* $\Theta; \Gamma \vdash V : T_x$*, then* $\Theta; \Gamma \vdash M[x := V] : T$.

2.2.5 Centralized Semantics

The semantic stepping rules for evaluating $\boldsymbol{\lambda_C}$ expressions are in Figure 2.5. In Sections 2.2.6 to 2.2.8 we will develop the "ground truth" of the distributed process semantics and show that the $\boldsymbol{\lambda_C}$'s semantics correctly capture distributed behavior.

$\boldsymbol{\lambda_C}$ is equipped with a substitution-based semantics that, after accounting for the $\triangleright$ operator and the specialized implementation of substitution, is quite standard among lambda-calculi. In particular, we make no effort here to support the out-of-order execution supported by some choreography languages. Because the

$$M[x := V] \quad \triangleq \quad \text{by pattern matching on } M\text{:}$$

$$y \quad \overset{\triangle}{\Longrightarrow} \quad \begin{cases} y \equiv x & \overset{\triangle}{\Longrightarrow} \quad V \\ y \not\equiv x & \overset{\triangle}{\Longrightarrow} \quad y \end{cases}$$

$$N_1 N_2 \quad \overset{\triangle}{\Longrightarrow} \quad N_1[x := V] N_2[x := V]$$

$$(\lambda y : T \,.\, N)@p^+ \quad \overset{\triangle}{\Longrightarrow} \quad \begin{cases} V \triangleright p^+ = V' & \overset{\triangle}{\Longrightarrow} \quad (\lambda y : T \,.\, N[x := V'])@p^+ \\ \text{otherwise} & \overset{\triangle}{\Longrightarrow} \quad M \end{cases}$$

$$\begin{array}{r} \mathsf{case}_{p^+}\, N \,\mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow M_l; \\ \mathsf{Inr}\, x_r \Rightarrow M_r \end{array} \quad \overset{\triangle}{\Longrightarrow} \quad \begin{cases} V \triangleright p^+ = V' & \overset{\triangle}{\Longrightarrow} \quad \mathsf{case}_{p^+}\, N[x := V] \,\mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow M_l[x := V']; \\ & \qquad\qquad\qquad\qquad \mathsf{Inr}\, x_r \Rightarrow M_r[x := V'] \\ \text{otherwise} & \overset{\triangle}{\Longrightarrow} \quad \mathsf{case}_{p^+}\, N[x := V] \,\mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow M_l; \\ & \qquad\qquad\qquad\qquad \mathsf{Inr}\, x_r \Rightarrow M_r \end{cases}$$

$$\mathsf{Inl}\, V_1 \quad \overset{\triangle}{\Longrightarrow} \quad \mathsf{Inl}\, V_1[x := V]$$

$$\mathsf{Inr}\, V_2 \quad \overset{\triangle}{\Longrightarrow} \quad \mathsf{Inr}\, V_2[x := V]$$

$$\mathsf{Pair}\, V_1 V_2 \quad \overset{\triangle}{\Longrightarrow} \quad \mathsf{Pair}\, V_1[x := V] V_2[x := V]$$

$$(V_1, \ldots, V_n) \quad \overset{\triangle}{\Longrightarrow} \quad (V_1[x := V], \ldots, V_n[x := V])$$

$$\left.\begin{array}{llll} ()@p^+ & \mathsf{fst}_{p^+} & \mathsf{snd}_{p^+} & \\ \mathsf{lookup}_i^{p^+} & \mathsf{com}_{s;r^+} & & \end{array}\right\} \quad \overset{\triangle}{\Longrightarrow} \quad M$$

Figure 2.4: The customised substitution used in $\boldsymbol{\lambda_C}$'s semantics.

language and corresponding computational model are parsimonious, no step-annotations are needed for the centralized semantics.

$$\textsc{AppAbs}\ \frac{V' = V \triangleright p^+}{((\lambda x : T\,.\,M)@p^+)V \longrightarrow M[x := V']} \qquad \textsc{App1}\ \frac{N \longrightarrow N'}{VN \longrightarrow VN'} \qquad \textsc{App2}\ \frac{M \longrightarrow M'}{MN \longrightarrow M'N}$$

$$\textsc{Case}\ \frac{N \longrightarrow N'}{\mathsf{case}_{p^+}\ N\ \mathsf{of}\ \mathsf{Inl}\,x_l \Rightarrow M_l; \mathsf{Inr}\,x_r \Rightarrow M_r \longrightarrow \mathsf{case}_{p^+}\ N'\ \mathsf{of}\ \mathsf{Inl}\,x_l \Rightarrow M_l; \mathsf{Inr}\,x_r \Rightarrow M_r}$$

$$\textsc{CaseL}\ \frac{V' = V \triangleright p^+}{\mathsf{case}_{p^+}\ \mathsf{Inl}\,V\ \mathsf{of}\ \mathsf{Inl}\,x_l \Rightarrow M_l; \mathsf{Inr}\,x_r \Rightarrow M_r \longrightarrow M_l[x_l := V']} \qquad \textsc{CaseR}\ \frac{\ldots}{\ldots}$$

$$\textsc{Proj1}\ \frac{V' = V_1 \triangleright p^+}{\mathsf{fst}_{p^+}(\mathsf{Pair}\,V_1V_2) \longrightarrow V'} \qquad \textsc{Proj2}\ \frac{\ldots}{\ldots} \qquad \textsc{ProjN}\ \frac{V' = V_i \triangleright p^+}{\mathsf{lookup}^i_{p^+}(V_1, \ldots, V_i, \ldots, V_n) \longrightarrow V'}$$

$$\textsc{Com1}\ \frac{()@p^+ \triangleright \{s\} = ()@s}{\mathsf{com}_{s;r^+}()@p^+ \longrightarrow ()@r^+} \qquad \textsc{ComPair}\ \frac{\mathsf{com}_{s;r^+}\,V_1 \longrightarrow V_1' \quad \mathsf{com}_{s;r^+}\,V_2 \longrightarrow V_2'}{\mathsf{com}_{s;r^+}(\mathsf{Pair}\,V_1V_2) \longrightarrow \mathsf{Pair}\,V_1'V_2'}$$

$$\textsc{ComInl}\ \frac{\mathsf{com}_{s;r^+}\,V \longrightarrow V'}{\mathsf{com}_{s;r^+}(\mathsf{Inl}\,V) \longrightarrow \mathsf{Inl}\,V'} \qquad \textsc{ComInr}\ \frac{\ldots}{\ldots}$$

Figure 2.5: $\boldsymbol{\lambda_C}$'s semantics.

The Com1 rule simply replaces one location-annotation with another. ComPair, ComInl, and ComInr are defined recursively amongst each other and Com1; the effect of this is that "data" values can be sent but other values (functions and variables) cannot.

As is typical for a typed lambda calculus, $\boldsymbol{\lambda_C}$ enjoys preservation and progress.

Theorem 2 (Preservation)**.** *If* $\Theta; \varnothing \vdash M : T$ *and* $M \longrightarrow M'$*, then* $\Theta; \varnothing \vdash M' : T$*.*

Theorem 3 (Progress)**.** *If* $\Theta; \varnothing \vdash M : T$*, then either* M *is of form* V *(which cannot step) or their exists* M' *s.t.* $M \longrightarrow M'$*.*

We prove these properties in Appendices A.2 and A.3 respectively.

2.2.6 The Local Process Language

In order to define EPP and a "ground truth" for λ_C computation, we need a locally-computable language, λ_L, into which it can project. λ_L is very similar to λ_C; to avoid ambiguity we denote λ_L expressions B (for "behavior") instead of M (which denotes a λ_C expression) and λ_L values L instead of V. The syntax is presented in Figure 2.6.

B	$::=$	$L \mid BB$	Process expressions.
		$\mid \mathsf{case}\ B\ \mathsf{of}\ \mathsf{Inl}\ x \Rightarrow B; \mathsf{Inr}\ x \Rightarrow B$	
L	$::=$	$x \mid () \mid \lambda x\,.\,B$	Process values.
		$\mid \mathsf{Inl}\ L \mid \mathsf{Inr}\ L \mid \mathsf{Pair}\ LL$	
		$\mid \mathsf{fst} \mid \mathsf{snd}$	
		$\mid (L, \ldots, L) \mid \mathsf{lookup}^n$	
		$\mid \mathsf{recv}_p \mid \mathsf{send}_{p^*}$	Receive from one party. Send to many.
		$\mid \mathsf{send}^*_{p^*}$	Send to many *and* keep for oneself.
		$\mid \bot$	"Missing" (located someplace else).

Figure 2.6: Syntax for the λ_L language.

λ_L differs from λ_C in a few ways. It's untyped, and the party-set annotations are mostly missing. λ_C's $\mathsf{com}_{p;q^+}$ operator is replaced by send_{q^+} and recv_p, as well as a $\mathsf{send}^*_{q^+}$, which differs from send_{q^+} only in that the process which calls it keeps a copy of the sent value for itself. Syntactically, the recipient lists of send and send^* may be empty; this keeps semantics consistent in the edge case implied by a λ_C expression like $\mathsf{com}_{s;\{s\}}$ (which is useless but legal). Finally, the value-form $\bot$ ("bottom") is a stand-in for parts of the choreography that do not involve the target party. In the context of choreographic languages, $\bot$ does not denote an error but should instead be read as "unknown" or "somebody else's problem".

The behavior of $\bot$ during semantic evaluation can be handled a few different ways, the pros-and-cons of which are not important in this work. We use a $\bot$-normalizing "floor" function, defined in Figure 2.7, during EPP and semantic stepping to avoid ever handling $\bot$-equivalent expressions like $\mathsf{Pair}\ \bot\bot$ or $\bot()$.

$\lfloor B \rfloor \triangleq$ by pattern matching on B: (Observe that floor is idempotent.)

$$B_1 B_2 \overset{\triangle}{\Longrightarrow} \begin{cases} \lfloor B_1 \rfloor = \bot, \lfloor B_2 \rfloor = L \overset{\triangle}{\Longrightarrow} \bot \\ \text{else} \overset{\triangle}{\Longrightarrow} \lfloor B_1 \rfloor \lfloor B_2 \rfloor \end{cases}$$

$$\mathsf{case}\ B_G\ \mathsf{of}\ \mathsf{Inl}\ x_l \Rightarrow B_l ; \mathsf{Inr}\ x_r \Rightarrow B_r \overset{\triangle}{\Longrightarrow} \begin{cases} \lfloor B_G \rfloor = \bot \overset{\triangle}{\Longrightarrow} \bot \\ \text{else} \overset{\triangle}{\Longrightarrow} \mathsf{case}\ \lfloor B_G \rfloor\ \mathsf{of}\ \mathsf{Inl}\ x_l \Rightarrow \lfloor B_l \rfloor ; \mathsf{Inr}\ x_r \Rightarrow \lfloor B_r \rfloor \end{cases}$$

$$\lambda x\,.\,B' \overset{\triangle}{\Longrightarrow} \lambda x\,.\,\lfloor B' \rfloor$$

$$\mathsf{Inl}\ L \overset{\triangle}{\Longrightarrow} \begin{cases} \lfloor L \rfloor = \bot \overset{\triangle}{\Longrightarrow} \bot \\ \text{else} \overset{\triangle}{\Longrightarrow} \mathsf{Inl}\ \lfloor L \rfloor \end{cases}$$

$$\mathsf{Inr}\ L \overset{\triangle}{\Longrightarrow} \begin{cases} \lfloor L \rfloor = \bot \overset{\triangle}{\Longrightarrow} \bot \\ \text{else} \overset{\triangle}{\Longrightarrow} \mathsf{Inr}\ \lfloor L \rfloor \end{cases}$$

$$\mathsf{Pair}\ L_1 L_2 \overset{\triangle}{\Longrightarrow} \begin{cases} \lfloor L_1 \rfloor = \bot = \lfloor L_2 \rfloor \overset{\triangle}{\Longrightarrow} \bot \\ \text{else} \overset{\triangle}{\Longrightarrow} \mathsf{Pair}\ \lfloor L_1 \rfloor \lfloor L_2 \rfloor \end{cases}$$

$$(L_1, \ldots, L_n) \overset{\triangle}{\Longrightarrow} \begin{cases} \forall_{i \in [1,n]} \lfloor L_i \rfloor = \bot \overset{\triangle}{\Longrightarrow} \bot \\ \text{else} \overset{\triangle}{\Longrightarrow} (\lfloor L_1 \rfloor, \ldots, \lfloor L_n \rfloor) \end{cases}$$

$$\left.\begin{array}{l} x \\ () \\ \mathsf{fst} \\ \mathsf{snd} \\ \mathsf{lookup}^i \\ \mathsf{send}_{p^*} \\ \mathsf{send}^*_{p^*} \\ \mathsf{recv}_p \\ \bot \end{array}\right\} \overset{\triangle}{\Longrightarrow} B$$

Figure 2.7: The "floor" function, which reduces $\bot$-based expressions.

$\boldsymbol{\lambda_L}$'s semantic stepping rules are given in Figure 2.8. Local steps are labeled with send ($\oplus$) and receive ($\ominus$) sets, like so: $B \xrightarrow{\oplus\{(p,L_1),(q,L_2)\};\ominus\{(r,L_3),(s,L_4)\}} B'$, or $B \xrightarrow{\oplus\mu;\ominus\eta} B'$ when we don't need to inspect the contents of the annotations. The floor function is used to keep expressions normalized during evaluation. Otherwise, most of the rules are analogous to the corresponding $\boldsymbol{\lambda_C}$ rules from Figure 2.5. The LSend- rules are defined recursively, similar to the Com- rules. LSendSelf shows that send^* is exactly like send except it locally acts like `id` instead of returning $\bot$. LRecv shows that the recv operator ignores its argument and can return *anything*, with the only restriction being that the return value must be reflected in the receive-set step-annotation.

$$\textsc{LAbsApp}\ \frac{}{(\lambda x\,.\,B)L \xrightarrow{\oplus\varnothing;\ominus\varnothing} \lfloor B[x := L]\rfloor} \qquad \textsc{LApp1}\ \frac{B \xrightarrow{\oplus\mu;\ominus\eta} B'}{LB \xrightarrow{\oplus\mu;\ominus\eta} \lfloor LB'\rfloor} \qquad \textsc{LApp2}\ \frac{B \xrightarrow{\oplus\mu;\ominus\eta} B'}{BB_2 \xrightarrow{\oplus\mu;\ominus\eta} \lfloor B'B_2\rfloor}$$

$$\textsc{LCase}\ \frac{B \xrightarrow{\oplus\mu;\ominus\eta} B'}{\mathsf{case}\ B\ \mathsf{of}\ \mathsf{Inl}\ x_l \Rightarrow B_l; \mathsf{Inr}\ x_r \Rightarrow B_r \xrightarrow{\oplus\mu;\ominus\eta} \lfloor \mathsf{case}\ B'\ \mathsf{of}\ \mathsf{Inl}\ x_l \Rightarrow B_l; \mathsf{Inr}\ x_r \Rightarrow B_r\rfloor}$$

$$\textsc{LCaseL}\ \frac{}{\mathsf{case}\ \mathsf{Inl}\ L\ \mathsf{of}\ \mathsf{Inl}\ x_l \Rightarrow B_l; \mathsf{Inr}\ x_r \Rightarrow B_r \xrightarrow{\oplus\varnothing;\ominus\varnothing} \lfloor B_l[x_l := L]\rfloor} \qquad \textsc{LCaseR}\ \frac{\cdots}{\cdots}$$

$$\textsc{LProj1}\ \frac{}{\mathsf{fst}\,(\mathsf{Pair}\ L_1\,L_2) \xrightarrow{\oplus\varnothing;\ominus\varnothing} L_1} \qquad \textsc{LProj2}\ \frac{\cdots}{\cdots} \qquad \textsc{LProjN}\ \frac{}{\mathsf{lookup}^i\,(L_1,\ldots,L_i,\ldots,L_n) \xrightarrow{\oplus\varnothing;\ominus\varnothing} L_i}$$

$$\textsc{LSend1}\ \frac{}{\mathsf{send}_{p^*}\,() \xrightarrow{\oplus\{(p,())\mid p\in p^*\};\ominus\varnothing} \bot} \qquad \textsc{LSendPair}\ \frac{\mathsf{send}_{p^*}\ L_1 \xrightarrow{\oplus\mu_1;\ominus\varnothing} \bot \quad \mathsf{send}_{p^*}\ L_2 \xrightarrow{\oplus\mu_2;\ominus\varnothing} \bot}{\mathsf{send}_{p^*}\,(\mathsf{Pair}\ L_1\,L_2) \xrightarrow{\oplus\{(p,\mathsf{Pair}\ L_1\,L_2)\mid p\in p^*\};\ominus\varnothing} \bot}$$

$$\textsc{LSendInL}\ \frac{\mathsf{send}_{p^*}\ L \xrightarrow{\oplus\mu;\ominus\varnothing} \bot}{\mathsf{send}_{p^*}\,(\mathsf{Inl}\ L) \xrightarrow{\oplus\{(p,\mathsf{Inl}\ L)\mid p\in p^*\};\ominus\varnothing} \bot} \qquad \textsc{LSendInR}\ \frac{\cdots}{\cdots} \qquad \textsc{LSendSelf}\ \frac{\mathsf{send}_{p^*}\ L \xrightarrow{\oplus\mu;\ominus\varnothing} \bot}{\mathsf{send}^*_{p^*}\ L \xrightarrow{\oplus\mu;\ominus\varnothing} L}$$

$$\textsc{LRecv}\ \frac{}{\mathsf{recv}_p\ L_0 \xrightarrow{\oplus\varnothing;\ominus\{(p,L)\}} L}$$

Figure 2.8: The semantics of $\boldsymbol{\lambda_L}$.

2.2.7 Endpoint Projection

Endpoint projection (EPP) is the translation between the choreographic language $\boldsymbol{\lambda_C}$ and the local process language $\boldsymbol{\lambda_L}$; necessarily it's parameterized by the specific local process you're projecting *to*. $[\![M]\!]_p$ is the projection of M to p, as defined in Figure 2.9. It does a few things: Most location annotations are removed,

some expressions become $\bot$, $\bot$-based expressions are normalized by the floor function, and $\mathsf{com}_{s;r^+}$ becomes send_{r^+}, $\mathsf{send}^*_{r^+}$, or recv_s, keeping only the identities of the peer parties and not the local party.

2.2.8 Process Networks

A single party evaluating local code can hardly be considered the ground truth of choreographic computation; for a message to be sent it must be received *by* someone (and *visa-versa*). Our third "language", $\boldsymbol{\lambda_N}$, is just concurrent asynchronous threads of $\boldsymbol{\lambda_L}$. An $\boldsymbol{\lambda_N}$ "network" $\mathcal{N}$ is a dictionary mapping each party in its domain to a $\boldsymbol{\lambda_L}$ program representing that party's current place in the execution. We express party-lookup as $\mathcal{N}(p) = B$. A singleton network, written $\mathcal{N} = p[B]$, has the one party p in its domain and assigns the expression B to it. Parallel composition of networks is expressed as $\mathcal{N} \mid \mathcal{N}'$ (the order doesn't matter). Thus, the following statements are basically equivalent:

- $\mathcal{N}(p) = B$
- $\mathcal{N} = p[B] \mid \mathcal{N}'$
- $p[B] \in \mathcal{N}$

When many compositions need to be expressed at once, we can write $\mathcal{N} = \Pi_{p \in p^+} p[B_p]$. Parallel projection of all participants in M is expressed as $[\![M]\!] = \Pi_{p \in \mathtt{roles}(M)} p[[\![M]\!]_p]$. For example, if p and q are the only parties in M, then $[\![M]\!] = p[[\![M]\!]_p] \mid q[[\![M]\!]_q]$.

