UNIVERSITY OF CALIFORNIA
SANTA CRUZ

PORTABLE, EFFICIENT, AND PRACTICAL LIBRARY-LEVEL CHOREOGRAPHIC PROGRAMMING

A dissertation submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

Shun Kashiwa

June 2024

The Dissertation of Shun Kashiwa
is approved:

Professor Lindsey Kuper, Chair

Professor Owen Arden

Professor Daniel Fremont

Peter Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

Abstract

Portable, Efficient, and Practical Library-Level Choreographic Programming

by

Shun Kashiwa

Choreographic programming (CP) is an emerging paradigm for programming distributed applications that run on multiple nodes. In CP, instead of implementing individual programs for each node, the programmer writes one, unified program, called a *choreography*, that is then transformed to individual programs for each node via a compilation step called endpoint projection (EPP). While CP languages have existed for over a decade, *library-level* CP — in which choreographies are expressed as programs in an existing host language, and choreographic language constructs and endpoint projection are provided entirely by a host-language library — is in its infancy. Library-level CP has the potential to improve the accessibility and practicality of CP by meeting programmers where they are, in their programming language of choice, with access to that language's ecosystem, however, the existing implementation approaches have portability, efficiency, and practicality drawbacks that hinder its adoption.

This thesis aims to advance the state of the art of library-level CP by proposing new implementation techniques: *endpoint projection as dependency injection* (EPP-as-DI), and *choreographic enclaves*. EPP-as-DI is a language-agnostic technique for implementing EPP at the library level. Unlike existing library-level approaches, EPP-as-DI asks little from the host language — support for higher-order functions is all that is

required — making it usable in a wide variety of host languages. Choreographic enclaves are a language feature that lets the programmer define *sub-choreographies* within a larger choreography. Within an enclave, "knowledge of choice" is propagated only among the enclave participants, enabling the seamless use of the host language's conditional constructs while addressing the efficiency limitations of existing library-level implementations of choreographic conditionals. This thesis presents ChoRus and Choreography.ts, two library-level CP implementations for Rust and TypeScript, respectively, that use EPP-as-DI and choreographic enclaves. We discuss how EPP-as-DI and choreographic enclaves are implemented in these two languages, and evaluate the usability and performance through two case studies and performance benchmarks.

Acknowledgments

I would like to express my sincere gratitude to my advisor, Professor Lindsey Kuper, for her guidance, support, and encouragement throughout my master's study and research. This work started as my personal hack project to kill time during the spring break, but her enthusiasm and encouragement turned it into a research project. I am grateful for her patience, understanding, and mentorship.

I would like to thank Professor Owen Arden and Professor Daniel Fremont for serving on my thesis committee and providing valuable feedback on my research.

I extend my appreciation to the CASL lab at UC Santa Cruz for providing a supportive and collaborative environment for my research. Special thanks to Gan Shen, who kindly invited me to join his HasChor project and cultivated my interest in choreographic programming.

Last but not least, I would like to thank my friends and family for their unwavering support and encouragement throughout my academic journey.

Chapter 1

Introduction

In a distributed system, a collection of independent nodes communicate with each other by sending and receiving messages. Programmers must ensure that nodes' local behaviors — sending and receiving messages, and taking internal actions — together amount to the desired global behavior of the entire system.

As a very simple example, consider a distributed protocol involving nodes Alice and Bob, in which Alice sends a greeting to Bob and Bob responds. In traditional distributed programming (assuming the existence of `send` and `receive` functions that implement message transport), Alice might run a node-local program `send("Hello!", Bob); receive(Bob)`. Meanwhile, Bob would run his own node-local program `receive(Alice); send("Hi!", Alice)`. Alice and Bob depend on each other to faithfully follow the protocol: if either of them forgets to call `send`, for instance, then their counterpart will wait forever to receive a message (or time out and report an error). This approach is

prone to bugs, including deadlocks.

The emerging paradigm of *choreographic programming* [4, 19] offers a way to rule out this class of bugs. Instead of programming individual nodes, the choreographic programmer writes a single program, called a *choreography*, that expresses the behavior of the entire system from an objective, third-party point of view. For example, the above protocol might be written as the choreography `Alice("Hello") ~> Bob; Bob("Hi") ~> Alice`. The `~>` operator denotes communication between a sender and a receiver [22, 4]. Choreographies are transformed into collections of node-local programs via a compilation step called *endpoint projection* (EPP) [20, 2, 3].