The rules for $\boldsymbol{\lambda_N}$ semantics are in Figure 2.10. $\boldsymbol{\lambda_N}$ semantic steps are annotated with *incomplete* send actions; $\mathcal{N} \xrightarrow{p:\{\ldots,(q_i,L_i),\ldots\}} \mathcal{N}'$ indicates a step in which p sent a respective L_i to each of the listed q_i and the q_is have *not* been noted as receiving. When there are no such incomplete sends and the p doesn't matter, it may be omitted (*e.g.* $\mathcal{N} \xrightarrow{\varnothing} \mathcal{N}'$ instead of $\mathcal{N} \xrightarrow{p:\varnothing} \mathcal{N}'$). **Only $\varnothing$-annotated steps are "real";** other steps are conceptual justifications used in the semantics's derivation trees. In other words, $\boldsymbol{\lambda_L}$ semantics only elevate to $\boldsymbol{\lambda_N}$ semantics when the message-annotations cancel out. Rule NCom allows annotations to cancel out. For example the network $[\![\mathsf{com}_{s;\{p,q\}}()@\{s\}]\!]$ gets to $[\![()@\{p,q\}]\!]$ by a *single* NCom step. The derivation tree for that step starts at the top with NPro: $s[\mathsf{send}_{\{p,q\}}()] \xrightarrow{s:\{(p,()),(q,())\}} s[\bot]$; this justifies two nestings of NCom in which the p step and q step (in either order) compose with the s step and remove the respective party from the step-annotation.

$$
\begin{array}{rcl}
[\![M]\!]_p & \triangleq & \text{by pattern matching on } M\text{:} \\
N_1 N_2 & \overset{\triangle}{\Longrightarrow} & \lfloor [\![N_1]\!]_p [\![N_2]\!]_p \rfloor \\
\mathsf{case}_{p^+}\ N\ \mathsf{of}\ \mathsf{Inl}\ x_l \Rightarrow M_l;\ \mathsf{Inr}\ x_r \Rightarrow M_r & \overset{\triangle}{\Longrightarrow} & \begin{cases} p \in p^+ & \overset{\triangle}{\Longrightarrow} \quad \lfloor \mathsf{case}\, [\![N]\!]_p\ \mathsf{of}\ \mathsf{Inl}\ x_l \Rightarrow [\![M_l]\!]_p; \mathsf{Inr}\ x_r \Rightarrow [\![M_r]\!]_p \rfloor \\ \text{else} & \overset{\triangle}{\Longrightarrow} \quad \lfloor \mathsf{case}\, [\![N]\!]_p\ \mathsf{of}\ \mathsf{Inl}\ x_l \Rightarrow \bot; \mathsf{Inr}\ x_r \Rightarrow \bot \rfloor \end{cases} \\
x & \overset{\triangle}{\Longrightarrow} & x \\
(\lambda x : T\,.\,N)@p^+ & \overset{\triangle}{\Longrightarrow} & \begin{cases} p \in p^+ & \overset{\triangle}{\Longrightarrow} \quad \lambda x\,.[\![N]\!]_p \\ \text{else} & \overset{\triangle}{\Longrightarrow} \quad \bot \end{cases} \\
()@p^+ & \overset{\triangle}{\Longrightarrow} & \begin{cases} p \in p^+ & \overset{\triangle}{\Longrightarrow} \quad () \\ \text{else} & \overset{\triangle}{\Longrightarrow} \quad \bot \end{cases} \\
\mathsf{Inl}\ V & \overset{\triangle}{\Longrightarrow} & \lfloor \mathsf{Inl} [\![V]\!]_p \rfloor \\
\mathsf{Inr}\ V & \overset{\triangle}{\Longrightarrow} & \lfloor \mathsf{Inr} [\![V]\!]_p \rfloor \\
\mathsf{Pair}\ V_1 V_2 & \overset{\triangle}{\Longrightarrow} & \lfloor \mathsf{Pair} [\![V_1]\!]_p [\![V_2]\!]_p \rfloor \\
(V_1, \ldots, V_n) & \overset{\triangle}{\Longrightarrow} & \lfloor ([\![V_1]\!]_p, \ldots, [\![V_n]\!]_p) \rfloor \\
\mathsf{fst}_{p^+} & \overset{\triangle}{\Longrightarrow} & \begin{cases} p \in p^+ & \overset{\triangle}{\Longrightarrow} \quad \mathsf{fst} \\ \text{else} & \overset{\triangle}{\Longrightarrow} \quad \bot \end{cases} \\
\mathsf{snd}_{p^+} & \overset{\triangle}{\Longrightarrow} & \begin{cases} p \in p^+ & \overset{\triangle}{\Longrightarrow} \quad \mathsf{snd} \\ \text{else} & \overset{\triangle}{\Longrightarrow} \quad \bot \end{cases} \\
\mathsf{lookup}^i_{p^+} & \overset{\triangle}{\Longrightarrow} & \begin{cases} p \in p^+ & \overset{\triangle}{\Longrightarrow} \quad \mathsf{lookup}^i \\ \text{else} & \overset{\triangle}{\Longrightarrow} \quad \bot \end{cases} \\
\mathsf{com}_{s;r^+} & \overset{\triangle}{\Longrightarrow} & \begin{cases} p = s, p \in r^+ & \overset{\triangle}{\Longrightarrow} \quad \mathsf{send}^*_{r^+\setminus\{p\}} \\ p = s, p \notin r^+ & \overset{\triangle}{\Longrightarrow} \quad \mathsf{send}_{r^+} \\ p \neq s, p \in r^+ & \overset{\triangle}{\Longrightarrow} \quad \mathsf{recv}_s \\ \text{else} & \overset{\triangle}{\Longrightarrow} \quad \bot \end{cases}
\end{array}
$$

Figure 2.9: EPP from $\boldsymbol{\lambda_C}$ to $\boldsymbol{\lambda_L}$.

$$
\textsc{NPro}\ \frac{B \xrightarrow{\oplus\mu;\ominus\varnothing} B'}{p[B] \xrightarrow{p:\mu} p[B']} \qquad \textsc{NCom}\ \frac{\mathcal{N} \xrightarrow{s:\mu\cup\{(r,L)\}} \mathcal{N}' \quad B \xrightarrow{\oplus\varnothing;\ominus\{(s,L)\}} B'}{\mathcal{N} \mid r[B] \xrightarrow{s:\mu} \mathcal{N}' \mid r[B']} \qquad \textsc{NPar}\ \frac{\mathcal{N} \xrightarrow{\varnothing} \mathcal{N}'}{\mathcal{N} \mid \mathcal{N}^+ \xrightarrow{\varnothing} \mathcal{N}' \mid \mathcal{N}^+}
$$

Figure 2.10: Semantic rules for $\boldsymbol{\lambda_N}$.

2.2.9 Deadlock Freedom

Above we introduced the necessary machinery of EPP and evaluation of a network of communicating processes. One of the advantages of choreogrphic programming is that a user can typically ignore this distributed computational setting, and just reason about their programs in a single-threaded way, *i.e.* under the centralized semantics of λ_C. Such an andvantage only holds water if EPP to λ_N is sound and complete with respect to λ_C; Theorems 4 and 5 show that it is.

Theorem 4 (Soundness)**.** *If* $\Theta; \varnothing \vdash M : T$ *and* $[\![M]\!] \xrightarrow{\varnothing}{}^{*} \mathcal{N}_n$*, then there exists* M' *such that* $M \longrightarrow^{*} M'$ *and* $\mathcal{N}_n \xrightarrow{\varnothing}{}^{*} [\![M']\!]$.

Theorem 5 (Completeness)**.** *If* $\Theta; \varnothing \vdash M : T$ *and* $M \longrightarrow M'$*, then* $[\![M]\!] \xrightarrow{\varnothing}{}^{*} [\![M']\!]$.

In Appendix A.4 we prove Theorem 4, which says that any behavior possible for the λ_N projection of a choreography is also possible in the original λ_C. In Appendix A.5 we prove Theorem 5, which says that any behavior possible in λ_C is also possible in the λ_N projection.

A foundational promise of choreographic programming is that participants in well-formed choreographies will never get stuck waiting for messages they never receive. This important property, *"deadlock freedom by design"*, is trivial once our previous theorems are in place.

Corollary 1 (Deadlock Freedom)**.** *If* $\Theta; \varnothing \vdash M : T$ *and* $[\![M]\!] \xrightarrow{\varnothing}{}^{*} \mathcal{N}$*, then either* $\mathcal{N} \xrightarrow{\varnothing}{}^{*} \mathcal{N}'$ *or for every* $p \in \texttt{roles}(M)$*,* $\mathcal{N}(p)$ *is a value.*

This follows from Theorems 2 to 5.

2.3 Comparisons with other systems

In this section we compare recent choreography languages to λ_C, primarily in terms of how their KoC strategies impact communication efficiency. By "communication efficiency" we refer to the amount of information sent from each party to each other party in a choreography acomplishing some desired global behavior or end state.

For readability, we render λ_C examples in this section as plain-text. We use `fn` for λ, `=>` for ⇒, `->` for →, and `*` for ×. The annotations on lambdas, unit, and keyword functions are given as comma-separated lists in square brackets (*e.g.* `lookup[2][p_1,p_2,q]` and `com[s][r_1]`). Furthermore, we sugar our

syntax with let-binding, *e.g.* $(\lambda var : T\,.\,M)@\Theta V$ is rendered as `let var : T = V; M`, and often we'll omit the type annotation `T`. We elide declarations of contextual functions and data types in our examples. We allow expressions in place of values, which can be de-sugared to temp variables. Some of the languages we compare against include polymorphic functions in their examples; we annotate such function names in our comparison code, similar to how our built-ins like `fst` get annotated.

2.3.1 HasChor

HasChor is a Haskell library for writing choreographies as values of a monad `Choreo` (Shen et al. 2023). The implementation is succinct and easy to use. HasChor does not have `select` statements; KoC is handled by broadcasting branch-guards to all participants in the choreography. This is not efficient. The choreography in Figure 1.4*(a)* is a translation into MultiChor of an example from (Shen et al. 2023), and shows explicitly the redundant communication that's implicit in HasChor choreographies. Figure 2.11 shows a $\boldsymbol{\lambda_C}$ version of the amended choreography from Figure 1.4*(b)*.

```
1  (fn request : (PutRequest + GetRequest)@[client] .
2    let request_ = com[client][primary] request;
3    let req = com[primary][primary, backup] request_;
4    let _ = case[primary, backup] req of
5      Inl _put => let _ack = com[backup][primary] (handleRequest@[backup] req);
6                  ()@[primary, backup];
7      Inr _get => ()@[primary, backup];
8    let response : Response@[primary] = handleRequest@[primary] request_;
9    com[primary][client] response
10 )@[client, primary, backup]
```

This choreography is represented as a function from a Sum-Type located at `client` (`request` on line 1) to some unspecified "response" type also located at `client` (the return type of `com[_][client]`, line 9). The census annotation follows the function body (line 10). `request`, `request_`, and `req` all contain the same data, but have different owners (respectively, `[client]`, `[primary]`, and `[primary,backup]`). The `case` expression (line 4) explicitly conclaves to the sub-census `[primary,backup]`. Although the choreography looks (and in practice would execute) very much like the MultiChor version in Figure 1.4*(b)*, the actual semantics does not use a monad; this representation would de-sugar to a nesting of lambda abstractions and applications.

Figure 2.11: A $\boldsymbol{\lambda_C}$ choreography implementing the same KVS as in Figure 1.4.

2.3.2 Pirouette

Pirouette (Hirsch and Garg 2022) is a functional choreographic language. It uses the select-&-merge KoC strategy formalized in (Carbone and Montesi 2013): a branching party sends flag symbols to peers who need to behave differently depending on the branch. These `select` statements are written explicitly by the user and can be quite parsimonious. Only if, and not until, the EPPs of the parallel program branches are different for a given user does that user need to be sent a `select`. EPP of an `if` statement uses a "merge" operation to combine program branches that are not distinguishable to a given party. `select` statements project as the `offer` and `choose` operations from multiparty-session-types.

The "merge" function is partial; if needed `select`s are missing from a program then EPP can fail because the merge of the EPPs of two paths is undefined. Pirouette's type system doesn't detect this; to check if a Pirouette program is well-formed one must do all of the relevant endpoint projections. (All select-&-merge systems we've investigated work this way.) (Hirsch and Garg 2022) provide a standalone implementation of Pirouette and Coq proofs of their theorems.

`select` gives good communication efficiency because not every choice needs to be communicated, but it has a limitation in common with HasChor. The `select` flags can't be used as data, and the Knowledge of Choice they communicate can't be recycled in subsequent conditionals. Figure 2.12 shows a λ_C choreography with sequential branches: on lines 2 and 7 `alice` and `bob` branch on their shared MLV `choice`. To represent this behavior in Pirouette without redundant messages, the sequential conditionals must be combined and Carroll's actions that happen in between (lines 5 and 6) must be duplicated in each branch. This is shown in Figure 2.13; Notice that Carroll is never informed which branch she is in; her actions are the same in each case. We believe Pirouette's communication efficiency is as good as λ_C's, but scaling the above strategy for combining sequential conditionals across a large codebase could be challenging.

2.3.3 Chorλ

Chorλ (Cruz-Filipe et al. 2022) is a functional choreographic language. The API and communication efficiency are similar to (Hirsch and Garg 2022) and (Giallorenzo et al. 2024), but (Cruz-Filipe et al. 2023) shows that Chorλ's semantics and typing can additionally support structures called *Distributed Choice Types*. A multiply-located `()@[p,q]` is isomorphic to a tuple of singly-located values `((()@p, ()@q)`.

```
let choice : ()+()@[alice, bob] = com[alice][alice, bob] alices_choice;
let query : Query@[alice] = case[alice, bob] choice of
  Inl _ => com[bob][alice] bobs_query;
  Inr _ => alices_query;
let answerer : (Query@[carroll] -> Response@[carroll])@[carroll] = carrolls_func;
let response = com[carroll][bob, alice] (answerer (com[alice][carroll] query));
case[alice, bob] choice of
  Inl _ => bobs_terminal response;
  Inr _ => alices_terminal response;
```

This choreography involves two clients and one server. Client `bob` may delegate a query against server `carroll`, or client `alice` may provide the query herself.

Figure 2.12: A λ_C implementation of a choreography involving sequential branches.

```
if alice.choice
  then alice[L] ~> bob;
       bob.bobs_query ~> alice.query;
       alice.query ~> carroll.query;
       carroll.(answerer(query)) ~> bob.response;
       carroll.(answerer(query)) ~> alice.response;
       bob.(terminal response)
  else alice[R] ~> bob;
       alice.alices_query ~> carroll.query;
       carroll.(answerer(query)) ~> bob.response;
       carroll.(answerer(query)) ~> alice.response;
       alice.(terminal response)
```

Figure 2.13: A Pirouette implementation of the client-server-delegation choreography in Figure 2.12

Distributed Choice Types extend this isomorphism to cover the entire algebra of Unit, Sum, and Product types in such a way that `p` and `q` never disagree about the value they each have. Specifically a multiply-located `(A + B)@[p,q]` becomes a singly-located `((A@p, A@q)+(B@p, B@q))`, a type which earlier systems do not support. [3]

Chorλ's "merge" operator supports branching on distributed choice types, so Chorλ can always match λ_C's communication efficiency with a similar program structure by declaring the needed `multicast` functions. The language still needs to support `select` (because Chorλ has no other way of implementing `multicast`), so well-formed-ness checking still depends on the partial function "merge".

Considering the other direction, λ_C can likewise match the communication efficiency of Chorλ and other `select`-based languages. Typically, this is as simple as multicasting the branch guard to all parties that would have received a `select` (and to oneself, the original branching party). In other situations a party might participate in branches without receiving a `select` because they don't need to know which one they are in; this is handled with the reverse of the transformation we showed between Figures 2.12 and 2.13

A fully-general algorithmic translation that never compromises on communication efficiency won't maintain the program's structure. The strategy is as follows:

- An expression M involving a party p who doesn't have KoC gets broken into three parts:
  - A computation N_1 of a cache data structure containing all variables bound up until the first part of M at which p actually does something.
  - A sub-expression N_2 involving p. p might be sending a message, receiving a message, receiving a `select`, or doing local computation.
  - A computation N_3 that unpacks the cache from N_1 and (possibly) the results from N_2 and proceeds with the *continuation*, the remainder of M. Note that N_3 will still need to undergo similar translation.
- Since there's KoC that p doesn't have, M must be a branch of a `case`. Since the original program was projectable, the other branch must have a similar breakdown *with the same N_2 middle part*. N_1, wrapped in a respective Inl or Inr, replaces M in the case statement. Depending if N_2 is to or from p,

[3] It should not be assumed that Chorλ is the last word in abstract models for the select-&-merge paradigm. Their `com` operator is defined for arbitrary arguments including functions; depending whether that's an aproperate defintion, `com` itself may not even be necessary.

the branches of the new `case` may also have to provide the argument to N_2, but this should *not* be wrapped in a Sum Type.

- If N_2 is a `select` operation, then it gets translated into a multicast. Its argument, provided by the preceding `case`, will be $\mathsf{Inl}()@q^+$ or $\mathsf{Inr}()@q^+$ depending on the symbol `select`ed[4], where q^+ are the parties who already have KoC. Then $\{p\} \cup q^+$ branch together on the multicast flag. The N_3 continuations will be handled in duplicate in both of the flag-branches; this will often involve dead branches for which applicable caches or behavior do not exist. Since these branches will never be hit, it's safe to populate them with default values of the appropriate type.

- Otherwise, sequencing of N_2 after the N_1-generating `case` is straightforward.

- To handle the N_3 continuations, branch on the cache value (which was wrapped in a Sum Type). In each branch, unpack the cached variables (and bind the results of N_2 if needed) and proceed with recursive translation of the continuation.

Neither (Cruz-Filipe et al. 2022) nor (Cruz-Filipe et al. 2023) contain examples requiring such a complicated translation. Figure 2.14 shows a made-up Chorλ choreography; translating it into $\boldsymbol{\lambda_C}$ without compromising communication efficiency is more involved than earlier examples were. Figure 2.15 shows it's translation via the steps described above; the code is intermediate in verbosity between an actual machine-generated translation and a thoughtful human reimplementation.

In the following chapter we present an eDSL implementation of conclave-&-MLVs choreographic programming, and demonstrate its use.

[4]Chorλ supports arbitrary symbols for `select`, but since we're concerned with bit-level efficiency we assume the only symbols are `L` and `R`.

```
case ( first_secret[p] ()@p ) of
  Inl _ => case ( second_secret[p] ()@p ) of
             Inl _ => let w = com[q][p] n_q1;
                      select[p][q] L;
                      let _ = com[p][q] (w + 1@p);
                      w + 1@p;
             Inr _ => let w = com[q][p] n_q1;
                      let y = 2@p;
                      select[p][q] L;
                      let _ = com[p][q] (w + y);
                      w;
  Inr _ => let w = com[q][p] n_q1;
           case (second_secret[p] ()@p ) of
             Inl s => select[p][q] L;
                      let _ = com[p][q] 5@p;
                      s;
             Inr _ => select[p][q] R;
                      let z = com[q][p] n_q2;
                      w + z;
```

Figure 2.14: A contrived Chorλ choreography that is complicated to efficiently translate into $\boldsymbol{\lambda_C}$.

Bibliography

Bates, M., S. Kashiwa, S. Jafri, G. Shen, L. Kuper, and J. P. Near (2025, June). Efficient, portable, census-polymorphic choreographic programming. *Proc. ACM Program. Lang. 9*(PLDI). Archive: `https://arxiv.org/abs/2412.02107`.

Carbone, M. and F. Montesi (2013). Deadlock-freedom-by-design: multiparty asynchronous global programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, New York, NY, USA, pp. 263–274. Association for Computing Machinery.

Cruz-Filipe, L., E. Graversen, L. Lugović, F. Montesi, and M. Peressotti (2023). Modular Compilation for Higher-Order Functional Choreographies. In K. Ali and G. Salvaneschi (Eds.), *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, Volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, pp. 7:1–7:37. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Cruz-Filipe, L., E. Graversen, L. Lugović, F. Montesi, and M. Peressotti (2022, September). *Theoretical Aspects of Computing*, Volume 13572 of *Lecture Notes in Computer Science*, Chapter Functional choreographic programming, pp. 212–237. Tbilisi, Georgia: Springer. Archive: `https://arxiv.org/abs/2111.03701`.

Giallorenzo, S., F. Montesi, and M. Peressotti (2024, jan). Choral: Object-oriented choreographic programming. *ACM Trans. Program. Lang. Syst. 46*(1).

Hirsch, A. K. and D. Garg (2022, January). Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang. 6*(POPL).

Kashiwa, S., G. Shen, S. Zare, and L. Kuper (2023). Portable, efficient, and practical library-level choreographic programming. `https://arxiv.org/abs/2311.11472`.

Shen, G., S. Kashiwa, and L. Kuper (2023, aug). Haschor: Functional choreographic programming for all (functional pearl). *Proc. ACM Program. Lang. 7*(ICFP).

```
let m1 = com[q][p] n_q1;
let (cache1, flag1) = case ( first_secret[p] ()@[p] ) of
  Inl _ => let (c1_, f1_) = case ( second_secret[p] ()@[p] ) of
              Inl _ => let w = m1;
                       (Inl w, Inl ()@[p]);
              Inr _ => let w = m1;
                       let y = 2@p;
                       (Inr (Pair w y), Inl ()@[p]);
           (Inl c1_, f1_);
  Inr _ => let (c1_, f1_) = let w = m1;
                            case ( second_secret[p] ()@[p] ) of
                              Inl s => (Inl (Pair w s), Inl ()@[p]);
                              Inr _ => (Inr w, Inr ()@[p]);
           (Inr c1_, f1_);
let f1_ = com[p][p,q] flag1;
case f1_ of Inl _ => let (cache2, m2) = case cache1 of
                       Inl c1l => let (c2_, m2_) = case c1l of
                                    Inl c1ll => let w = c1ll;
                                                (Inl w, w + 1@[p]);
                                    Inr c1lr => let (Pair w y) = c1lr;
                                                (Inr (Pair w y), w + y);
                                  (Inl c2_, m2_);
                       Inr c1r => let (c2_, m2_) = case c1r of
                                    Inl c1rl => let (Pair w s) = c1rl;
                                                (Pair w s, 5@[p]);
                                    Inr c1rr => (DEFAULT, DEFAULT); # DEAD BRANCH
                                  (Inr c2_, m2_);
                     let _ = com[p][q] m2;
                     case cache2 of
                       Inl c2l => case c2l of
                         Inl c2ll => let w = c2ll;
                                     w + 1@[p];
                         Inr c2lr => let (Pair w y) = c2lr;
                                     w;
                       Inr c2r => let (Pair w s) = c2r;
                                  s;
           Inr _ =>  let cache2 = case cache1 of
                       Inl c1l => DEFAULT; # DEAD BRANCH
                       Inr c1r => case c1r of
                         Inl c1rl => DEFAULT; # DEAD BRANCH
                         Inr c1rr => let w = c1rr;
                                     w;
                     let m2 = com[q][p] n_q2;
                     let w = cache2;
                     let z = m2;
                     w + z
```

Figure 2.15: An algorithmic λ_C translation of the choreography from Figure 2.14.

Urban, C., S. Berghofer, and M. Norrish (2007). Barendregt's variable convention in rule inductions. In F. Pfenning (Ed.), *Automated Deduction – CADE-21*, Berlin, Heidelberg, pp. 35–50. Springer Berlin Heidelberg.

Chapter 3

The MultiChor Library for Haskell

3.1 Introduction

In this chapter we demonstrate the practicality of conclaves-&-MLVs choreographic programming by presenting our implementation: the MultiChor Haskell library. MultiChor is a "just a library" CP system in the style of HasChor. We adopt HasChor's freer-monads and handlers design pattern, and embed the key aspects of λ_C's type system as type-level constraints with a bespoke proof-witness system. Furthermore, the flexibility of MultiChor's API and the power of Haskell's type system together suffice to support *census polymorphism*, a novel capability in CP systems.

A key innovation of λ_C is that KoC is enforced entirely by type-level management of the census. By representing the census as a type-level variable in Haskell, MultiChor enables polymorphism over both the size and membership of the census, a feature not considered in the construction of λ_C. All Haskell typing happens statically, and MultiChor's EPP happens at runtime (like HasChor's). This means that, like other cases of polymorphism in Haskell, location polymorphism in MultiChor must be resolved statically.

A few other desiderata motivate our implementation:

1. It should be possible to broadcast, *i.e.* to multicast a value to the entire census, and to use values known to the entire census as normal (un-located) values of their type.

2. It should be possible to know from an appropriately-written choreography's type that some certain party or parties are not involved, are not in its census. Users should be able to embed such "conclave" choreographies inside choreographies with larger, possibly polymorphic, censuses.

3. The type system should be able to reason about parties' membership in a census or ownership-set with normal subset reasoning.

The choreography in Figure 3.1 showcases the above points. The census of the whole program appears in the type and does not specify who the players are. The `conclave` on line 16 embeds a choreography whose census is exactly the monomorphic `"dealer"` and a polymorphic `player` (#2). The helper-function `broadcast` on line 18 functions as described in #1. Many examples of #3 are automated or hidden in MultiChor, but on line 16 the function `inSuper` is applied to `players :: Subset players ("dealer" : players)` and `player :: Member player players` to attest that `player` is present in the census.

3.2 Censuses, Conclaves, and MLVs in Haskell

MultiChor uses the same free-monad approach as HasChor (Shen et al. 2023) to implement choreographic programming, EPP, and the final interpretation to a real communication mechanism. Also like HasChor, MultiChor's `Choreo` monad is parameterised by a *local monad* in which parties' local computations can be expressed. A MultiChor type `Located ps t` is a multiply-located `t` owned by the parties `ps`. It is possible to write MultiChor functions that look and work like each of HasChor's three primitive operators, but the derived API in which users write MultiChor choreographies contains a clear analog of only one of HasChor's primitives. Haskell's monadic-`do` notation and purity-oriented type system make MultiChor code concise and safe (in the sense that users are unlikely to accidentally invalidate important invariants).

As explained in Chapter 2, our KoC strategy requires that the correctness (well-typed-ness) of choreographies be judged in the context of a census. MultiChor adds the census as a type parameter of the `Choreo` monad. Its kind is `[Symbol]`, which is to say that the census is a type-level list of parties and parties are type-level strings. `Choreo` is *not* an *indexed* monad (that is, executing a monadic operation doesn't change the census), but monadic operations can take choreographies with smaller censuses as arguments.

The fundamental operations of MultiChor's `Choreo` monad are `conclave`, `broadcast'`, `locally'`, and `congruently'`. Their type signatures are given in Figure 3.2. Like in HasChor, these are free-monad

```
game :: forall players m. (KnownSymbols players)
     => Choreo ("dealer" ': players) (CLI m) ()
game = do
  let players = consSuper (refl @players)
      dealer = listedFirst @"dealer"  -- `listedFirst` is basically just `First`.
      everyone = refl @("dealer" ': players)
  hand1 <- (fanIn everyone \(player :: Member player players) -> do
      card1 <- locally dealer (\_ ->
                   getInput ("Enter random card for " ++ toLocTm player))
      (dealer, card1) ~> everyone
    ) >>= naked everyone
  wantsNextCard <- parallel players \_ _ -> do
      putNote $ "All cards on the table: " ++ show hand1
      getInput "I'll ask for another? [True/False]"
  hand2 <- fanOut \(player :: Member player players) ->
    conclave (inSuper players player @@ dealer @@ nobody) do
        let dealer = listedSecond @"dealer"
        choice <- broadcast (listedFirst @player, localize player wantsNextCard)
        if choice then do
            cd2 <- locally dealer (\_ ->
                       getInput (toLocTm player ++ "'s second card:"))
            card2 <- broadcast (dealer, cd2)
            return [getLeaf hand1 player, card2]
        else return [getLeaf hand1 player]
  tblCrd <- locally dealer (\_ -> getInput "Enter a single card for everyone:")
  tableCard <- (dealer, tblCrd) ~> players
  void $ parallel players \player un -> do
      let hand = un player tableCard : viewFacet un player hand2
      putNote $ "My hand: " ++ show hand
      putOutput "My win result:" $ sum hand > card 19
```

This choreography is polymorphic over the number and identity of the players, but the party named `"dealer"` is an explicit member. The inner monad `CLI` that all parties have access to is a simple freer monad that can be handled to IO operations, or as `State` for testing purposes. The `newtype Card` encapsulates the modulo operation in its `Num` instance.

Figure 3.1: A card game expressed as a choreography written in MultiChor.

```
locally' :: (KnownSymbol l)
         => (Unwrap l -> m a)
         -> Choreo '[l] m a

congruently' :: (KnownSymbols ls)
             => (Unwraps ls -> a)
             -> Choreo ls m a

broadcast' :: (Show a, Read a, KnownSymbol l)
           => Member l ps
           -> (Member l ls, Located ls a)
           -> Choreo ps m a

conclave :: (KnownSymbols ls)
         => Subset ls ps
         -> Choreo ls m a
         -> Choreo ps m (Located ls a)
```

Of these four operators, `conclave` is the only one users will usually call directly; the other three can combine with each other (and with `conclave`) to make more user-friendly alternatives.

Figure 3.2: The fundamental operators for writing expressions in MultiChor's `Choreo` monad.

constructors; their behavior is implemented in interpreters that carry out EPP or implement a centralized semantics. Three of them have their names "primed" because the un-primed versions of these names are reserved for more ergonomic derived functions. For example, `locally'` takes a single argument, a computation in the local monad, and requires that the census contains *a single party*, who will execute that computation. The un-primed `locally` takes an additional argument that identifies a single party from a larger census; it uses `conclave` to correctly call `locally'`. `broadcast` shares a `Located` value with the entire census so the unwrapped value can be used; by combining this with `conclave` we can implement point-to-point or multicast communication. From the perspective of a centralized semantics, `conclave` doesn't do anything at all besides run the sub-choreography, but EPP to a party *not* in the sub-census skips the sub-choreography and just returns `Empty`.

`congruently` lets us leverage MLVs to concisely write actively-replicated computations. In contrast to `locally`, the computation is performed by multiple parties and the result is multiply-located across all of them.[1] For the execution of these actively-replicated computations to correctly return an MLV, all the parties must be guaranteed to be doing a pure computation on the same data. Haskell makes it easy to enforce such a guarantee to a practical (but not unbreakable) extent. This is why `congruently` does not grant access to the

[1]The entire census participates in the primed version, and its result is returned naked. The behavior of `conclave` and the more fundamental rules of monadic programming ensure the un-primed `congruently` behaves correctly.

local monad `m`. It also requires that the computation not have access to the specific identity of the computing party, unlike `locally` and the similar-looking function `parallel` mentioned in Section 3.4. Weakening (or subverting) these restrictions would allow a user to violate MultiChor's invariant that MLVs (`Located` values) have the same value across all their owners.

It is critical to the safety of MultiChor that the projection of a choreography to any given party will not use any other party's `Located` values. We use the same basic strategy for this as HasChor: `Located`'s constructors, `Wrap` and `Empty`, are hidden inside the core module and afforded only by dependency injection to `locally` and `congruently`. The specific "unwrapper" functions afforded to `locally` and `congruently` are known to user code only by their type signatures, which have respective aliases `Unwrap` and `Unwraps`. `Located`'s constructors are also used by two less-critical functions, `flatten` and `othersForget`. These are needed for shrinking ownership sets or un-nesting `Located` values; they could be written using `congruently`, but by implementing them in the core module where they can pattern-match `Located` values we are able to make them not-monadic, and so more convenient.

3.3 Membership Constraints & Proof Witnesses

It is not trivial for Haskell's type-checker (a component of GHC, the compiler) to judge if a particular participant owns a multiply-located value or is present in a particular census when the party or the set are polymorphic. Declaring membership and subset relations as class constraints can work in some situations, but this strategy has serious limitations which we find unacceptable. For example, a rule as obvious as $(p \in ps_1 \wedge ps1 \subseteq ps_2) \rightarrow p \in ps_2$, represented in Haskell as `instance ( IsMember p ps1, IsSubset ps1 ps2) => IsMember p ps2`, would be impossible to use because the compiler has no way of guessing which set `ps1` it should be checking `p`'s membership in. (Even if it could *guess*, it wouldn't backtrack and try a different guess when its first try didn't work).

To work around such limitations, MultiChor uses a strategy of *proof witnesses* like those described by (Noonan 2018). These are light-weight runtime values with specially crafted types; the (static) type encodes a logical assertion (*e.g.* $p \in p^+$) and the existence of a value of that type guarantees the truth of the assertion. We do not actually use the `gdp` [2] package; we found that writing our own purpose-specific system had a few advantages. First, we were able to write everything we needed without hand-waving any foundations as

[2] "Ghosts of Departed Proofs" (Noonan 2019)

`axiom`s. Second, pattern matching against the constructors of `Member l ls` suffices to convince GHC that `ls` is not empty, which is sometimes useful. Finally, the implicit paradigm of "memberships as indices & subsets as functions" was qualitatively easier to work with when we were building the census-polymorphism tools described in Section 3.4.

Figure 3.3 shows the implementation of this system and some of its idioms. In MultiChor, locations are identified by type-level strings, uninhabited types with "kind" `Symbol`. A values of type `Member p ps` can be used both as proof that `p` is eligible to take some action (because of their membership in `ps`) and as a term-level identifier for the party `p`. The actual form of a value of type `Member p ps` is equivalent to an index in the type-level list `ps` at which `p` appears. Subset relations are expressed and used similarly: A value of type `Subset ps qs` has the underlying form of a function from `Member p ps` to `Member p qs`, *universally quantified over* `p`. Because these logical structures can be built from scratch inside Haskell's type system, all of the machinery we use to do so can safely be exposed to end-users so that they can write their own proofs, as needed, inside choreographies. For example, a user could import the types and functions shown in Figure 3.3 to write

```
-- | Cons an element to the superset in a `Subset` value.
consSuper :: Subset xs ys -> Subset xs (y ': ys)
consSuper sxy = transitive sxy (Subset Later)
```

By far the most common such manipulation we used in our case studies was building "lists" (`Subset` values) out of constituent `Member` items using the pattern `p @@ q @@ nobody` (read as *"`p` and `q` and nobody else"*), which makes a `Subset '[p, q] ps` out of a `Member p ps` and a `Member q ps`.

```
1  -- Aliases for "parties" or "locations" represented by strings
2  type LocTm = String
3  type LocTy = GHC.TypeLits.Symbol  -- type level strings
4  -- Membership proofs
5  data Member (x :: k) (xs :: [k]) where
6    First :: Member x (x ': xs')
7    Later :: Member x xs -> Member x (y ': xs)
8  -- Subset proofs
9  newtype Subset xs ys = Subset { inSuper :: forall x. Member x xs -> Member x ys }
10
11 -- Examples
12 refl :: Subset xs xs
13 refl = Subset id
14 transitive :: Subset xs ys -> Subset ys zs -> Subset xs zs
15 transitive sxy syz = Subset $ inSuper syz . inSuper sxy
16 nobody :: Subset '[] ys
17 nobody = Subset \case {}
18 (@@) :: Member x ys -> Subset xs ys -> Subset (x ': xs) ys
19 mxy @@ sxy = Subset \case { First -> mxy
20                           ; Later mxxs -> inSuper sxy mxxs }
```

The proof witnesses are values of type `Member` or `Subset`. `Member p ps` (line 5) is a generic algebraic data type who's constructors can only be called when `p` actually is a member of `ps`. `Subset ps qs` (line 9) values are isomorphic to functions from membership-in-`ps` to membership-in-`qs`. (Haskell's `newtype` types are intermediate between data structures and type-aliases. To access the function form of a `Subset` one calls `inSuper` on it. This pattern avoids impredicative typing errors.) Lines 12–20 show examples of how proof witnesses can be made and combined, in particular the polymorphic value `nobody` (line 17) and the infix operator `@@` (line 19).

Figure 3.3: MultiChor's proof witness system for membership and subset constraints.

3.4 Census Polymorphism

So far, the example choreographies we have discussed have had fixed numbers of participants. In all prior CP systems this has been a syntactic constraint[3]: even systems that allow polymorphism over the identities of participants require the participants' "roles" to be explicitly defined in-context. This is a serious limitation for writing choreographic software; modern concurrent systems often use dozens to thousands of participants and are defined parametrically over their number of participants (Beck et al. 2023, Corrigan-Gibbs and Boneh 2017, Wu et al. 2020, Bonawitz et al. 2017, Keeler et al. 2023). We assert that such parametric protocol declarations are a required feature for CP to find mainstream use; our systems provide it in the form of *census polymorphism*.

By "census polymorphism", we mean that a choreographic expression is polymorphic over its census type-variable, including not just the specific identities listed but also the quantity.[4] Naïvely, this is trivial; any MultiChor expression can easily be written with a type variable as its census and the relevant parties (whose exact identities can also be polymorphic) can be guaranteed to be present by taking membership proofs as arguments. However, this approach has a limitation: Since the number of type variables of a choreography must be fixed and there is no way to *explicitly list* a variable number of parties, it follows that there may be parties in the census who are not identified by the proof arguments. Such un-enumerated parties will receive any broadcasts and participate in any active replication that applies to the whole census, but there's no way to specify them as the senders of messages, nor is there any way to specify that they should receive a message except by broadcasting it. For this reason, when we speak of "census polymorphism", we mean *useful* polymorphism that lets an unspecified quantity of parties actively participate in the choreography. For example one might wish to write a `gather` operation in which a polymorphic list of participants each send a computed value to a common recipient who aggregates them. Figure 3.4 shows an example MultiChor choreography for a key-value store with a polymorphic list of backup servers. In Section 3.5.1 we implement the GMW protocol (Goldreich et al. 2019), a foundational protocol in multi-party cryptography. In earlier CP systems it would have been necessary to hard-code the number of participants when writing such choreographies;

[3](Cruz-Filipe and Montesi 2016) describes an extension to Procedural Choreographies (PC) to allow lists of processes as arguments to procedures; although PC has not been implemented, the extended version would clearly be an exception to the above statement.

[4]In principle, one can split hairs between census polymorphism and similar polymorphism over other sets of parties, *e.g.* ownership sets. We have not found such distinctions to be useful for describing system capabilities, but they can be relevant when talking about the type of a given expression.

Census polymorphism is precisely the absence of such a restriction. Census polymorphism is achieved in MultiChor library by type-level programming in modern Haskell.

3.4.1 Loops, Facets, and Quires

The first thing that is necessary is a way to loop over a polymorphic list of parties. Census polymorphism as discussed in this work is *static*, *i.e.*, while one can write choreographies and choreographic functions that are census-polymorphic, it is always possible in principal to unroll the top-level choreography (that actually gets compiled) into a monomorphic form before you actually run anything. In Section 3.4.2 we discuss MultiChor's `sequenceP`, a runtime loop over statically-defined type level lists.

Less flexible options would still be viable. The most recent versions of ChoRus and ChoreoTS lack constructs analogous to `sequenceP`, and instead offer the pair of functions `fanOut` and `fanIn` (Bates et al. 2025b). These are both "for loops" over parties; `fanOut`'s return type is a heterogeneous structure of the returned values for each looped-over party (see next paragraph) and `fanIn` works similarly except the owners of the aggregated data do not vary over the loop. It's an open question whether the additional flexibility of MultiChor's approach has any real-world use! We also conjecture that even more restricted implementations would suffice for a majority of use-cases, specifically by offering the three operations `scatter`, `gather`, and `parallel`. `scatter` is multi-cast operation in which a distinct value is sent to each recipient, and `gather` is its dual. `parallel` is exactly like `locally`, except a list of parties execute the local computation in parallel. In MultiChor, these are derived operations, and we use them frequently in our case studies.

The second thing required for useful census polymorphism is the ability to express and use divergent data known by un-enumerated parties. We call such data structures *faceted values*[5]. (They're basically the same as the faceted values introduced in (Austin and Flanagan 2012), except their public facet is always "⊥" and multiple parties have distinct private facets.) Conceptually, a faceted value is similar to an MLV, in that it projects to an owner as a simple value and to a non-owner as a placeholder, but different owners of a faceted value will have different values for it. To see the need for faceted values, consider how one would express a census-polymorphic `gather` operation using only a type-level `for`-loop: The argument couldn't simply be a list, because `Located` values with different owners have different types. Each sender would need to generate its value to send *inside* the loop body, and the only way for the sent values to be distinct would be

[5]The word "faceted" is most commonly used in reference to a cut gemstone, but analogy to the facets of polyhedral playing dice might be more on-the-nose.

```
1  handleRequest :: forall backups. (KnownSymbols backups)
2                => Located '["primary"] Request
3                -> (Located '["primary"] (IORef State),
4                    Faceted backups '[] (IORef State))
5                -> Choreo ("primary" ': backups) IO (Located '["primary"] Response)
6  handleRequest request (primaryStateRef, backupsStateRefs) =
7    broadcast (primary, request) >>= \case
8      Put key value -> do oks <- parallel backups \backup un ->
9                                    handlePut (viewFacet un backup backupsStateRefs)
10                                              key value
11                          gathered <- gather backups (primary @@ nobody) oks
12                          locally primary \un ->
13                              if all isOk (un primary gathered)
14                                  then handlePut (un primary primaryStateRef)
15                                       key value
16                                  else return errorResponse
17     Get key -> locally primary \un -> handleGet (un primary primaryStateRef) key
18   where primary = listedFirst
19         backups = consSuper refl
20
21 kvs :: forall backups. (KnownSymbols backups)
22     => Located '["client"] Request
23     -> (Located '["primary"] (IORef State), Faceted backups '[] (IORef State))
24     -> Choreo ("client" ': "primary" ': backups) IO (Located '["client"] Response)
25 kvs request stateRefs = do
26     request' <- (client, request) ~> primary @@ nobody
27     response <- conclave (primary @@ backups) (handleRequest request' stateRefs)
28     (primary, flatten (First @@ nobody) (First @@ nobody) response)
29             ~> client @@ nobody
30   where client = listedFirst
31         primary = listedSecond
32         backups = consSuper $ consSuper refl
```

The main action happens in `handleRequest`, a choreography involving only the servers which is called via `conclave` on line 27. `handleRequest`'s census explicitly includes the primary server, but is polymorphic over the list of backup servers. The primary server broadcasts the request (line 7); the backups will update their state and report their health only for a `Put` request. On lines 8–10 the backups call the local IO function `handlePut` in `parallel` using their individual state references; `oks` is therefore a `Faceted backups '[] Response`. (The extra `'[]` denotes that no party yet knows all of the `oks`.) `gather` (line 11) communicates all the `oks` to the primary server where they're stored as a `Quire backups Response`. If all the backups are ok, then the primary server also handles the request (line 14).

Figure 3.4: A key-value store choreography with an unspecified number of backup servers.

by using private local state accessed by `locally`. This would hardly be satisfying, and the dual case of `scatter` would be even worse: Any use to which the received values were to be put would also have to fit inside the body of the `for`-loop. Again, one couldn't simply append the `scatter`ed values to a list and return it because (in Haskell) all the values in a list must have the same type.

The dual of a faceted value is a "quire"[6], a vector of values indexed by type-level parties. Quires are not inherently located, but they can be located the same way as any other data structure. For example, the return type of `gather` is `Located recipients (Quire senders a)`.

3.4.2 Census Polymorphism in MultiChor

We leverage the type system of modern Haskell to achieve useful census polymorphism in MultiChor. This behavior is implemented as a layer *on top of* MultiChor's central monad and data-types; from a theory perspective MultiChor gets census polymorphism "for free" because it's a Haskell library. (Therefore, we do not bother with a separate proof of the soundness of census polymorphism.) The MultiChor repository contains over a dozen example choreographies, several of which use census polymorphism. In Figure 3.4 we showcase a key-value store choreography that's polymorphic over the number of backup servers. Section 3.5.1 presents a more involved census-polymorphic example.

Key to MultiChor's strategy is Haskell's ability to express quantified type variables. For example, a `Faceted` value is (underneath a little boiler-plate) a function. Its argument is a `Member` proof that some party is in the list of owners, and it returns a `Located` value known to the party in question. Notably, nothing about the type, `Faceted ps cmn x`, indicates who the (type-level!) party indicated by the argument might be. (The second type parameter, `cmn`, represents parties who know *all* the contained values; it's frequently `'[]`.)

`Faceted ps cmn x` is actually a special case of a more general type, `PIndexed ps f`, where `f` can be any *type-level function* from a party to a concrete type. A `PIndexed` is like a type-indexed vector, except that the type of the value retrieved depends on the index. (The case where it does not depend on the index, *i.e.* when `f` is `Const`, is precisely `Quire`.) Because of its unusual `kind`, type classes that one would expect to apply to vectors generally do not apply to `PIndexed`. What's actually needed for census polymorphism is the ability to `sequence` a `PIndexed` of choreographies. Since `PIndexed` is not

[6] "Quire" is pronounced "choir"; it rhymes with "buyer" and means "a stack of sheets of paper, all cut to the same size". Each individual piece of paper is a "leaf".

an instance of `Traversable`, we implement the needed function `sequenceP`, which is effectively just a `for`-loop (in any monad) over type-level lists of parties. These loops are not unrolled at compile time; the type class `KnownSymbols` affords to the runtime environment sufficient knowledge of the type-level list.

```
sequenceP :: forall b (ls :: [Symbol]) m. (KnownSymbols ls, Monad m)
        => PIndexed ls (Compose m b) -> m (PIndexed ls b)

fanOut :: (KnownSymbols qs)
       => (forall q. (KnownSymbol q) => Member q qs
                                     -> Choreo ps m (Located (q ': common) a))
       -> Choreo ps m (Faceted qs common a)

scatter :: forall census sender recipients a m.
           (KnownSymbol sender, KnownSymbols recipients, Show a, Read a)
        => Member sender census
        -> Subset recipients census
        -> Located '[sender] (Quire recipients a)
        -> Choreo census m (Faceted recipients '[sender] a)
```

Figure 3.5: Type signatures for `sequenceP`, `fanOut`, and `scatter`.

The type-level programming necessary to use `sequenceP` and `PIndexed` directly can involve some boilerplate. We provide the derived functions `fanOut` and `fanIn` which suffice for every situation studied so far. `fanOut`'s argument is a choreography that results in a `Located` value at the party identified by the loop variable; it aggregates these results as a `Faceted`. `fanIn` is almost the same, except that the locations of the resulting values do not vary, and they are aggregated in a `Quire` located at some list of recipients. Figure 3.5 shows the type signatures for `sequenceP`, `fanOut`, and `scatter`. Keen readers may notice that the "`cmn`" parties' views of a `Faceted` are effectively just a `Quire`, and so wonder at the need for `fanIn`. In fact, `fanIn` *is* less often used than `fanOut`, but it's necessary for expressing choreographic loops that yield values which aren't known to the parties over whom the loop is defined. For example, the GMW protocol, which we implement using MultiChor in Section 3.5.1, cannot be written using only `fanOut`.

Modern Haskell language features, especially type-variable quantification, enable MultiChor's implementation of census polymorphism to be entirely type-safe and transparent to users. This is a flexible system within which users can easily write their own novel and bespoke functions and data structures.

3.5 Utility and Usability of MultiChor

We have already shown a few programs written in MultiChor, however these have been bespoke examples crafted for the purpose of demonstrating MultiChor. Section 3.5.1 describes our implementation of the Goldreich-Micali-Widgerson protocol for secure multiparty computation. Of the protocols we wrote during testing of MultiChor, we choose to showcase this one because:

- It is a pre-existing published protocol that other researchers have studied and implemented in non-choreographic systems.
- Our implementation is complete (in contrast to *e.g.* our implementation of the lottery mechanism from (Keeler et al. 2023), which would need further work to implement the complete the differentially-private secure aggregation protocol DPrio).
- Its implementation in any system is necessarily non-trivial because the protocol itself has some complexity.

In Section 3.5.2 we discuss other people's experience writing programs in MultiChor. No quantitative data on this subject exists yet, but we have received sufficient (negative) feedback to motivate some future work discussed in Chapter 5.

3.5.1 The GMW Protocol in MultiChor

Secure multiparty computation (Evans et al. 2018) (MPC) is a family of techniques that allow a group of parties to jointly compute an agreed-upon function of their distributed data without revealing the data or any intermediate results to the other parties. We consider an MPC protocol named Goldreich-Micali-Widgerson (GMW) (Goldreich et al. 2019) after its authors. The GMW protocol requires the function to be computed to be specified as a binary circuit, and each of the parties who participates in the protocol may provide zero or more inputs to the circuit. At the conclusion of the protocol, all participating parties learn the circuit's output.

The GMW protocol is based on two important building blocks: *additive secret sharing*, a method for encrypting distributed data that still allows computing on it, and *oblivious transfer* (OT) (Naor and Pinkas 2001), a building-block protocol in applied cryptography. The GMW protocol starts by asking each party to secret share its input values for the circuit. Then, the parties iteratively evaluate the gates of the circuit

while keeping the intermediate values secret shared. Oblivious transfer is used to evaluate AND gates. When evaluation finishes, the parties reveal their secret shares of the output to decrypt the final result.

```
gmw :: forall parties m. (KnownSymbols parties, MonadIO m, CRT.MonadRandom m)
    => Circuit parties -> Choreo parties (CLI m) (Faceted parties '[] Bool)
gmw circuit = case circuit of
  InputWire p -> do         -- process a secret input value from party p
    value :: Located '[p] Bool <- _locally p $ getInput "Your secret input value:"
    secretShare p value
  LitWire b -> do           -- process a publicly-known literal value
    let chooseShare :: forall p. (KnownSymbol p)
                    => Member p parties -> Choreo parties (CLI m) (Located '[p] Bool)
        chooseShare p = congruently (p @@ nobody) $ \_ -> case p of First  -> b
                                                                    Later _-> False
    fanOut chooseShare
  AndGate l r -> do         -- process an AND gate
    lResult <- gmw l; rResult <- gmw r;
    fAnd lResult rResult
  XorGate l r -> do         -- process an XOR gate
    lResult <- gmw l; rResult <- gmw r
    parallel (allOf @parties) \p un -> pure $ xor [viewFacet un p lResult,
                                                   viewFacet un p rResult]

data Circuit :: [LocTy] -> Type where
  InputWire :: (KnownSymbol p) => Member p ps -> Circuit ps
  LitWire :: Bool -> Circuit ps
  AndGate :: Circuit ps -> Circuit ps -> Circuit ps
  XorGate :: Circuit ps -> Circuit ps -> Circuit ps

mpc :: forall parties m. (KnownSymbols parties, MonadIO m, CRT.MonadRandom m)
    => Circuit parties -> Choreo parties (CLI m) ()
mpc circuit = do
  outputWire <- gmw circuit
  result <- reveal outputWire
  void $ _parallel (allOf @parties ) $ putOutput "The resulting bit:" result
```

This choreography works for an arbitrary number of parties. Figure 3.7 contains the `secretShare` choreography to handle an INPUT; Figure 3.8 shows the `fAnd` choreography to compute the result of an AND gate, and the choreography `reveal`. `xor` is a non-choreographic fold function. `mpc` uses `gmw` and `reveal`, and prints the resulting bit at each party.

Figure 3.6: A choreography for the GMW protocol.

Additive secret sharing We begin by describing additive secret sharing, a common building block in MPC protocols. A secret bit x can be *secret shared* by generating n random *shares* $s_1, \ldots, s_n$ such that $x = \sum_{i=1}^{n} s_i$. If $n - 1$ of the shares are generated uniformly and independently randomly, and the final share is chosen to satisfy the property above, then the shares can be safely distributed to the n parties without revealing x—

```
secretShare :: forall parties p m. (KnownSymbols parties, KnownSymbol p, MonadIO m)
            => Member p parties -> Located '[p] Bool
            -> Choreo parties m (Faceted parties '[] Bool)
secretShare p value = do
  shares <- locally p \un -> genShares p (un singleton value)
  PIndexed fs <- scatter p (allOf @parties) shares
  return $ PIndexed $ Facet . othersForget (First @@ nobody) . getFacet . fs

genShares :: forall ps p m. (MonadIO m, KnownSymbols ps)
          => Member p ps -> Bool -> m (Quire ps Bool)
genShares p x = quorum1 p gs'
  where gs' :: forall q qs. (KnownSymbol q, KnownSymbols qs)
            => m (Quire (q ': qs) Bool)
        gs' = do freeShares <- sequence $ pure $ liftIO randomIO--n-1 random shares
                 return $ qCons (xor (qCons @q x freeShares)) freeShares
```

`secretShare` handles Input gates by secret sharing `p`'s secret value among `parties`. `genShares` uses `Quire` to map each member `p` in `ps` to a generated secret share `Bool`.

Figure 3.7: Helper functions for the GMW protocol (1 of 2).

recovering x requires access to all n shares. Importantly, secret shares are *additively homomorphic*—adding together shares of secrets x and y produces a share of $x + y$.

MultiChor choreographies for performing secret sharing in the arithmetic field of booleans appear in Figures 3.7 and 3.8. The function `secretShare` takes a single secret bit located at party `p`, generates `shares`, a `Quire` which maps each member in `parties` to a share, and then uses `scatter` to send the assigned share to each member. However `scatter` would return a `Faceted parties '[p] Bool` since by default it includes the sender. The choreographic function `gmw` expects shares of wires to be secret, so we must return a `Faceted parties '[] Bool`. We accomplish this by deconstructing and reconstructing via `PIndexed`, and using `othersForget (First @@ nobody)`. The resulting `Faceted` "bit" actually represents the differing values located at all parties; the bits held by the parties sum up to the original secret. `reveal` takes exactly such a shared value and broadcasts all the shares so everyone can reconstruct the plain-text.

Oblivious transfer The other important building block of the GMW protocol is oblivious transfer (OT) (Naor and Pinkas 2001). OT is a 2-party protocol between a *sender* and a *receiver*. In the simplest variant (*1 out of 2* OT, used in GMW), the sender inputs two secret bits b_1 and b_2, and the receiver inputs a single secret *select*

```haskell
fAnd :: forall parties m.
        (KnownSymbols parties, MonadIO m, CRT.MonadRandom m)
     => Faceted parties '[] Bool
     -> Faceted parties '[] Bool
     -> Choreo parties (CLI m) (Faceted parties '[] Bool)
fAnd uShares vShares = do
  let genBools = sequence $ pure randomIO
  a_j_s :: Faceted parties '[] (Quire parties Bool) <- _parallel (allOf @parties)
                                                                 genBools
  bs :: Faceted parties '[] Bool <- fanOut \p_j -> do
      let p_j_name = toLocTm p_j
      b_i_s <- fanIn (p_j @@ nobody) \p_i ->
        if toLocTm p_i == p_j_name
          then _locally p_j $ pure False
          else do
              bb <- locally p_i \un -> let a_ij = getLeaf (viewFacet un p_i a_j_s)
                                                          p_j
                                           u_i = viewFacet un p_i uShares
                                       in pure (xor [u_i, a_ij], a_ij)
              conclaveTo (p_i @@ p_j @@ nobody)
                         (listedSecond @@ nobody)
                         (ot2 bb $ localize p_j vShares)
      locally p_j \un -> pure $ xor $ un singleton b_i_s
  parallel (allOf @parties) \p_i un ->
    let computeShare u v a_js b = xor $ [u && v, b]
                                     ++ toList (qModify p_i (const False) a_js)
    in pure $ computeShare (viewFacet un p_i uShares) (viewFacet un p_i vShares)
                           (viewFacet un p_i a_j_s)   (viewFacet un p_i bs)

ot2 :: (KnownSymbol sender, KnownSymbol receiver, MonadIO m, CRT.MonadRandom m)
    => Located '[sender] (Bool, Bool) -> Located '[receiver] Bool
    -> Choreo '[sender, receiver] (CLI m) (Located '[receiver] Bool)
ot2 bb s = do
  let sender = listedFirst :: Member sender '[sender, receiver]
  let receiver = listedSecond :: Member receiver '[sender, receiver]

  keys <- locally receiver  \un -> liftIO $ genKeys $ un singleton s
  pks <- (receiver, \un -> let (pk1, pk2, _) = un singleton keys
                           in return (pk1, pk2)) ~~> sender @@ nobody
  encrypted <- (sender,
                \un -> let (b1, b2) = un singleton bb
                       in liftIO $ encryptS (un singleton pks) b1 b2
               ) ~~> receiver @@ nobody
  locally receiver  \un -> liftIO $ decryptS (un singleton keys)
                                             (un singleton s)
                                             (un singleton encrypted)

reveal :: forall ps m. (KnownSymbols ps) => Faceted ps '[] Bool -> Choreo ps m Bool
reveal shares = xor <$> (gather ps ps shares >>= naked ps)
  where ps = allOf @ps
```

`fANd` computes the result of an AND gate on secret-shared inputs using pairwise oblivious transfer. The choreography works for an arbitrary number of parties, and leverages 1-out-of-2 OT. `ot` performs 1-out-of-2 oblivious transfer (OT) using RSA public-key encryption. The choreography involves exactly two parties, `sender` and `receiver`. `encryptS` `decryptS` (which are omitted for brevity) use the cryptonite library for encryption and decryption. In `reveal`, all parties broadcast their shares of the value to each other, the gathered shares are `xor`ed to compute the plaintext result.

Figure 3.8: Helper functions for the GMW protocol (2 of 2).

bit s. If $s = 0$, then the receiver receives b_1; if $s = 1$, then the receiver receives b_2. Importantly, the sender does *not* learn which of b_1 or b_2 has been selected, and the receiver does *not* learn the non-selected value.

Oblivious transfer is a *two-party protocol*; it would be erroneous for any third-parties to be involved in the implementation. MultiChor's `Faceted` values and utilities for type-safe embedding of conclaved sub-protocols within larger censuses make it possible to embed the use of pairwise oblivious transfer between parties in a general version of multi-party GMW.

Computing secret-shared AND gates Computing the result of an AND gate using only additively-homomorphic secret shares requires $n^2 - n$ 1-out-of-2 OT procedures. This process is described in (Prabhakaran 2023); here we follow their example in writing addition and multiplication expressions to represent XOR and AND operations respectively. Supposing the plaintext relation $w = uv$, the goal is for each party i to acquire a secret share w_i of w without revealing their secret shares of u and v to the other parties. For each of the $n^2 - n$ ordered pairs of distinct participants, two values are computed: a_{ij} known to party i and b_{ij} known to party j.

- i chooses a_{ij} randomly.
- i and j make $b_{ij} = u_i v_j - a_{ij}$ using OT, where v_j is j's select bit and the two secrets offered by i are the two possible values of b_{ij}.

Each party i then sets $w_i = u_i v_i + \sum_{j \neq i} a_{ij} + \sum_{j \neq i} b_{ji}$. Algebraically:

$$
\begin{aligned}
\sum_i w_i &= \sum_i \left(u_i v_i + \sum_{j \neq i} a_{ij} + \sum_{j \neq i} b_{ji} \right) \\
&= \sum_i \left(u_i v_i + \sum_{j \neq i} a_{ij} + \sum_{j \neq i} \left(u_j v_i - a_{ji} \right) \right) && \text{expand } b_{ji} \\
&= \sum_i \left(u_i v_i + \sum_{j \neq i} u_j v_i \right) + \sum_i \left(\sum_{j \neq i} a_{ij} - \sum_{j \neq i} a_{ji} \right) && \text{rearrange summations} \\
&= \sum_i v_i \left(u_i + \sum_{j \neq i} u_j \right) && \text{cancel terms \& factor} \\
&= \sum_i v_i u = u \sum_i v_i = uv = w && \text{by definition of secret shares}
\end{aligned}
$$

Therefore the resulting values w_i are a valid secret shares of the output wire value w. When the parties actually compute their shares in our implementation, rather than skipping the $i = j$ cases they simply force all a_{ii} and b_{ii} to be 0 (`False`). The implementing choreography is `fAnd` (Figure 3.8 lines 1–27); it takes `Faceted` values holding the parties' shares of the input values, and returns a `Faceted` value representing each party's share of the output. On line 10, the parties perform a `fanOut` to begin the pairwise computation. The `fanIn` on line 12 completes the pairing, and uses `conclaveTo` (line 20) to embed pairwise OTs (via `ot2` , defined on line 33) in the larger set of parties. The variable names used in Figure 3.8 attempt to reflect the homogeneous perspective of MLVs, quires, and faceted values, but no best-practices for variable naming in this context have been established yet.

The GMW protocol The complete GMW protocol operates as summarized earlier, by secret sharing input values and then evaluating the circuit gate-by-gate. Our implementation as a MultiChor choreography appears in Figure 3.6, defined as a recursive function over the structure of the circuit. The choreography returns a `Faceted` value, representing the secret-shared output of the circuit. For "input" gates (lines 4–6), the choreography runs the secret sharing protocol in Figure 3.7 to distribute shares of the secret value. For XOR gates (Figure 3.6 lines 16–18), the parties recursively run the GMW protocol to compute the two inputs to the gate and then each party computes one share of the gate's output by XORing their shares of the inputs. This approach leverages the additive homomorphism of additive secret shares. For AND gates (lines 13–15), the parties compute shares of the gate's inputs, then use the `fAnd` protocol to perform multiplication of the two inputs. Since additive secret shares are not multiplicatively homomorphic, this operation leverages the oblivious transfer protocol to perform the multiplication. This implements the many-party extension of the protocol as described in (Evans et al. 2018, Prabhakaran 2023).

Our implementation of GMW leverages MultiChor's `Faceted` values and utilities for type-safe parallel, conclaved, and pairwise choreographies to build a fully-general implementation of the protocol that works for an arbitrary number of parties.

3.5.2 User challenges in MultiChor

Industry use of CP concepts remains nascent, but enough embedded or semi-embedded implementations now exist or are in development that prospective users will need to actively choose between them. Just within

the Haskell ecosystem, it's possible that an engineer might accept the excess communication necessitated by HasChor's KoC strategy in order to avoid the conceptual (and textual) overhead of census tracking. Indeed, although we know of no "in the wild" use of MultiChor, anecdotal reports from academic peers who have attempted to use the library suggest it would benefit from substantial further cosmetic work, and may need theoretical breakthroughs to appeal to non-academic developers.

A few people that we know of (besides ourselves) have actually attempted to write programs using MultiChor. A couple of our fellow students accepted our invitation to attempt a programming challenge modeled after a job-interview exercise. The exercise itself is described in Appendix B. These sessions were not structured as a controlled usability study; volunteers were invited to ask for help with any part of the exercise they wished. Nonetheless, none of the volunteers were able to implement the described protocol, which had been designed both for brevity and to exactly fit MultiChor's capabilities. This was our most detailed source of feedback. Separately, we also received feedback on the software artifact (Bates et al. 2025a) submitted alongside (Bates et al. 2025b).

One point of feedback has been practically unanimous: MultiChor's existing documentation is insufficient. Relying on type signatures to communicate behavior presupposes familiarity with MultiChor's types, and the textual documentation, however systematic, is not suitable for bootstrapping a new user's understanding. The example choreographies included in the MultiChor repository are not presented as a form of documentation, and therefor do not serve that purpose.

The proof-witnesses system One major hurdle to writing correct choreographies with MultiChor is constructing and managing the proof-witness arguments as described in Section 3.3. Regardless of whether or not the system is overly-complicated (to quote both the volunteers from the usability exercise: *"It's kinda complicated."*) the cognitive load of using the proof-witness system is *additional* to the complexity of writing the actual choreography. In other words, a user must consider both how to represent a choreographic behavior using MultiChor's operators and how to prove that the relevant parties have the relevant memberships, and because the proofs serve double-duty as identifiers, the user must think about those two problems simultaneously.

"Compute this" operators MultiChor offers three "basic" operators for embedding non-choreographic computation in a choreography: `locally`, `congruently`, and `parallel`. Each of these is derived from

more primitive forms, and each has further derived forms (*e.g.* `_locally_`, `purely`). Choosing the best of these options for any given task is worse than a needle-in-haystack problem, because multiple of them may actually work, and because some may appear to work for the immediate task while causing problems later in the program.

The `CanSend` type class To send a value in MultiChor one must (in addition to specifying the recipients) provide a MLV and prove that the sender both owns the value and is present in the census. This can be quite repetitive. To minimise boiler-plate, the surface API of MultiChor uses a class `CanSend` so that the broad- and multi-cast functions can take the proof arguments in different formats. Unfortunately, by offering so many options to a user, MultiChor's API obfuscates the simple question of how to actually write a communication statement.

At a broader level, we observe that the existing implementation-focused module structure, which separates operators across three modules ("core", "surface", and "batteries"), fails to guide new users to the tools they need to do their work, contributes nothing to the management of `import` statements, and is of dubious benefit even to the users who know the system best: ourselves. We will discuss practical implications of the above points in Chapter 5.

Bibliography

Austin, T. H. and C. Flanagan (2012). Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, New York, NY, USA, pp. 165–178. Association for Computing Machinery.

Bates, M., S. Kashiwa, S. Jafri, G. Shen, L. Kuper, and J. Near (2025a). Efficient, portable, census-polymorphic choreographic programming. `https://doi.org/10.5281/zenodo.15048718`. Archival Artifact.

Bates, M., S. Kashiwa, S. Jafri, G. Shen, L. Kuper, and J. P. Near (2025b, June). Efficient, portable, census-polymorphic choreographic programming. *Proc. ACM Program. Lang. 9*(PLDI). Archive: `https://arxiv.org/abs/2412.02107`.

Beck, G., A. Goel, A. Hegde, A. Jain, Z. Jin, and G. Kaptchuk (2023). Scalable multiparty garbling. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, New York, NY, USA, pp. 2158–2172. Association for Computing Machinery.

Bonawitz, K., V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth (2017). Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1175–1191.

Corrigan-Gibbs, H. and D. Boneh (2017). Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX symposium on networked systems design and implementation (NSDI 17)*, pp. 259–282.

Cruz-Filipe, L. and F. Montesi (2016). Choreographies in practice. In E. Albert and I. Lanese (Eds.), *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings*, Volume 9688 of *Lecture Notes in Computer Science*, pp. 114–123. Springer.

Evans, D., V. Kolesnikov, M. Rosulek, et al. (2018). A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security 2*(2-3), 70–246.

Goldreich, O., S. Micali, and A. Wigderson (2019). How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pp. 307–328.

Keeler, D., C. Komlo, E. Lepert, S. Veitch, and X. He (2023, 07). Dprio: Efficient differential privacy with high utility for prio. *Proceedings on Privacy Enhancing Technologies 2023*, 375–390.

Naor, M. and B. Pinkas (2001). Efficient oblivious transfer protocols. In *SODA*, Volume 1, pp. 448–457.

Noonan, M. (2018). Ghosts of departed proofs (functional pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, New York, NY, USA, pp. 119–131. Association for Computing Machinery. Archive: `https://kataskeue.com/gdp.pdf`.

Noonan, M. (2019). gdp: Reason about invariants and preconditions with ghosts of departed proofs. `https://hackage.haskell.org/package/gdp-0.0.3.0`.

Prabhakaran, M. M. (2023, August). Advanced tools from modern cryptography: Lecture 6. `https://www.cse.iitb.ac.in/~mp/teach/advcrypto/f23/slides/06.pdf`. Archive: `https://web.archive.org/web/20250708141313/https://www.cse.iitb.ac.in/~mp/teach/advcrypto/f23/slides/06.pdf`.

Shen, G., S. Kashiwa, and L. Kuper (2023, aug). Haschor: Functional choreographic programming for all (functional pearl). *Proc. ACM Program. Lang. 7*(ICFP).

Wu, W., L. He, W. Lin, R. Mao, C. Maple, and S. Jarvis (2020). Safa: A semi-asynchronous protocol for fast federated learning with low overhead. *IEEE Transactions on Computers 70*(5), 655–668.

Chapter 4

MiniChor is MultiChor, Just Smaller

λ_C and the associated theorems demonstrate the theoretical soundness of the conclaves-&-MLVs CP paradigm, and MultiChor demonstrates that it is practical to implement and use that paradigm. That said, it is unsatisfying that the syntactic structures of these two systems are so different from each other. Furthermore, it is not clear that these systems as they stand are good foundations for the development of more advanced CP techniques: the λ_C system is not readily adaptable for proving the safety of further extensions, and MultiChor has some known usability problems as discussed in Section 3.5.2. As a step toward future practical and theoretical developments, we developed MiniChor (Bates 2025), a research-prototype fork of MultiChor. MiniChor is able to express all the same choreographies (see caveat in Section 4.3) using a parred-down core API which we believe is simple enough to directly model in a formalism. MiniChor also demonstrates some immediate insights that will affect the design of future MultiChor versions.

Of particular note is that MiniChor does not feature located values (multiply or singly) as understood by prior CP systems. While the type `Located` appears in MiniChor and is used similarly to the type of the same name in MultiChor, in MiniChor `Located` is just an alias for `Choreo`! The theoretical implication of this is that MLVs are actually just a special case of census-annotated choregraphies. We discuss further implications in Section 4.4.

Most of this chapter will describe the differences between MultiChor and MiniChor narratively. The first change is to remove the freer-monad system and instead implement `Functor`, `Applicative`, and `Monad` for `Choreo` directly. This has no effect on the rest of the system or on the case studies; it's simply a moving

part which we have the ability to remove[1]. Second, we remove `othersForget` and `flatten` from the core API and re-implement them as monadic operations in the surface API using `congruently'`. This requires some small changes in the case studies; *e.g.*

```
do result <- (listedFirst,
              alice @@ nobody,
              flatten aliceInConclaveA aliceinConclaveB value) ~> bob @@ nobody
   return result
```

becomes

```
do value' <- flatten aliceInConclaveA aliceinConclaveB value
   result <- (listedFirst, alice @@ nobody, value') ~> bob @@ nobody
   return result
```

Third, we remove the type parameter `m` (for monad) from `Choreo` and simply assume that the local monad is always `CLI IO`. This is basically the same as just `IO`, and use-cases for local monads that *aren't* basically just `IO` seem uncommon. The rest of the changes descried in this chapter are more impactful, but the process follows a similar pattern of refactoring the core API and then either shimming the difference in the surface API (so that the exposed system behaves the same) or propagating semantically-inconsequential changes into the case studies. Most of the case studies have robust unit tests based on them, to detect any mistakes during this process.

4.1 Monadic Unwrapping

We replace the core operator `congruently'` with `naked`, simplifying the core API.

HasChor enforces the rule that only the owner of a located value may call `unwrap` on it by hiding `unwrap` in a module (only its type, `Unwrap` is exported) and affording it to users only as an argument to `locally`'s callback argument. MultiChor uses the exact same pattern, but a design goal was to also represent *pure* computation actively replicated across the owners of the relevant MLVs. The way MultiChor

[1]The main selling point of freer monads is how they compose with each other, and how little boiler-plate is needed when writing them. Neither of these are needed for MiniChor. There may also be performance considerations; the need for methods for comparing the performance of CP systems was acknowledged by the community of CP researchers attending PLDI24.

does this is by duplicating the `locally Unwrap` pattern to make `congruently Unwraps`, as shown in Figure 4.1**(a)**.

An alternative to `congruently'` (which actively replicates a pure computation using MLVs known to the entire census) is `naked`, which unwraps a single MLV known to the entire census. The two strategies are equivalent in what they can express, but `naked` has the disadvantage that it can't be adapted for use in a larger census as ergonomically as `congruently'` can; the equivalent of the un-primed `congruently` in a `naked`-based system is a family of functions for each fixed N that each handle pure computations on N arguments. Since MiniChor doesn't care about ergonomics, this is acceptable.

The advantage of replacing `congruently'` with `naked` is that it can also replace the call-back pattern of `locally`. This intermediate system is shown in Figure 4.1**(b)**.

```
type Unwrap (q :: LocTy) = forall ls a. Member q ls -> Located ls a -> a
type Unwraps (qs :: [LocTy]) = forall ls a. Subset qs ls -> Located ls a -> a

locally'     :: (KnownSymbol  l ) => (Unwrap  l  -> CLI IO a) -> Choreo '[l] a
congruently' :: (KnownSymbols ls) => (Unwraps ls -> a       ) -> Choreo ls   a

naked :: (KnownSymbols ps) => Subset ps qs -> Located qs a -> Choreo ps a
naked                         ownership       a            =
                                          congruently' (\un -> un ownership a)
```

(a) The MultiChor approach. The two types `Unwrap` and `Unwraps` are used as the argument types in callback functions used by `locally'` and `congruently'`. `naked` in this system is a derived function.

```
locally'     :: (KnownSymbol  l ) => (              CLI IO a) -> Choreo '[l] a
naked :: (KnownSymbols ps) => Subset ps qs -> Located qs a -> Choreo ps a

congruentlyN :: (KnownSymbols ls) => Subset ls census ->
  (Subset ls owners1, Located owners1 arg1) ->
  ... ->
  (Subset ls ownersN, Located ownersN argN) ->
  (arg1 -> ... -> argN -> a) ->
  Choreo census (Located ls a)
congruentlyN present (owns1, arg1) ... (ownsN, argN) f =
    conclave present $ f <$> naked owns1 arg1 <*> ... <*> naked ownsN argN
```

(b) The `naked`-based approach described in Section 4.1. In this system, `locally'` just lifts local monadic effects (`CLI IO`) into singleton choreographies (which can be conclaved). The pseudo-code `congruentlyN` shows how, for any fixed number N of MLVs that will be used in the pure computation, an analog of `congruently` can be written. A similar pattern for `locallyN` is not shown.

Figure 4.1: Different strategies for local effects and pure active replication.

4.2 MLVs as quantified functions

In the `naked`-based system of Figure 4.1**(b)**, `naked` is the only means by which the actual value of an MLV can be accessed. This suggests removing `naked` from the foundational signature of `Choreo`, and instead making it the actual definition of `Located`. Figure 4.2**(b)** shows this change.

A design pattern of MultiChor was that the `Core` module needed to be "trusted"; our own reasoning outside of Haskell's type system is what guarantees that no user working outside of `Core` can call `unwrap` on `Empty`. None of our changes in the MiniChor fork alter this pattern; even the nature of the invariant we're maintaining is the same: That a party will never compute on an MLV they don't own. The changes in Figures 4.2 and 4.3 are only to where the impossible error lives, from the case-wise definition of `unwrap` to an undefined value returned by a choreography generated at runtime (Figure 4.2**(b)** line 16).

The point of the change described in Figure 4.2**(b)** is to make intuitive the remaining jumps to MiniChor, a core API for choreographic programming that doesn't have located values at all!

4.3 MLVs *are* Choreographies

We get rid of MLVs by relaxing our demands of them. Previously it sufficed for one or more owners of an MLV to be present in a census for them to do something with that value, but now we will require that *all* owners be present. In terms of implementation, we demote `Located` from a `newtype` wrapper around a function down to just a type alias for `Choreo` (Figure 4.3 line 1).

To understand the conceptual difference, consider some formal DSL of no specific purpose: the syntax of expressions in that language contains as a subset its syntax of values. In other words, 5 is a computation that happens to evaluate to the same thing as $2+3$. Similarly, in our earlier model $\boldsymbol{\lambda_C}$, $5@p^+$ is a computation that evaluates by p^+ to five; we promise that no-one not in p^+ will attempt to evaluate it, and such non-owning parties replace it with $\perp$ at runtime. Any (multiply) located value like $5@p^+$ can be perfectly represented by a choreography which

- has exactly p^+ as its census and
- evaluates to the (not located) value "five".

```haskell
-- MLVs are secretly isomorphic to Maybe values.
data Located ls a = Wrap a | Empty

-- Core operations
locally' :: ... ;  naked :: ... ;  broadcast' :: ... ;  conclave :: ...

-- Endpoint Projection, selected cases
epp (Naked ownership value) self = let unwraps = ...
                                   in pure $ unwraps ownership value
epp (Broadcast sender (ownership, value)) self
    | self == toLocTm sender = do let val = unwrap ownership value
                                  Send val otherRecipients
                                  pure val
    | otherwise              = Recv $ toLocTm sender
epp (Conclave subcensus choreo) self
    | self `elem` toLocs subcensus = Wrap <$> epp choreo self
    | otherwise                    = pure Empty
epp (...) self = ...
```

(a) The `naked`-based system from Section 4.1 and Figure 4.1**(b)**.

```haskell
-- MLVs are explicitly choreographies quantified over subsets of their owners.
newtype Located ls a = Located {
    naked :: forall census. Subset census ls -> Choreo census a
}

-- Core operations, naked has moved
locally' :: ... ;                  broadcast' :: ... ;  conclave :: ...

-- Endpoint Projection, selected cases
epp (Broadcast sender (ownership, value)) self
    | self == toLocTm sender = do val <- epp (naked value (ownership @@ nobody)) self
                                  Send val otherRecipients
                                  pure val
    | otherwise              = Recv $ toLocTm sender
epp (Conclave subcensus choreo) self
    | self `elem` toLocs subcensus = do val <- epp choreo self
                                        pure $ Located \_ -> pure val
    | otherwise = pure $ Located \_ -> pure undefined
epp (...) self = ...
```

(b) Another intermediate system described in Section 4.2. The data constructors `Wrap` and `Empty` (**(a)** line 1) and the AST form `Naked` (**(a)** line 6), are absent. Instead, `naked` is the accessor of the data type `Located`, which wraps a function from proof of ownership to a choreography over the specified subset of the owners (line 2). At runtime, the placeholder used for remote MLVs is a choreography that returns `undefined` (an error) (line 16). In practice one's own MLVs will be represented at runtime by ASTs for trivial choreographies (*e.g.* `Return 5`), this is what `pure` does in the `Choreo` monad (line 15).

Figure 4.2: Under-the-hood implementation changes for redefining MLVs out of existence. *(1/2)*

```haskell
-- MLVs are literally choreographies; the owners are the census.
type Located ls a = Choreo ls a

-- Core operations, naked is gone and there are other changes.
locally' :: (...)   => CLI IO a -> Choreo '[l] a
broadcast' :: (...) => Member sender ps -> Located '[sender] a -> Choreo ps a
conclaveTo :: (...) => Subset subcensus census -> Subset owners subcensus
                    -> Choreo subcensus (Located owners a)
                    -> Choreo census (Located owners a)

-- Endpoint Projection, selected cases
epp (Broadcast sender value) self
    | self == toLocTm sender = do val <- epp value self
                                  Send val otherRecipients
                                  pure val
    | otherwise              = Recv $ toLocTm sender
epp (Conclave subcensus owners choreo) self
    | self `elem` toLocs subcensus = epp choreo self
    | otherwise = pure $ pure undefined
epp (...) self = ...
```

The MiniChor system described in Section 4.3. Here `Located` is just an alias for `Choreo`. As discussed in Section 4.3, this requires swapping `conclave` for `conclaveTo` and changing the signature of `broadcast'`. The implementation of EPP is basically the same, there's just no construction or unwrapping of located values; `naked` no longer exists.

Figure 4.3: Under-the-hood implementation changes for redefining MLVs out of existence. *(2/2)*

Giving up the ability to use a "located value" when not all of its owners are present has two big effects on the overall system. First, reusable software components can no longer take arguments with open-ended polymorphic ownership sets; an MLV is useless without proof that all its owners are present in the choreography. `othersForget` can still be used to reduce ownership sets, but it now needs all the original owners to also be present. This implies a small **caveat** on the assertion that MiniChor can do anything MultiChor can do: Higher-order choreographies in MiniChor enjoy slightly less-open-ended polymorphic typing than in MultiChor. Reusable components must be restrictive (instead of lenient) about the owners of their arguments. In the case studies, we were still able to preserve all the same behavior by applying `othersForget` one or more layers up in the program's architecture from where the values in question are actually used.

A more fundamental change is that it is no longer possible to write the function `flatten`. Consider its hypothetical type signature:

```haskell
flatten :: (KnownSymbols ls) =>
  Subset newOwners census -> Subset newOwners outer -> Subset newOwners inner ->
  Located outer (Located inner a) -> Choreo census (Located newOwners a)
```

An implementation would take as an argument a `Located outer (...)`; in order to *use* that it would have to conclave to `outer`. Inside the conclave, it would have a `Located inner a`, but there'd be nothing it could do with it because there'd be no proof that all of `inner` are present in `outer`. There may be multiple solutions to this problem; MiniChor's solution is to make `flatten` unnecessary by replacing the core operation `conclave` with `conclaveTo`, who's body-argument is required to return a located value, and which does not add a layer of location-wrapping (Figure 4.3 line 6). (In MultiChor, `conclaveTo` is a derived function using `flatten`. In MiniChor, `conclave` is a derived function using monad-bind.)

4.4 Implications

Although MiniChor was not intended to ever see real-world use, the usability trade-offs between it and MultiChor are not obvious, and we can learn several things from it which can guide the development of a future CP frameworks. Most significantly, MLVs are emergent rather than fundamental to choreographic programming!

Many prior CP systems (*e.g.* Pirouette (Hirsch and Garg 2022)) have featured syntactic distinctions between "global" choreographic code and local code; these systems feature dualities between choreographic variables and local variables (which have their own namespaces) and choreographic and local functions (which have distinct syntax). In contrast, in MiniChor choreographies are first-class values in the same syntax and namespace as all other Haskell code; one can (and does) write values with types like `Choreo ps (Choreo qs Int)`. Other prior systems (*e.g.* Chorλ (Cruz-Filipe et al. 2022)) keep everything in a single "layer", all values are located and all computation is choreographic. Such systems still maintain distinct senses in which a value can be located: data is annotated with owners and functions are annotated with participants; these annotations are handled differently by the typing and semantic rules. Because MiniChor privileges the un-located ("naked") level of computation and treats choreographies as first-class values, it is able to use the single syntactic construct `conclaveTo` (and the corresponding typing property, censuses) as the sole ascription of locality.

That located values are (or can be) an emergent pattern in choreographic programming is a significant theoretical insight on its own. Whether or not this pattern can be replicated for other styles of CP (*e.g.* select-&-merge) is beyond the scope of this work. Here we explore three ways the concepts used in MiniChor may affect the future development of MultiChor.

First, as was the original intention, MiniChor is a minimalist implementation; sufficiently concise in its inner workings to be targeted by a formal model. The `Choreo` data type (the ASTs of choreographies) has three important constructors, plus the `Return` and `Bind` constructors it needs to implement `Functor` and `Monad`. Each of the constructors individually is simple; excepting the proof witnesses they each take a single argument. Two concise functions `epp` and `runChoreo` implement the distributed and centralized semantics, respectively. We leave as future work to compose a formal model of MiniChor, and to prove theorems about it (especially that it enjoys some equivalence with a corresponding select-&-merge system).

Second, while the actual implementation of MiniChor doesn't use any laziness beyond what's normal for Haskell programs, the use of *choreographies* as the arguments and return-types of the choreographic operators makes laziness-based programming patterns immediately available to users. Specifically, while the MLVs that arise naturally in naïve use of MiniChor are generally trivial `Return` ASTs, nothing about the type system forbids passing a non-trivial choreography into any function that consumes an MLV. One can write a `lazyMulticast x` that performs no immediate action itself but just returns (as the "MLV") the choreography `broadcast x`. It would be worthwhile to explore the utility, performance, and limitations of such a system, and consider recapitulating it in MultiChor.

Third, MLVs are functors! (Specifically, they are endofunctors, which is what's meant by the Haskell type class `Functor`. They are also instances of `Applicative` and `Monad`.) This insight can be immediately ported back to MultiChor: since MultiChor's representation of `Located` is isomorphic to `Maybe`, the class implementations are straightforward; all that was missing was confidence that those interfaces were safe to expose, and MiniChor gives us that. For any construct in Haskell to implement these classes is a major usability advantage (Sajanikar 2017, Chapter 4). As an example, a MultiChor declaration of a variable `filtered` like

```
do let myFilter :: (Key->Widgit->Bool) -> Key -> [Widgit] -> [Widgit] = ...
       comparator :: Located servers (Key->Widgit->Bool) = ...
       key :: Located workers Key = ...
       values :: Located workers [Widgit] = ...
   filtered <- congruently (transitive workers servers) \un ->
         myFilter (un workers comparator) (un refl key) (un refl values)
   ...
```

can be rewritten using "map" (`<$>` from `Functor`) and "splat" (`<*>` from `Applicative`) as

```
do let ...
   let filtered = myFilter <$> (othersForget workers comparator)
                           <*> key
                           <*> values
   ...
```

In MiniChor's case studies this pattern mostly replaces the analogs of `congruently`; it's less clear if this pattern improves or degrades the ergonomics of `locally` and its variants.

We don't present MiniChor as "MultiChor-V2" because it's advantages or disadvantages, compared to MultiChor in terms of performance or ergonomics, are ambiguous. Although precisely assessing the usability of a system like MultiChor will require a future human subjects study, it's plausible that that the advantages of the above pattern could more-than-offset the deficits of a `naked`-based system. For this reason, we suggest including MiniChor or a system like it in any such future usability study.

Bibliography

Bates, M. (2025, 4). Minichor. https://github.com/ShapeOfMatter/MiniChor/tree/4d36b477ea85f4a168a99f2b83534c1a14fb0a2c.

Cruz-Filipe, L., E. Graversen, L. Lugović, F. Montesi, and M. Peressotti (2022, September). *Theoretical Aspects of Computing*, Volume 13572 of *Lecture Notes in Computer Science*, Chapter Functional choreographic programming, pp. 212–237. Tbilisi, Georgia: Springer. Archive: https://arxiv.org/abs/2111.03701.

Hirsch, A. K. and D. Garg (2022, January). Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang. 6*(POPL).

Sajanikar, Y. (2017). *Haskell Cookbook: Build functional applications using Monads, Applicatives, and Functors*. Packt Publishing.

Chapter 5

Conclusions

In Chapter 2 we introduced a new paradigm for choreographic programming, showed that it has the basic properties foundational to the field, and described how to replicate the expressivity and efficiency of other paradigms in this new *conclaves-&-MLVs* paradigm. In Chapter 3 we showed how to implement library-level CP with conclaves and MLVs in Haskell, and showed how Haskell's type system enabled CP design patterns like census polymorphism that were previously unavailable. In Chapter 4 we demonstrated that some aspects of the MultiChor system are extraneous; the CP system MiniChor can represent all of the same case-study programs without using freer monad handlers or having a built-in notion of located values.

Our intention in writing λ_C and MultiChor was to build a CP system that was easier to reason about and faster for new users to learn, but we do not yet have a quantitative assessment of our success in this dimension. What we have shown is that our systems are the first CP systems in which the well-formed-ness of programs is entirely type-directed. Furthermore, MultiChor is the first implementation to support census polymorphism; we demonstrated the utility of this feature by example.

While we believe the advantages of MultiChor to be unique at this time, we do not expect the field to remain stagnant and there are critical things MultiChor choreographies can not do. Furthermore, our experience observing other people attempt to use MultiChor suggests that its flexibility may not suffice, in the minds of prospective users, to justify its corresponding cognitive overhead. Substantial work remains to do to make choreographic programming an attractive paradigm for industry users. Our hope at this time is that MultiChor and MiniChor are well built and well positioned as stepping-off points for such future work.

5.1 Suggestions for Future Work

In Sections 3.5 and 4.4 we suggested immediate changes that could be made to MultiChor. For version-2.0 *per se*, we advocate incorporating as many of them as practical in a timely fashion.

- Identify a limited curated set of functions, values, and types to to expose to end users, so that users can more easily find the relevant tools for their tasks.
- Restructure the modules to better reflect how we expect the library to be used. Ideally, everything used for writing choreographies should go in one module and everything used for running them should go in another module, and there should be no other modules for users to worry about.
- *Either*
  - find a better way to represent, enforce, and satisfy membership and subset constraints, or
  - separate the set-membership proof witness system into its own (well documented) library that's completely agnostic of the particular use to which MultiChor puts it.
- Improve the documentation. Include a narrative tutorial in the landing page and include examples of how to use most of the important functions.
- Provide `instance`s for `Located` of `Functor`, `Applicative`, and `Monad`.
- *Possibly,* encourage the use of (or migrate the whole API to use) `naked`-based active replication instead of `congruently`.
- Furthermore, MultiChor should be augmented with some mechanism, however inelegant, for representing fallible communication.

Further future work constitutes a substantial and open-ended research campaign that might be tackled in any order:

- Compose a formal model of MiniChor and a comparable select-&-merge model, with the goal of showing an equivalence between them.

- Conduct a structured study comparing the usability of MultiChor, MiniChor, and other relevant systems that target real industry use.
- Conduct a structured study comparing the performance of MultiChor, MiniChor, and other relevant systems that target real industry use.
- Develop a system for lazy choreographies as discussed in Section 4.4.
- Explore options for dynamic census polymorphism in library level CP.
- Systematizing the increasingly diverse domain of multiparty programming techniques, and better illuminating the relationships between them.

In the meantime, we hope that both researchers and interested industry practitioners will see the work presented here as the cutting edge of applied choreographic programming, and as a suitable foundation for further development.

Bibliography

Austin, T. H. and C. Flanagan (2012). Multiple facets for dynamic information flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, New York, NY, USA, pp. 165–178. Association for Computing Machinery.

Baeten, J. (2004). *A brief history of process algebra*. Computer science reports. Technische Universiteit Eindhoven.

Bates, M. (2025, 4). Minichor. `https://github.com/ShapeOfMatter/MiniChor/tree/4d36b477ea85f4a168a99f2b83534c1a14fb0a2c`.

Bates, M., S. Kashiwa, S. Jafri, G. Shen, L. Kuper, and J. Near (2025a). Efficient, portable, census-polymorphic choreographic programming. `https://doi.org/10.5281/zenodo.15048718`. Archival Artifact.

Bates, M., S. Kashiwa, S. Jafri, G. Shen, L. Kuper, and J. P. Near (2025b, June). Efficient, portable, census-polymorphic choreographic programming. *Proc. ACM Program. Lang. 9*(PLDI). Archive: `https://arxiv.org/abs/2412.02107`.

Beck, G., A. Goel, A. Hegde, A. Jain, Z. Jin, and G. Kaptchuk (2023). Scalable multiparty garbling. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, CCS '23, New York, NY, USA, pp. 2158–2172. Association for Computing Machinery.

Bonawitz, K., V. Ivanov, B. Kreuter, A. Marcedone, H. B. McMahan, S. Patel, D. Ramage, A. Segal, and K. Seth (2017). Practical secure aggregation for privacy-preserving machine learning. In *proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1175–1191.

Carbone, M. and F. Montesi (2013). Deadlock-freedom-by-design: multiparty asynchronous global programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, New York, NY, USA, pp. 263–274. Association for Computing Machinery.

Castagna, G., M. Dezani-Ciancaglini, and L. Padovani (2011). On global types and multi-party sessions. In R. Bruni and J. Dingel (Eds.), *Formal Techniques for Distributed Systems*, Berlin, Heidelberg, pp. 1–28. Springer Berlin Heidelberg.

Chakraborty, K. (2024). Unichorn. `https://share.unison-lang.org/@kaychaks/unichorn/`.

Corrigan-Gibbs, H. and D. Boneh (2017). Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX symposium on networked systems design and implementation (NSDI 17)*, pp. 259–282.

Cruz-Filipe, L., E. Graversen, L. Lugović, F. Montesi, and M. Peressotti (2023). Modular Compilation for Higher-Order Functional Choreographies. In K. Ali and G. Salvaneschi (Eds.), *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, Volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Dagstuhl, Germany, pp. 7:1–7:37. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Cruz-Filipe, L., E. Graversen, L. Lugović, F. Montesi, and M. Peressotti (2022, September). *Theoretical Aspects of Computing*, Volume 13572 of *Lecture Notes in Computer Science*, Chapter Functional choreographic programming, pp. 212–237. Tbilisi, Georgia: Springer. Archive: `https://arxiv.org/abs/2111.03701`.

Cruz-Filipe, L. and F. Montesi (2016). Choreographies in practice. In E. Albert and I. Lanese (Eds.), *Formal Techniques for Distributed Objects, Components, and Systems - 36th IFIP WG 6.1 International Conference, FORTE 2016, Held as Part of the 11th International Federated Conference on Distributed*

Computing Techniques, DisCoTec 2016, Heraklion, Crete, Greece, June 6-9, 2016, Proceedings, Volume 9688 of *Lecture Notes in Computer Science*, pp. 114–123. Springer.

Cruz-Filipe, L. and F. Montesi (2017). Procedural choreographic programming. In A. Bouajjani and A. Silva (Eds.), *Formal Techniques for Distributed Objects, Components, and Systems - 37th IFIP WG 6.1 International Conference, FORTE 2017, Held as Part of the 12th International Federated Conference on Distributed Computing Techniques, DisCoTec 2017, Neuchâtel, Switzerland, June 19-22, 2017, Proceedings*, Volume 10321 of *Lecture Notes in Computer Science*, pp. 92–107. Springer.

Cruz-Filipe, L. and F. Montesi (2020). A core model for choreographic programming. *Theor. Comput. Sci. 802*, 38–66.

Evans, D., V. Kolesnikov, M. Rosulek, et al. (2018). A pragmatic introduction to secure multi-party computation. *Foundations and Trends® in Privacy and Security 2*(2-3), 70–246.

Giallorenzo, S., F. Montesi, and M. Peressotti (2024, jan). Choral: Object-oriented choreographic programming. *ACM Trans. Program. Lang. Syst. 46*(1).

Giallorenzo, S., F. Montesi, M. Peressotti, D. Richter, G. Salvaneschi, and P. Weisenburger (2021). Multiparty languages: The choreographic and multitier cases (pearl). In A. Møller and M. Sridharan (Eds.), *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, Volume 194 of *LIPIcs*, pp. 22:1–22:27. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

Goldreich, O., S. Micali, and A. Wigderson (2019). How to play any mental game, or a completeness theorem for protocols with honest majority. In *Providing Sound Foundations for Cryptography: On the Work of Shafi Goldwasser and Silvio Micali*, pp. 307–328.

Graversen, E., A. K. Hirsch, and F. Montesi (2024). Alice or bob?: Process polymorphism in choreographies. *Journal of Functional Programming 34*, e1.

Hewitt, C., P. Bishop, and R. Steiger (1973). A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*, IJCAI'73, San Francisco, CA, USA, pp. 235–245. Morgan Kaufmann Publishers Inc.

Hirsch, A. K. and D. Garg (2022, January). Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang. 6*(POPL).

Honda, K., N. Yoshida, and M. Carbone (2008). Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, New York, NY, USA, pp. 273–284. Association for Computing Machinery.

Jongmans, S.-S. (2025). First-person choreographic programming with continuation-passing communications. In V. Vafeiadis (Ed.), *Programming Languages and Systems*, Volume 2 of *Lecture Notes in Computer Science*, pp. 62–90. Springer.

Jongmans, S.-S. and P. van den Bos (2022). *A Predicate Transformer for Choreographies (Full Version)*. Number 01 in OUNL-CS (Technical Reports). Open Universiteit Nederland.

Kashiwa, S., G. Shen, S. Zare, and L. Kuper (2023). Portable, efficient, and practical library-level choreographic programming. `https://arxiv.org/abs/2311.11472`.

Keeler, D., C. Komlo, E. Lepert, S. Veitch, and X. He (2023, 07). Dprio: Efficient differential privacy with high utility for prio. *Proceedings on Privacy Enhancing Technologies 2023*, 375–390.

Le Brun, M. A., S. Fowler, and O. Dardha (2025). Multiparty session types with a bang! In V. Vafeiadis (Ed.), *Programming Languages and Systems*, Cham, pp. 125–153. Springer Nature Switzerland.

Lugović, L. and S.-S. Jongmans (2024). Klor: Choreographies in clojure. `https://github.com/lovrosdu/klor`.

Montesi, F. (2014). *Choreographic Programming*. Ph. D. thesis, Denmark.

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

Montesi, F. and M. Peressotti (2017). Choreographies meet communication failures. *CoRR abs/1712.05465*.

Naor, M. and B. Pinkas (2001). Efficient oblivious transfer protocols. In *SODA*, Volume 1, pp. 448–457.

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

Noonan, M. (2018). Ghosts of departed proofs (functional pearl). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, New York, NY, USA, pp. 119–131. Association for Computing Machinery. Archive: `https://kataskeue.com/gdp.pdf`.

Noonan, M. (2019). gdp: Reason about invariants and preconditions with ghosts of departed proofs. `https://hackage.haskell.org/package/gdp-0.0.3.0`.

Petri, C. A. (1962). *Kommunikation mit Automaten*. Ph. D. thesis, Bonn. 128 Seiten.

Plotkin, G. and J. Power (2003). Algebraic operations and generic effects. *Applied categorical structures 11*, 69–94.

Plotkin, G. D. and M. Pretnar (2013, December). Handling algebraic effects. *Logical Methods in Computer Science Volume 9, Issue 4*.

Prabhakaran, M. M. (2023, August). Advanced tools from modern cryptography: Lecture 6. `https://www.cse.iitb.ac.in/~mp/teach/advcrypto/f23/slides/06.pdf`. Archive: `https://web.archive.org/web/20250708141313/https://www.cse.iitb.ac.in/~mp/teach/advcrypto/f23/slides/06.pdf`.

Rastogi, A., M. A. Hammer, and M. Hicks (2014). Wysteria: A programming language for generic, mixed-mode multiparty computations. In *2014 IEEE Symposium on Security and Privacy*, pp. 655–670.

Sajanikar, Y. (2017). *Haskell Cookbook: Build functional applications using Monads, Applicatives, and Functors*. Packt Publishing.

Shen, G., S. Kashiwa, and L. Kuper (2023, aug). Haschor: Functional choreographic programming for all (functional pearl). *Proc. ACM Program. Lang. 7*(ICFP).

Shen, G. and L. Kuper (2024). Toward verified library-level choreographic programming with algebraic effects. `https://arxiv.org/abs/2407.06509`.

Stutz, F. and E. D'Osualdo (2025). An automata-theoretic basis for specification and type checking of multiparty protocols. In V. Vafeiadis (Ed.), *Programming Languages and Systems*, Cham, pp. 314–346. Springer Nature Switzerland.

Sweet, I., D. Darais, D. Heath, W. Harris, R. Estes, and M. Hicks (2023, February). Symphony: Expressive secure multiparty computation with coordination. *The Art, Science, and Engineering of Programming 7*(3).

Urban, C., S. Berghofer, and M. Norrish (2007). Barendregt's variable convention in rule inductions. In F. Pfenning (Ed.), *Automated Deduction – CADE-21*, Berlin, Heidelberg, pp. 35–50. Springer Berlin Heidelberg.

W3C (2005). WS Choreography Description Language. `http://www.w3.org/TR/ws-cdl-10/`.

Weisenburger, P., M. Köhler, and G. Salvaneschi (2018, October). Distributed system development with scalaloci. *Proc. ACM Program. Lang. 2*(OOPSLA).

Wiersdorf, A. and B. Greenman (2024). Chorex: Choreographic programming in elixir. `https://github.com/utahplt/chorex`.

Wu, W., L. He, W. Lin, R. Mao, C. Maple, and S. Jarvis (2020). Safa: A semi-asynchronous protocol for fast federated learning with low overhead. *IEEE Transactions on Computers 70*(5), 655–668.

Zakhour, G., P. Weisenburger, and G. Salvaneschi (2023, October). Type-safe dynamic placement with first-class placed values. *Proc. ACM Program. Lang. 7*(OOPSLA2).

Appendix A: Proofs of Theorems

A.1 Proof of The Substitution Theorem

Theorem 1 says that if $\Theta; \Gamma, (x : T_x) \vdash M : T$ and $\Theta; \Gamma \vdash V : T_x$, then $\Theta; \Gamma \vdash M[x := V] : T$. We first prove a few lemmas.

Lemma 1 (Enclave)**.** *If* $\Theta; \Gamma \vdash V : T$ *and* $\Theta' \subseteq \Theta$ *and* $T' = T \triangleright \Theta'$ *is defined then* $V' = V \triangleright \Theta'$ *is defined, and* $\Theta'; \Gamma \vdash V' : T'$.

A.1.1 Proof of Lemma 1

This is vacuous if T' doesn't exist, so assume it does. Do induction on the definition of masking for T:

- MTDATA: $\Theta; \Gamma \vdash V : d@p^+$ and $p^+ \cap \Theta' \neq \varnothing$ so $T' = d@(p^+ \cap \Theta')$. Consider cases for typing of V:
  - TVAR: $V' = V$ by MVVAR and it types by TVAR b.c. T' exists.
  - TUNIT: We've already assumed the preconditions for MVUNIT, and it types.
  - TPAIR: $V = \mathsf{Pair}\, V_1 V_2$, and $\Theta; \Gamma \vdash V_1 : d_1@(p_1^+ \supseteq p^+)$ and $\Theta; \Gamma \vdash V_2 : d_2@(p_2^+ \supseteq p^+)$. By MTDATA, these larger-owernership types will still mask with Θ', so this case come by induction.
  - TINL, TINR: Follows by simple induction.
- MTFUNCTION: $T' = T$ and $p^+ \subseteq \Theta'$, so lambdas and function-keywords all project unchanged, and the respective typings hold.
- MTVECTOR: Simple induction.

Lemma 2 (Quorum)**.** ***A)*** *If* $\Theta; \Gamma, (x : T_x) \vdash M : T$ *and* $T'_x = T_x \triangleright \Theta$*, then* $\Theta; \Gamma, (x : T'_x) \vdash M : T$*.*

B) *If* $\Theta; \Gamma, (x : T_x) \vdash M : T$ *and* $T_x \triangleright \Theta$ *is not defined, then* $\Theta; \Gamma \vdash M : T$*.*

A.1.2 Proof of Lemma 2

By induction on the typing of M. The only case that's not recursive or trivial is TVar, for which we just need to observe that masking on a given party-set is idempotent.

Lemma 3 (Unused)**.** *If* $\Theta; \Gamma \vdash M : T$ *and* $x \notin \Gamma$*, then* $M[x := V] = M$*.*

A.1.3 Proof of Lemma 3

By induction on the typing of M. There are no non-trivial cases.

A.1.4 Theorem 1

Theorem 1 says that if $\Theta; \Gamma, (x : T_x) \vdash M : T$ and $\Theta; \Gamma \vdash V : T_x$, then $\Theta; \Gamma \vdash M[x := V] : T$.

The proof is in 13 cases. TProjN, TProj1, TProj2, TCom, and TUnit are trivial base cases. TInL, TInR, TVec, and TPair are trivial recursive cases.

- TLambda where $T'_x = T_x \triangleright p^+$: $M = (\lambda y : T_y \,.\, N)@p^+$ and $T = (T_y \to T')@p^+$.

  1. $\Theta; \Gamma, (x : T_x) \vdash (\lambda y : T_y \,.\, N)@p^+ : (T_y \to T')@p^+$ by assumption.
  2. $\Theta; \Gamma \vdash V : T_x$ by assumption.
  3. $p^+; \Gamma, (x : T_x), (y : T_y) \vdash N : T'$ per preconditions of TLambda.
  4. $\Theta; \Gamma, (y : T_y) \vdash V : T_x$ by weakening (or strengthening?) #2.
  5. $V' = V \triangleright p^+$ and $p^+; \Gamma, (y : T_y) \vdash V' : T'_x$ by Lemma 1.
  6. $p^+; \Gamma, (x : T'_x), (y : T_y) \vdash N : T'$ by applying Lemma 2 to #3.
  7. $p^+; \Gamma, (y : T_y) \vdash N[x := V'] : T'$ by induction on #6 and #5.
  8. $M[x := V] = (\lambda y : T_y \,.\, N[x := V'])@p^+$ by definition, which typechecks by #7 and TLambda.

  QED.

- TLambda where $T_x \triangleright p^+$ is undefined: $M = (\lambda y : T_y \,.\, N)@p^+$.

1. $p^+; \Gamma, (x : T_x), (y : T_y) \vdash N : T'$ per preconditions of TLambda.
2. $p^+; \Gamma, (y : T_y) \vdash N : T'$ by Lemma 2 B.
3. $N[x := V] = N$ by Lemma 3, so regardless of the existence of $V \triangleright p^+$ the substitution is a noop, and it typechecks by #2 and TLambda.

- TVar: Follows from the relevant definitions, whether $x \equiv y$ or not.
- TApp: This is also a simple recursive case; the masking of T_a doesn't affect anything.
- TCase: Follows the same logic as TLambda, just duplicated for M_l and M_r.

A.2 Proof of The Preservation Theorem

Theorem 2 says that if $\Theta; \varnothing \vdash M : T$ and $M \longrightarrow M'$, then $\Theta; \varnothing \vdash M' : T$. We'll need a few lemmas first.

Lemma 4 (Sub-Mask)**.** *If $\Theta; \Gamma \vdash V : d@p^+$ and $\varnothing \neq q^+ \subseteq p^+$, then **A:** $d@p^+ \triangleright q^+ = d@q^+$ is defined and **B:** $V \triangleright q^+$ is also defined and types as $d@q^+$.*

A.2.1 Proof of Lemma 4

Part A is obvious by MTData. Part B follows by induction on the definition of masking for values.

- MVLambda: Base case; can't happen because it wouldn't allow a data type.
- MVUnit: Base case; passes definition and typing.
- MVInL, MVInR: Recursive cases.
- MVPair: Recursive case.
- MVVector: Can't happen because it wouldn't allow a data type.
- MVProj1, MVProj2, MVProjN, and MVCom: Base cases, can't happen because they wouldn't allow a data type.
- MVVar: Base case, trivial.

Lemma 5 (Maskable)**.** *If $\Theta; \Gamma \vdash V : T$ and $T \triangleright p^+ = T'$, then **A:** $V \triangleright p^+ = V'$ is defined and **B:** $\Theta; \Gamma \vdash V' : T'$.*

A.2.2 Proof of Lemma 5

By induction on the definition of masking for values.

- MVLambda: Base case. From the type-masking assumption, MTFunction, p^+ is a superset of the owners, so $T' = T$, so $V' = V$.
- MVUnit: Base case; passes definition and typing.
- MVInL, MVInR: Recursive cases.
- MVPair: Recursive case.
- MVVector: Recursive case.
- MVProj1, MVProj2, MVProjN, and MVCom: From the typing assumption, p^+ is a superset of the owners, so $T' = T$ and $V' = V$.
- MVVar: Base case, trivial.

Lemma 6 (Exclave)**.** *If* $\Theta; \varnothing \vdash M : T$ *and* $\Theta \subseteq \Theta'$ *then* $\Theta'; \varnothing \vdash M : T$.

A.2.3 Proof of Lemma 6

By induction on the typing of M.

- TLambda: The recursive typing is unaffected, and the other tests are fine with a larger set.
- TVar: Can't apply with an empty type context.
- All other cases are unaffected by the larger party-set.

A.2.4 Theorem 2

To repeat: Theorem 2 says that if $\Theta; \varnothing \vdash M : T$ and $M \longrightarrow M'$, then $\Theta; \varnothing \vdash M' : T$.

We prove this by induction on typing rules for M. The eleven base cases (values) fail the assumption that M can step, so we consider the recursive cases:

- TCase: M is of form $\mathsf{case}_{p^+}\, N\, \mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow M_l; \mathsf{Inr}\, x_r \Rightarrow M_r$. There are three ways it might step:

– CaseL: N is of form $\mathsf{Inl}\, V$, V' exists, and $M' = M_l[x_l := V']$.

1. $p^+; (x_l : d_l @ p^+) \vdash M_l : T$ by the preconditions of TCase.
2. $\Theta; \varnothing \vdash V : d_l @ p^+$ because N must type by TInL.
3. $p^+; \varnothing \vdash V' : d_l @ p^+$ by Lemma 1 and MTData.
4. $p^+; \varnothing \vdash M_l[x_l := V'] : T$ by Theorem 1.
5. $\Theta; \varnothing \vdash M_l[x_l := V'] : T$ by Lemma 6. **QED.**

– CaseR: Same as CaseL.

– Case: $N \longrightarrow N'$, and by induction and TCase, $\Theta; \Gamma \vdash N' : T_N$, so the original typing judgment will still apply.

• TApp: M is of form FA, and F is of a function type and A also types (both in the empty typing context). If the step is by App2or App1, then recursion is easy. There are eight other ways the step could happen:

– AppAbs: F must type by TLambda. $M = ((\lambda x : T_x \,.\, B) @ p^+) A$. We need to show that $A' = A \triangleright p^+$ exists and $\Theta; \varnothing \vdash B[x := A'] : T$.

1. $p^+; (x : T_x) \vdash B : T$ by the preconditions of TLambda.
2. $\Theta; \varnothing \vdash A : T'_a$ such that $T_x = T'_a \triangleright p^+$, by the preconditions of TApp.
3. A' exists and $p^+; \varnothing \vdash A' : T_x$ by Lemma 1 on #2.
4. $p^+; \varnothing \vdash B[x := A'] : T$ by Theorem 1.
5. **QED.** by Lemma 6.

– Proj1: $F = \mathsf{fst}_{p^+}$ and $A = \mathsf{Pair}\, V_1 V_2$ and $M' = V_1 \triangleright p^+$. Necessarily, by TPair $\Theta; \varnothing \vdash V_1 : d_1 @ p_1^+$ where $p^+ \subseteq p_1^+$. By Lemma 4, $\Theta; \varnothing \vdash M' : T$.

– Proj2: same as Proj1.

– ProjN: $F = \mathsf{lookup}^i_{p^+}$ and $A = (\ldots, V_i, \ldots)$ and $M' = V_i \triangleright p^+$. Necessarily, by TVec $\Theta; \varnothing \vdash V_i : T_i$ and $\Theta; \varnothing \vdash A : (\ldots, T_i, \ldots)$. By TApp, $(\ldots, T_i, \ldots) \triangleright p^+ = T_a$, so by MTVector $T_i \triangleright p^+$ exists and (again by TApp and TProjN) it must equal T. **QED.** by Lemma 5.

– Com1: By TCom and TUnit.

– ComPair: Recusion among the Com* cases.

- ComInl: Recusion among the Com* cases.
- ComInr: Recusion among the Com* cases.

A.3 Proof of The Progress Theorem

Theorem 3 says that if $\Theta; \varnothing \vdash M : T$, then either M is of form V (which cannot step) or their exists M' s.t. $M \longrightarrow M'$.

The proof is by induction of typing rules. There are eleven base cases and two recursive cases. Base cases:

- TLambda
- TVar (can't happen, by assumption)
- TUnit
- TCom
- TPair
- TVec
- TProj1
- TProj2
- TProjN
- TInl
- TInr

Recursive cases:

- TCase: M is of form $\mathsf{case}_{p^+}\, N\, \mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow M_l; \mathsf{Inr}\, x_r \Rightarrow M_r$ and $\Theta; \varnothing \vdash N : (d_l + d_r)@p^+$. By induction, either N can step, in which case M can step by Case, or N is a value. The only typing rules that would give an N of form V the required type are TVar (which isn't compatible with the assumed

empty Γ), and TInl and TInr, which respectively force N to have the required forms for M to step by CaseL or CaseR. From the typing rules, MTData, and the first part of Lemma 1, the masking required by the step rules is possible.

- TApp: M is of form FA, and F is of a function type and A also types (both in the same empty Γ). By induction, either F can step (so M can step by App2), or A can step (so M can step by App1), or F and A are both values. Ignoring the impossible TVar cases, there are five ways an F of form V could type as a function; in each case we get to make some assumption about the type of A. Furthermore, by TApp and Lemma 1, we know that A can mask to the owners of F.
  - TProj1: A must be a value of type $(d_1 \times d_2)@q^+$, and must type by TPair, so it must have form $\mathsf{Pair}\, V_1 V_2$, so M must step by Proj1. We know V_1 can mask by MVPair.
  - TProj2: (same as TProj1)
  - TProjN: A must be a value of type $(T_1, \ldots, T_n)$ with $i \leq n$ and must type by TVec, so it must have from $(V_1, \ldots, V_n)$. M must step by ProjN. We known V_i can step by MVVector.
  - TCom: A must be a value of type $d@q^+$, such that $d@q^+ \triangleright s^+ = d@s^+$. For that to be true, MTData requires that $s^+ \subseteq q^+$. A can type that way under TUnit, TPair, TInl, or TInr, which respectively force forms $()@q^+$, $\mathsf{Pair}\, V_1 V_2$, $\mathsf{Inl}\, V$, and $\mathsf{Inr}\, V$, which respectively require that M reduce by Com1, ComPair, ComInl, and ComInr. In the case of $()$, this follows from Lemma 4, since $\{s\} \subseteq s^+ \subseteq q^+$; the other three are recursive among each other.
  - TLambda: M must reduce by AppAbs. By the assumption of TApp and Lemma 5, it can.

A.4 Proof of The Soundness Theorem

Theorem 4 says that if $\Theta; \varnothing \vdash M : T$ and $[\![M]\!] \xrightarrow{\varnothing}{}^{*} \mathcal{N}_n$, then there exists M' such that $M \longrightarrow^* M'$ and $\mathcal{N}_n \xrightarrow{\varnothing}{}^{*} [\![M']\!]$. We'll need a few lemmas first.

Lemma 7 (Values)**.** ***A):*** $[\![V]\!]_p = L$. ***B):*** *If* $[\![M]\!]_p = L \neq \bot$ *then* M *is a value* V.

Proof is by inspection of the definition of projection.

Corollary 2. *If* N *is well-typed and* $[\![N]\!]$ *can step at all, then* ***(A)*** N *can step to some* N' *and* ***(B)*** $[\![N]\!]$ *can multi-step to* $[\![N']\!]$ *with empty annotation.*

***A** follows from Lemma 7 and Theorem 3. **B** is just Theorem 5.*

Lemma 8 (Determinism)**.** *If* $\mathcal{N}_a \mid \mathcal{N}_0 \xrightarrow{\varnothing} \mathcal{N}_a \mid \mathcal{N}_1$ *s.t. for every* $p[B_0] \in \mathcal{N}_0$, $\mathcal{N}_1(p) \neq B_0$, and $\mathcal{N}_b \mid \mathcal{N}_0 \xrightarrow{\varnothing} \mathcal{N}_c \mid \mathcal{N}_2$ *s.t. the domain of* $\mathcal{N}_2$ *equals the domain of* $\mathcal{N}_0$*, then* either

- $\mathcal{N}_2 = \mathcal{N}_0$, or
- $\mathcal{N}_2 = \mathcal{N}_1$ *and* $\mathcal{N}_b = \mathcal{N}_c$.

A.4.1 Proof of Lemma 8

First, observe that for every non-value expression in the process language, there is at most one rule in the process semantics by which it can step. (For values, there are zero.) Furthermore, the only way for the step annotation and resulting expression to *not* be fully determined by the initial expression is if the justification is based on a LRecv step, in which case the send-annotation will be empty and the resulting expression will match the (single) item in the receive-annotation.

$\mathcal{N}_a \mid \mathcal{N}_0 \xrightarrow{\varnothing} \mathcal{N}_a \mid \mathcal{N}_1$ must happen by NPar, so consider the $\mathcal{N}_0$ step that enables it; call that step $\mathfrak{S}$. $\mathfrak{S}$ can't be by NPar; that would imply parties in $\mathcal{N}_0$ who don't step.

- If $\mathfrak{S}$ is by NPro, then $\mathcal{N}_0 = p[B_0]$ is a singleton and $\mathfrak{S}$ is justified by a process step with empty annotation. As noted above, that process step is the only step B_0 can take, so the $\mathcal{N}_b \mid \mathcal{N}_0 \xrightarrow{\varnothing} \mathcal{N}_c \mid \mathcal{N}_2$ step must either be a NPar composing some other party(ies) step with $\mathcal{N}_0$ (satisfying the first choice), or a NPar composing $\mathfrak{S}$ with $\mathcal{N}_b$ (satisfying the second).
- If $\mathfrak{S}$ is by NCom, then there must be both a singleton NPro step justified by a process step (by some party s) with nonempty send-annotation and a nonempty sequence of other party steps (covering the rest of $\mathcal{N}_0$'s domain) that it gets matched with each with a corresponding receive-annotation. The send-annotated NPro step is deterministic in the same way as an empty-annotated NPro step. In order for the parties to cancel out, it can only compose by NCom with (a permutation of) the same sequence of peers. Considered in isolation, the peers are non-deterministic, but their process-steps can only be used in the network semantics by composing with s via NCom, and their resulting expressions are determined by the matched process annotation, which is determined by s's step.

  Thus, for any $p[B_2] \in \mathcal{N}_2$, $B_2 \neq \mathcal{N}_0(p)$ implies that for all $q[B_2'] \in \mathcal{N}_2$, $B_2' = \mathcal{N}_1(p)$. In the case where $\mathcal{N}_2 = \mathcal{N}_1$, the step from $\mathcal{N}_0$ could only have composed with $\mathcal{N}_b$ by NPar, so $\mathcal{N}_b = \mathcal{N}_c$, Q.E.D.

Lemma 9 (Parallelism)**.** ***A):*** *If* $\mathcal{N}_1 \xrightarrow{\varnothing}{}^{*} \mathcal{N}_1'$ *and* $\mathcal{N}_2 \xrightarrow{\varnothing}{}^{*} \mathcal{N}_2'$ *then* $\mathcal{N}_1 \mid \mathcal{N}_2 \xrightarrow{\varnothing}{}^{*} \mathcal{N}_1' \mid \mathcal{N}_2 \xrightarrow{\varnothing}{}^{*} \mathcal{N}_1' \mid \mathcal{N}_2'$.
B): *If* $\mathcal{N}_1 \mid \mathcal{N}_2 \xrightarrow{\varnothing}{}^{*} \mathcal{N}_1' \mid \mathcal{N}_2 \xrightarrow{\varnothing}{}^{*} \mathcal{N}_1' \mid \mathcal{N}_2'$*, then* $\mathcal{N}_1 \xrightarrow{\varnothing}{}^{*} \mathcal{N}_1'$ *and* $\mathcal{N}_2 \xrightarrow{\varnothing}{}^{*} \mathcal{N}_2'$.

A.4.2 Proof of Lemma 9

A is just repeated application of NPar.

For **B**, observer that in the derivation tree of ever step of the sequence, some (possibly different) minimal sub-network will step by NPro or NCom as a precondition to some number of layers of NPar. The domains of these minimal sub-networks will be subsets of the domains of $\mathcal{N}_1$ and $\mathcal{N}_2$ respectively, so they can just combine via NPar to get the needed step in the respective sequences for $\mathcal{N}_1$ and $\mathcal{N}_2$.

A.4.3 Theorem 4

Theorem 4 says that if $\Theta; \varnothing \vdash M : T$ and $[\![M]\!] \xrightarrow{\varnothing}{}^{*} \mathcal{N}_n$, then there exists M' such that $M \longrightarrow^{*} M'$ and $\mathcal{N}_n \xrightarrow{\varnothing}{}^{*} [\![M']\!]$.

Declare the predicate $\mathsf{sound}(\mathcal{N})$ to mean that there exists some $M_{\mathcal{N}}$ such that $M \longrightarrow^{*} M_{\mathcal{N}}$ and $\mathcal{N} \xrightarrow{\varnothing}{}^{*} [\![M_{\mathcal{N}}]\!]$.

Consider the sequence of network steps $[\![M]\!] = \mathcal{N}_0 \xrightarrow{\varnothing} \ldots \xrightarrow{\varnothing} \mathcal{N}_n$. By Corollary 2, $\mathsf{sound}(\mathcal{N}_0)$. Select the largest i s.t. $\mathsf{sound}(\mathcal{N}_i)$. We will derive a contradiction from an assumption that $\mathcal{N}_{i+1}$ is part of the sequence; this will prove that $i = n$, which completes the proof of the Theorem.

Choose a sequence of network steps (of the possibly many such options) $\mathcal{N}_i = \mathcal{N}_i^a \xrightarrow{\varnothing} \ldots \xrightarrow{\varnothing} \mathcal{N}_m^a = [\![M^a]\!]$ where $M \longrightarrow^{*} M^a$.

Assume $\mathcal{N}_{i+1}$ is part of the original sequence. Decompose the step to it as $\mathcal{N}_i = \mathcal{N}_i^0 \mid \mathcal{N}_i^1 \xrightarrow{\varnothing} \mathcal{N}_i^0 \mid \mathcal{N}_{i+1}^1 = \mathcal{N}_{i+1}$ where $\mathcal{N}_i^1$'s domain is as large as possible. We will examine two cases: either the parties in $\mathcal{N}_i^1$ make steps in the sequence to $\mathcal{N}_m^a$, or they do not. Specifically, consider the largest j s.t. $\mathcal{N}_j^a = \mathcal{N}_j^b \mid \mathcal{N}_i^1$.

- Suppose $j < m$.

  By Lemma 8 and our decision that j is as large as possible, $\mathcal{N}_{j+1}^a = \mathcal{N}_j^b \mid \mathcal{N}_{i+1}^1$. Thus we have $\mathcal{N}_i^0 \mid \mathcal{N}_i^1 \xrightarrow{\varnothing}{}^{*} \mathcal{N}_j^b \mid \mathcal{N}_i^1 \xrightarrow{\varnothing} \mathcal{N}_j^b \mid \mathcal{N}_{i+1}^1$. By Lemma 9, we can reorganize that into an alternative sequence where $\mathcal{N}_i^0 \mid \mathcal{N}_i^1 \xrightarrow{\varnothing} \mathcal{N}_i^0 \mid \mathcal{N}_{i+1}^1 \xrightarrow{\varnothing}{}^{*} \mathcal{N}_j^b \mid \mathcal{N}_{i+1}^1$. Since $\mathcal{N}_i^0 \mid \mathcal{N}_{i+1}^1 = \mathcal{N}_{i+1}$ and $\mathcal{N}_{j+1}^a \xrightarrow{\varnothing}{}^{*} [\![M^a]\!]$, this contradicts our choice that i be as large as possible.

- Suppose $j = m$, so $[\![M^a]\!] = \mathcal{N}_m^b \mid \mathcal{N}_i^1$.

  By Lemma 9, $[\![M^a]\!]$ can step (because $\mathcal{N}_i^1$ can step) so by Corollary 2, $M^a \longrightarrow M^{a+1}$. We can repeat our steps from our choice of $\mathcal{N}_i^a \xrightarrow{\varnothing}{}^* \mathcal{N}_m^a = [\![M^a]\!]$, but using M^{a+1} instead of M^a. Since $\boldsymbol{\lambda_C}$ doesn't have recursion, eventually we'll arrive at a M^{a++} that can't step, and then-or-sooner we'll be in the first case above. Q.E.D.

A.5 Proof of The Completeness Theorem

Theorem 5 says that if $\Theta; \varnothing \vdash M : T$ and $M \longrightarrow M'$, then $[\![M]\!] \xrightarrow{\varnothing}{}^* [\![M']\!]$. We'll need a few lemmas first.

Lemma 10 (Cruft)**.** *If* $\Theta; \varnothing \vdash M : T$ *and* $p \notin \Theta$*, then* $[\![M]\!]_p = \bot$.

A.5.1 Proof of Lemma 10

By induction on the typing of M:

- TLambda: $p^+ \subseteq \Theta$, therefore $p \notin p^+$, therefore $[\![M]\!]_p = \bot$.
- TVar: Can't happen because M types with empty Γ.
- TUnit, TCom, TProj1, TProj2, and TProjN: Same as TLambda.
- TPair, TVec, TInl, and TInr: In each of these cases we have some number of recursive typing judgments to which we can apply the inductive hypothesis. This enables the respective cases of the definition of floor (as used in the respective cases of the definition of projection) to map to $\bot$.
- TApp: $M = N_1 N_2$. By induction, $[\![N_1]\!]_p = \bot$ and $[\![N_2]\!]_p = \bot$, so $[\![M]\!]_p = \bot$
- TCase: Similar to TLambda, by induction the guard projects to $\bot$ and therefore the whole thing does too.

Lemma 11 (Existence)**.** *If* $\Theta; \Gamma \vdash V : d@p^+$ *and* $p, q \in p^+$*, then* $[\![V]\!]_p = [\![V]\!]_q \neq \bot$.

A.5.2 Proof of Lemma 11

By induction on possible typings of V:

- TVar: Projection is a no-op on variables.
- TUnit: $[\![V]\!]_p = [\![V]\!]_q = ()$.
- TPair: $p, q \in p_1^+ \cap p_2^+$, so both are in each of them, so we can recurse on V_1 and V_2.
- TInl and TInr: simple induction.

Lemma 12 (Bottom)**.** *If* $\Theta; \varnothing \vdash M : T$ *and* $[\![M]\!]_p = \bot$ *and* $M \longrightarrow M'$ *then* $[\![M']\!]_p = \bot$.

A.5.3 Proof of Lemma 12

By induction on the step $M \longrightarrow M'$.

- AppAbs: $M = (\lambda x : T_x \,.\, N)@p^+V$, and necessarily $[\![(\lambda x : T_x \,.\, N)@p^+]\!]_p = \bot$. Since the lambda doesn't project to a lambda, $p \notin p^+$. $M' = N[x := V \triangleright p^+]$. By TLambda, Theorem 1, and Lemma 10, $[\![N[x := V \triangleright p^+]]\!]_p = \bot$.
- App1: $M = VN$ and necessarily $[\![V]\!]_p = [\![N]\!]_p = \bot$. By induction on $N \longrightarrow N'$, $[\![N']\!]_p = \bot$.
- App2: Same as App1.
- Case: The guard must project to $\bot$, so this follows from induction.
- CaseL (and CaseR by mirror image): $M = \mathsf{case}_{p^+}\, \mathsf{Inl}\, V\, \mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow M_l; \mathsf{Inr}\, x_r \Rightarrow M_r$ and $M' = M_l[x_l := V \triangleright p^+]$. Necessarily, $[\![V]\!]_p = \bot$. By TCase and MTData, $\mathsf{Inl}\, V$ types as data, so by Lemma 11 $p \notin p^+$. By TCase, Theorem 1, and Lemma 10, $[\![M']\!]_p = [\![M_l[x_l := V \triangleright p^+]]\!]_p = \bot$.
- Proj1: $M = \mathsf{fst}_{p^+}(\mathsf{Pair}\, V_1 V_2)$, and $p \notin p^+$. $M' = V_1 \triangleright p^+$. Since $\Theta; \varnothing \vdash V_1 : T'$ (by TPair) and $T' \triangleright p^+ = T''$ is defined (by TApp and the indifference of MTData to the data's structure), by Lemma 1 $p^+; \varnothing \vdash V_1 \triangleright p^+ : T''$. By Lemma 10 this projects to $\bot$.
- Proj2, ProjN, and Com1 are each pretty similar to Proj1.
- Com1, ComPair, ComInl, and ComInr: For M to project to $\bot$, p must be neither a sender nor a recipient. By induction among these cases (with Com1 as the base case), M' will be some structure of $()@r^+$; since $p \notin r^+$ and projection uses floor, this will project to $\bot$.

Lemma 13 (Masked)**.** *If* $p \in p^+$ *and* $V' = V \triangleright p^+$ *then* $[\![V]\!]_p = [\![V']\!]_p$.

A.5.4 Proof of Lemma 13

By (inductive) case analysis of endpoint projection:

- $[\![x]\!]_p = x$. By MVVar the mask does nothing.
- $[\![(\lambda x : T\,.\,M)@q^+]\!]_p$: Since $V \triangleright p^+$ is defined, by MVLambda it does nothing.
- $[\![()@q^+]\!]_p$: By MVUnit $V' = ()@(p^+ \cap q^+)$. p is in that intersection iff $p \in q^+$, so the projections will both be () or $\bot$ correctly.
- $\mathsf{Inl}\, V_l$, $\mathsf{Inr}\, V_r$, $\mathsf{Pair}\, V_1 V_2$, $(V_1, \ldots, V_n)$: simple recursion.
- fst_{q^+}, snd_{q^+}, $\mathsf{lookup}^i_{q^+}$, $\mathsf{com}_{q;q^+}$: Since the masking is defined, it does nothing.

Lemma 14 (Floor Zero)**.** $[\![M]\!]_p = \lfloor [\![M]\!]_p \rfloor$

A.5.5 Proof of Lemma 14

There are thirteen forms. Six of them (application, case, injection-r/l, pair and vector) apply floor directly in the definition of projection. Six of them (variable, unit, the three lookups, and com) can only project to values such that floor is a no-op. For a lambda $(\lambda x : T_x\,.\,N)@p^+$, the proof is by induction on the body N.

Lemma 15 (Distributive Substitution)**.** *If* $\Theta; (x : T_x) \vdash M : T$ *and* $p \in \Theta$*,*
then $[\![M[x := V]]\!]_p = \lfloor [\![M]\!]_p [x := [\![V]\!]_p] \rfloor$*. (Because* $[\![V]\!]_p$ *may be* $\bot$*, this isn't really distribution; an extra flooring operation is necessary.)*

A.5.6 Proof of Lemma 15

It'd be more elegant if substitution really did distribute over projection, but this weaker statement is what we really need anyway. The proof is by inductive case analysis on the form of M:

- $\mathsf{Pair}\, V_1 V_2$: $[\![M[x := V]]\!]_p = [\![\mathsf{Pair}\, V_1[x := V] V_2[x := V]]\!]_p$
  $= \lfloor \mathsf{Pair} [\![V_1[x := V]]\!]_p [\![V_2[x := V]]\!]_p \rfloor$
  and $[\![M]\!]_p [x := [\![V]\!]_p] = \lfloor \mathsf{Pair} [\![V_1]\!]_p [\![V_2]\!]_p \rfloor\, [x := [\![V]\!]_p]$.

– Suppose one of $[\![V_1]\!]_p, [\![V_2]\!]_p$ is not $\bot$. Then

$[\![M]\!]_p[x := [\![V]\!]_p] = (\mathsf{Pair}\, \lfloor [\![V_1]\!]_p \rfloor\, \lfloor [\![V_2]\!]_p \rfloor)[x := [\![V]\!]_p]$

which by Lemma 14 $= (\mathsf{Pair}[\![V_1]\!]_p[\![V_2]\!]_p)[x := [\![V]\!]_p]$

$= \mathsf{Pair}([\![V_1]\!]_p[x := [\![V]\!]_p])([\![V_2]\!]_p[x := [\![V]\!]_p])$.

Thus $\lfloor [\![M]\!]_p[x := [\![V]\!]_p] \rfloor = \lfloor \mathsf{Pair}([\![V_1]\!]_p[x := [\![V]\!]_p])([\![V_2]\!]_p[x := [\![V]\!]_p]) \rfloor$.

By induction, $[\![V_1[x := V]]\!]_p = \lfloor [\![V_1]\!]_p[x := [\![V]\!]_p] \rfloor$

and $[\![V_2[x := V]]\!]_p = \lfloor [\![V_2]\!]_p[x := [\![V]\!]_p] \rfloor$; with that in mind,

∗ Suppose one of $[\![V_1[x := V]]\!]_p, [\![V_1[x := V]]\!]_p$ is not $\bot$.

$\lfloor [\![M]\!]_p[x := [\![V]\!]_p] \rfloor = \mathsf{Pair}\, \lfloor [\![V_1]\!]_p[x := [\![V]\!]_p] \rfloor\, \lfloor [\![V_2]\!]_p[x := [\![V]\!]_p] \rfloor$,

and $[\![M[x := V]]\!]_p = \mathsf{Pair}\, \lfloor [\![V_1[x := V]\!]_p \rfloor\, \lfloor [\![V_2[x := V]]\!]_p \rfloor$

$= \mathsf{Pair}[\![V_1[x := V]\!]_p[\![V_2[x := V]]\!]_p$ Q.E.D.

∗ Otherwise, $\lfloor [\![M]\!]_p[x := [\![V]\!]_p] \rfloor = \bot = [\![M[x := V]]\!]_p$.

– Otherwise, $[\![M]\!]_p[x := [\![V]\!]_p] = \lfloor \mathsf{Pair}\, \bot\bot \rfloor\, [x := [\![V]\!]_p] = \bot$.

Note that, by induction *etc*, $[\![V_1]\!]_p = \bot = [\![V_1]\!]_p[x := [\![V]\!]_p] = \lfloor [\![V_1]\!]_p[x := [\![V]\!]_p] \rfloor = [\![V_1[x := V]]\!]_p$, and the same for V_2, so $[\![M[x := V]]\!]_p = \bot$, Q.E.D.

- $\mathsf{Inl}\, V_l$, $\mathsf{Inr}\, V_r$, $(V_1, \ldots, V_n)$: Follow the same inductive pattern as Pair.

- N_1N_2: $[\![M[x := V]]\!]_p = [\![N_1[x := V]N_2[x := V]]\!]_p = \lfloor [\![N_1[x := V]]\!]_p [\![N_2[x := V]]\!]_p \rfloor$

$$= \begin{cases} \lfloor [\![N_1[x := V]]\!]_p \rfloor = \bot, \lfloor [\![N_2[x := V]]\!]_p \rfloor = L : & \bot \\ \text{else} : & \lfloor [\![N_1[x := V]]\!]_p \rfloor\, \lfloor [\![N_2[x := V]]\!]_p \rfloor \end{cases}$$

$$= \begin{cases} [\![N_1[x := V]]\!]_p = \bot, [\![N_2[x := V]]\!]_p = L : & \bot \\ \text{else} : & [\![N_1[x := V]]\!]_p [\![N_2[x := V]]\!]_p \end{cases}$$

and $\lfloor [\![M]\!]_p[x := [\![V]\!]_p] \rfloor = \lfloor \lfloor [\![N_1]\!]_p [\![N_2]\!]_p \rfloor\, [x := [\![V]\!]_p] \rfloor$

$$= \begin{cases} \lfloor [\![N_1]\!]_p \rfloor = \bot, \lfloor [\![N_2]\!]_p \rfloor = L : & \lfloor \bot[x := [\![V]\!]_p] \rfloor = \bot \\ \text{else} : & \lfloor (\lfloor [\![N_1]\!]_p \rfloor\, \lfloor [\![N_2]\!]_p \rfloor)[x := [\![V]\!]_p] \rfloor \\ & \quad = \lfloor ([\![N_1]\!]_p[x := [\![V]\!]_p])([\![N_2]\!]_p[x := [\![V]\!]_p]) \rfloor \end{cases}$$

$$= \begin{cases} \lfloor [\![N_1]\!]_p[x := [\![V]\!]_p] \rfloor = \bot, \lfloor [\![N_2]\!]_p[x := [\![V]\!]_p] \rfloor = L : \bot \\ \text{else} : \lfloor [\![N_1]\!]_p[x := [\![V]\!]_p] \rfloor\, \lfloor [\![N_2]\!]_p[x := [\![V]\!]_p] \rfloor \end{cases}$$

(Note that we collapsed the $\lfloor [\![N_1]\!]_p \rfloor = \bot, \ldots$ case. We can do that because if $[\![N_1]\!]_p = \bot$ then so does $\lfloor [\![N_1]\!]_p [x := [\![V]\!]_p] \rfloor$ and if $[\![N_2]\!]_p = L$ then $\lfloor [\![N_2]\!]_p [x := [\![V]\!]_p] \rfloor$ is also a value.)
By induction, $[\![N_1[x := V]]\!]_p = \lfloor [\![N_1]\!]_p [x := [\![V]\!]_p] \rfloor$
and $[\![N_2[x := V]]\!]_p = \lfloor [\![N_2]\!]_p [x := [\![V]\!]_p] \rfloor$.

- y: trivial because EPP and floor are both no-ops.
- $(\lambda y : T_y \,.\, N)@p^+$:
  - If $p \notin p^+$, both sides of the equality are $\bot$.
  - If $V' = V \triangleright p^+$ is defined, then
    $[\![(\lambda y : T_y \,.\, N)@p^+[x := V]]\!]_p = [\![(\lambda y : T_y \,.\, N[x := V'])@p^+]\!]_p = \lambda y \,.[\![N[x := V']]\!]_p$
    and $\lfloor [\![(\lambda y : T_y \,.\, N)@p^+]\!]_p [x := [\![V]\!]_p] \rfloor$
    $= \lfloor (\lambda y \,.[\![N]\!]_p)[x := [\![V]\!]_p] \rfloor$
    $= \lfloor \lambda y \,.([\![N]\!]_p [x := [\![V]\!]_p]) \rfloor$
    $= \lfloor \lambda y \,.([\![N]\!]_p [x := [\![V']\!]_p]) \rfloor$ (by Lemma 13)
    $= \lambda y \,. \lfloor ([\![N]\!]_p [x := [\![V']\!]_p]) \rfloor$
    Then we do induction on N and V'.
  - Otherwise, substitution in the central program is a no-op.
    * $[\![(\lambda y : T_y \,.\, N)@p^+[x := V]]\!]_p = [\![(\lambda y : T_y \,.\, N)@p^+]\!]_p = \lambda y \,.[\![N]\!]_p$
      and
      $\lfloor [\![(\lambda y : T_y \,.\, N)@p^+]\!]_p [x := [\![V]\!]_p] \rfloor = \lfloor (\lambda y \,.[\![N]\!]_p)[x := [\![V]\!]_p] \rfloor$
      $= \lfloor \lambda y \,.([\![N]\!]_p [x := [\![V]\!]_p]) \rfloor$
      $= \lambda y \,. \lfloor [\![N]\!]_p [x := [\![V]\!]_p] \rfloor$.
    * Since we already known $(\lambda y : T_y \,.\, N)@p^+[x := V] = (\lambda y : T_y \,.\, N)@p^+$, we can apply Theorem 1 to M and unpack the typing of $M[x := V] = M$ to get $p^+; (y : T_y) \vdash N : T'$.
    * By Lemma 3, we get $N[x := V] = N$.
    * By induction on N and V, we get $\lfloor [\![N]\!]_p [x := [\![V]\!]_p] \rfloor = [\![N[x := V]]\!]_p = [\![N]\!]_p$, QED.
- $\mathsf{case}_{p^+} N \,\mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow N_l; \mathsf{Inr}\, x_r \Rightarrow N_r$:

- If $[\![N]\!]_p = \bot$ then $\lfloor [\![N]\!]_p[x := [\![V]\!]_p] \rfloor = \bot = [\![N[x := V]]\!]_p$ (by induction), so both halfs of the equality are $\bot$.
- Else if $p \notin p^+$, then we get

  $[\![\mathsf{case}_{p^+}\, N[x := V]\, \mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow N'_l; \mathsf{Inr}\, x_r \Rightarrow N'_r]\!]_p = \mathsf{case}_{p^+}[\![N[x := V]]\!]_p\, \mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow \bot; \mathsf{Inr}\, x_r \Rightarrow \bot$

  and

  $\lfloor [\![\mathsf{case}_{p^+}\, N\, \mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow N_l; \mathsf{Inr}\, x_r \Rightarrow N_r]\!]_p[x := [\![V]\!]_p] \rfloor$
  $= \lfloor (\mathsf{case}_{p^+}[\![N]\!]_p\, \mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow \bot; \mathsf{Inr}\, x_r \Rightarrow \bot)[x := [\![V]\!]_p] \rfloor$
  $= \lfloor \mathsf{case}_{p^+}[\![N]\!]_p[x := [\![V]\!]_p]\, \mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow \bot; \mathsf{Inr}\, x_r \Rightarrow \bot \rfloor$.

  Since we've assumed $\lfloor [\![N]\!]_p[x := [\![V]\!]_p] \rfloor \neq \bot$, these are equal by induction.
- Else if $V' = V \triangleright p^+$ is defined then we can do induction similar similar to how we did for the respective lambda case, except the induction is three-way.
- Otherwise, it's similar to the respective lambda case, just more verbose.

- $()@p^+$, fst_{p^+}, snd_{p^+}, $\mathsf{lookup}^i_{p^+}$, and $\mathsf{com}_{s;r^+}$: trivial because substitution and floor are no-ops.

Lemma 16 (Weak Completeness)**.** *If $\Theta; \varnothing \vdash M : T$ and $M \longrightarrow M'$ then $[\![M]\!]_p \xrightarrow{\oplus\mu;\ominus\eta}{}^{?} [\![M']\!]_p$. (i.e. it takes zero or one steps to get there.)*

A.5.7 Proof of Lemma 16

If $[\![M]\!]_p = \bot$ then this is follows trivially from Lemma 12, so assume it doesn't. We proceed with induction on the form of $M \longrightarrow M'$:

- AppAbs: $M = (\lambda x : T_x\,.\,N)@p^+V$, and $M' = N[x := V \triangleright p^+]$. By assumption, the lambda doesn't project to $\bot$, so $p \in p^+$ and $[\![M]\!]_p \xrightarrow{\oplus\varnothing;\ominus\varnothing} \lfloor [\![N]\!]_p[x := [\![V]\!]_p] \rfloor$ by LAbsApp.
  By Lemma 13 and Lemma 15 $\lfloor [\![N]\!]_p[x := [\![V]\!]_p] \rfloor = \lfloor [\![N]\!]_p[x := [\![V \triangleright p^+]\!]_p] \rfloor = [\![N[x := V \triangleright p^+]]\!]_p = [\![M']\!]_p$.
- App1: $M = VN \longrightarrow VN' = M'$. By induction, $[\![N]\!]_p \xrightarrow{\oplus\mu;\ominus\eta}{}^{?} [\![N']\!]_p$.

- Assume $[\![V]\!]_p = \bot$. By our earlier assumption, $[\![N]\!]_p \neq \bot$. Since $[\![N]\!]_p$ can step; that step justifies a LApp1 step with the same annotations. If $[\![N']\!]_p$ is a value then that'll be handled by the floor built into LApp1.
- Otherwise, the induction is even simpler, we just don't have to worry about possibly collapsing the whole thing to $\bot$.

- App2: $M = N_1 N_2 \longrightarrow N_1' N_2 = M'$. By induction, $[\![N_1]\!]_p \xrightarrow{\oplus\mu;\ominus\eta}{}^{?} [\![N_1']\!]_p$.
  - Assume $[\![N_2]\!]_p = L$. By our earlier assumption, $[\![N_1]\!]_p \neq \bot$. Since $[\![N_1]\!]_p$ steps, that step justifies a LApp2 step with the same annotations. If $[\![N_1']\!]_p$ is a value then that'll be handled by the floor built into LApp2.
  - Otherwise, the induction is even simpler.

- Case: By our assumptions, the guard can't project to $\bot$; we just do induction on the guard to satisfy LCase.

- CaseL (CaseR mirrors): $M = \mathsf{case}_{p^+} \mathsf{Inl}\, V \,\mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow M_l; \mathsf{Inr}\, x_r \Rightarrow M_r$, and $[\![M]\!]_p = \mathsf{case}\, \mathsf{Inl}[\![V]\!]_p \,\mathsf{of}\, \mathsf{Inl}\, x_l \Rightarrow B_l; \mathsf{Inr}\, x_r \Rightarrow B_r$. $[\![M]\!]_p \xrightarrow{\oplus\varnothing;\ominus\varnothing} \lfloor B_l[x_l := [\![V]\!]_p] \rfloor$ by LCaseL. $M' = M_l[x_l := V \triangleright p^+]$. If $p \in p^+$ then $B_l = [\![M_l]\!]_p$ and by Lemma 13 and Lemma 15 $\lfloor B_l[x_l := [\![V]\!]_p] \rfloor = \lfloor [\![M_l]\!]_p[x_l := [\![V]\!]_p] \rfloor = \lfloor [\![M_l]\!]_p[x_l := [\![V \triangleright p^+]\!]_p] \rfloor$
  $= [\![M_l[x_l := V \triangleright p^+]]\!]_p = [\![M']\!]_p$.
  Otherwise, $B_l[x_l := [\![V]\!]_p] = \bot$ and by TCase, Theorem 1, and Lemma 10, $[\![M']\!]_p = \bot$.

- Proj1: $M = \mathsf{fst}_{p^+}(\mathsf{Pair}\, V_1 V_2)$ and $M' = V_1 \triangleright p^+$. Since we assumed $[\![M]\!]_p \neq \bot$, $p \in p^+$.
  $[\![M]\!]_p = \mathsf{fst} \lfloor \mathsf{Pair}[\![V_1]\!]_p [\![V_2]\!]_p \rfloor = \mathsf{fst}(\mathsf{Pair}[\![V_1]\!]_p [\![V_2]\!]_p)$ by Lemma 11 and TPair. This steps by LProj1 to $[\![V_1]\!]_p$, which equals $[\![M']\!]_p$ by Lemma 13.

- Proj2, ProjN: Same as Proj1.

- Com1: $M = \mathsf{com}_{s;r^+}()@p^+$ and $M' = ()@r^+$.
  - $s = p$ and $p \in r^+$: By MVUnit, $p \in p^+$, so $[\![M]\!]_p = \mathsf{send}^*_{r^+\setminus\{p\}}()$, which steps by LSendSelf (using LSend1) to $()$. $[\![M']\!]_p = ()$.

- $s = p$ and $p \notin r^+$: By MVUnit, $p \in p^+$, so $[\![M]\!]_p = \mathsf{send}_{r^+}()$, which steps by LSend1 to $\bot$. $[\![M']\!]_p = \bot$.
- $s \neq p$ and $p \in r^+$: $[\![M]\!]_p = \mathsf{recv}_s[\![()@p^+]\!]_p$, which can step (arbitrarily, but with respective annotation) by LRecv to $[\![M']\!]_p$.
- Otherwise, we violate our earlier assumption.

- ComPair, ComInl, and ComInr: Each uses the same structure of proof as Com1, using induction between the cases to support the respective process-semantics step.

A.5.8 Theorem 5

Theorem 5 says that if $\Theta; \varnothing \vdash M : T$ and $M \longrightarrow M'$, then $[\![M]\!] \xrightarrow{\varnothing}{}^* [\![M']\!]$.

The proof is by case analysis on the semantic step $M \longrightarrow M'$:

- AppAbs, CaseL, CaseR, Proj1, Proj2, and ProjN: Necessarily, the set of parties p^+ for whom $[\![M]\!]_{p \in p^+} \neq \bot$ is not empty. For every $p \in p^+$, by Lemma 16 $[\![M]\!]_p \xrightarrow{\oplus\varnothing;\ominus\varnothing}{}^? [\![M']\!]_p$ (checking the cases to see that the annotations are really empty!). By NPro, each of those is also a network step, which by Lemma 9 can be composed in any order to get $[\![M]\!] \xrightarrow{\varnothing}{}^* \mathcal{N}$. For every $p \in p^+$, $\mathcal{N}(p) = [\![M']\!]_p$, and (by Lemma 12) for every $q \notin p^+$, $\mathcal{N}(q) = \bot = [\![M']\!]_q$, Q.E.D.

- Com1, ComPair, ComInl, and ComInr: $M = \mathsf{com}_{s;r^+} V$. By the recursive structure of Com1, ComPair, ComInl, and ComInr, M' is some structure of $\{\mathsf{Pair}, \mathsf{Inl}, \mathsf{Inr}, ()@r^+\}$, and $[\![M']\!]_{r \in r^+} = [\![V]\!]_s$. For every $q \notin r^+ \cup \{s\}$, $[\![M]\!]_q = \bot = [\![M']\!]_q$ by Lemma 12. Consider two cases:

  - $s \notin r^+$:

    By Lemma 16 $[\![M]\!]_s = \mathsf{send}_{r^+}[\![V]\!]_s \xrightarrow{\oplus\{(r,[\![V]\!]_s)\mid r\in r^+\};\ominus\varnothing} \bot$.

    By the previously mentioned structure of M', $[\![M']\!]_s = \bot$.

    For every $r \in r^+$, by Lemma 16 $[\![M]\!]_r = \mathsf{recv}_s[\![V]\!]_r \xrightarrow{\oplus\varnothing;\ominus\{(s,[\![V]\!]_s)\}} [\![V]\!]_s = [\![M']\!]_r$.

    By NPro, $s[[\![M]\!]_s] \xrightarrow{s:\{(r,[\![V]\!]_s)\mid r\in r^+\}} s[\bot = [\![M']\!]_s]$.

    This composes in parallel with each of the $r_{\in r^+}[[\![M]\!]_r]$ by NCom in any order until the unmactched send is empty. Everyone in and not-in $r^+ \cup \{s\}$ has stepped, if needed, to the respective projection of M'.

– $s \in r^+$: Let $r_0^+ = r^+ \setminus \{s\}$.

By Lemma 16 $[\![M]\!]_s = \mathsf{send}^*_{r_0^+}[\![V]\!]_s \xrightarrow{\oplus\{(r,[\![V]\!]_s)\mid r\in r_0^+\};\ominus\varnothing} [\![V]\!]_s = [\![M']\!]_{s\in r^+}$.

For every $r \in r_0^+$, by Lemma 16 $[\![M]\!]_r = \mathsf{recv}_s[\![V]\!]_r \xrightarrow{\oplus\varnothing;\ominus\{(s,[\![V]\!]_s)\}} [\![V]\!]_s = [\![M']\!]_r$.

We proceed as in the previous case.

- App1 (App2 and Case are similar): $M = VN$. By induction, $[\![N]\!] \xrightarrow{\varnothing}{}^* [\![N']\!]$. Every N step in that process in which a single party advances by NPro can justify a corresponding M step by LApp1. NCom steps are basically the same: each of the participating parties will justify a LApp1 M step with a N step; since this doesn't change the send & receive annotations, the cancellation will still work.

Appendix B: Usability exercise

Appendix B.1, including headings, is the exact text of the instructions provided to volunteers so they could help asses the usability of MultiChor. Figure B.1 is a sequence diagram that was included with the instructions. The volunteers, fellow graduate students with relevant experience and prior introduction to the concepts used in MultiChor, were not able to complete this exercise in the scheduled two-hour sessions. These sessions were open-book/open-universe, and included unstructured interactive guidance from the author.

B.1 MultiChor Demo Exercise

B.1.1 Instructions

Pull the MultiChor repository:

```
git clone -b auction-demo git@github.com:ShapeOfMatter/MultiChor.git
```

Enter its dir and check that you're at the HEAD of the `auction-demo` branch.

Run the (reduced) unit tests to confirm you're set up.

```
cabal test -f test
```

This will also build the whole project, so it may take a little while the first time. One more likely source of problems is your GHC version; version 9.10.1 is preferred.

Open `examples/Auction.hs`. Observe that there's an example choreography `auction` on line 33. You'll be editing this to correctly implement the below protocol.

The goal here is to find specific short-comings of MultiChor as a library that people might actually use. The goal is *not* to test your own skill, or acquire a perfect implementation of the below protocol. Set yourself

a timer for two hours, and quit when it goes off. Ask Mako questions, including about how to do particular things, at any point.

B.1.2 Exercise

A group of companies are setting up an automated system that will run at midnight every night to set the price of doodads for the following day. Five of the companies are buyers, and there is one seller. There is also a "proctor" participant, who provides some oversight. This will be a semi-blind Vickrey auction: Each of the buyers will send their bids to the seller, who will inform everyone of the bid amount and identities of the top two bids. Because the parties are all well-informed and doodads don't change much in value from one day to the next, ties are likely. In the event of a tie, preference will be given to the various parties randomly. The parties trust each other; multi-round commitments are considered unnecessary.

Here is the specific protocol:

1. All buyers send their bids to the seller and the proctor.
2. **IF AND ONLY IF** there is a (possibly many-way) tie for highest bid:
    - The proctor randomly chooses one of the highest-bidding buyers and sends that choice to the seller.
3. The seller sends everyone the two tuples (winner, bid) and (second-place, bid). In the case of a many-way tie, it doesn't matter who is chosen as second-place.
4. All parties print the name of the winner and the amount of the second-place bid.

B.1.3 *Nota bene*

- The Hackage documentation may be easier to navigate than the source code in `src/`, but it may be a little out of date. In particular, we used to use the word "enclave" instead of "conclave".
- If you need a `KnownSymbol` or `KnownSymbols` constraint, you can usually just add it wherever you need it.

- If you need to work with a `Quire`, notice that its instances for `Functor` *etc* don't give you access to the party names; you can use `stackLeaves` to make a new `Quire` and `toLocTm` to get the term-level name of a party.
- The monad `CLI IO` is a shim around `IO`. It satisfies `MonadIO` (so you can always use `liftIO`), and it affords `getInput` and `putOutput`, which are like normal input and output except they can be mocked during testing.
- Most of the test cases have been de-activated so you can run the tests quickly and often. Should you want to turn them all back on, change `examples/Tests.hs` line 41.
- Effectively calling 'main' via 'cabal run...' would require coordinating seven open shells; I don't suggest you bother.

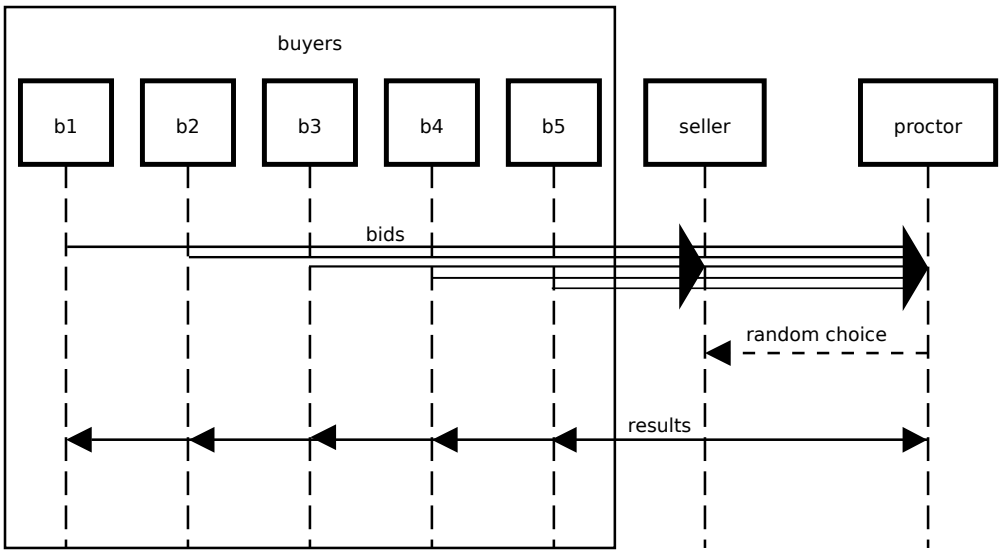


Figure B.1: sequence diagram