In the last ten years, several choreographic programming (CP) languages have been proposed [4, 19, 9, 8, 11, 13]. However, *library-level* CP — in which choreographies are expressed as programs in an existing host language, and choreographic operators and EPP are provided entirely by a host-language library — is just beginning to emerge.

Library-level CP has the potential to improve the accessibility and practicality of CP by meeting programmers where they are — in their programming language of choice, with access to that language's ecosystem. A library-level implementation of choreographic programming in a given host language enjoys the usual advantages of embedded DSLs [16]: it is installable just like any host-language library, compilable just like any host-language program, and works with any host-language-specific tools for development, debugging, and deployment. Library-level CP also aids the integration of choreographic components into larger, non-choreographic systems, without the need to change languages. Library-level CP fits especially nicely into workflows that port a

non-distributed program — say, a turn-based game in which players sit next to each other at the same machine — to a distributed implementation in which the players interact over a network. With CP, we start and end the process with *one* program, and — with library-level CP — one language.

Given these advantages, how can we realize library-level CP? So far, the only library-level CP implementation is the HasChor framework [22], which implements support for CP as a domain-specific language embedded in Haskell. The HasChor framework represents the current state of the art of library-level CP, but its implementation is quite Haskell-specific and not directly portable. Given the appeal of library-level CP, it makes sense to ask: what is the minimal set of host-language features needed for a viable implementation of library-level CP? In other words, can we "desugar" HasChor?

This thesis proposes new implementation techniques for library-level CP. We make the following specific contributions:

- We propose *endpoint projection as dependency injection* (EPP-as-DI), a novel and language-agnostic implementation technique for library-level CP (Chapter 3). Unlike the HasChor implementation approach, EPP-as-DI asks little from the host language: support for *higher-order functions* is all that is required. As such, the EPP-as-DI approach is straightforward to use in a wide variety of host languages.

- We propose a novel design and implementation technique for implementing efficient conditionals in library-level CP: *choreographic enclaves* (Chapter 4). Using enclaves, a programmer can sidestep the bandwidth inefficiency of a naive imple-

mentation of choreographic conditionals, while still making seamless use of the host language's conditional constructs.

- We present ChoRus and Choreography.ts, two choreographic programming libraries for Rust and TypeScript, respectively, implemented using our proposed techniques (Chapter 5, Chapter 6). We discuss how we implemented EPP-as-DI and choreographic enclaves in the two languages, taking advantage of their unique features, and how we integrated the libraries into the languages' ecosystems.

- Using ChoRus as a model implementation of our proposed techniques, we empirically evaluate the usability and performance of ChoRus compared to traditional distributed programming in Rust (Chapter 7).

Chapter 2

Background

In this chapter, we provide a brief introduction to choreographic programming and its library-level implementation. In Section 2.1, we identify the key elements of choreographic programming using an example implemented in the standalone CP language Choral [12]. Then, in Section 2.2, we give an overview of library-level CP using the HasChor framework [22], and we discuss the strengths and limitations of library-level CP as it stands today.

2.1 The Elements of Choreographic Programming

To illustrate the key concepts of CP, let us consider a well-known example from the literature: the "bookseller" protocol [2, 3, 15, 23]. This protocol describes the interactions between a bookseller and a potential book buyer. First, the buyer sends the name of a book they wish to purchase to the seller. In response, the seller looks

up the catalog and sends back the price of the book to the buyer, who then checks whether the price is within their budget. If the buyer has the means to purchase the book, they notify the seller and obtain an estimated delivery date. Alternatively, if the book's cost exceeds their budget, they communicate to the seller their decision not to proceed with the purchase. Figure 2.1 shows how this protocol might be implemented in a traditional (non-choreographic) fashion as two individual programs, running on the buyer and seller's distinct nodes. We use Python in Figure 2.1 as a representative mainstream programming language. We assume that the `send` and `receive` functions are provided by some library that implements network communication between nodes.

```
def buyer():
    title = input()
    send(title, "seller")
    price = receive("seller")
    decision = price <= budget
    if decision:
        send(True, "seller")
        delivery = receive("seller")
    else:
        send(False, "seller")
```

```
def seller():
    title = receive("buyer")
    price = catalog.get_price(title)
    send(price, "buyer")
    decision = receive("buyer")
    if decision:
        delivery = catalog.get_delivery(title)
        send(delivery_date, "buyer")
```

Figure 2.1: The bookseller protocol implemented as individual node-local programs

Even for simple protocols like the bookseller protocol, it is easy to introduce bugs. For example, the programmer might forget to send the decision to the seller, in which case the seller will wait indefinitely for the buyer's response, causing a deadlock.

```
1  String@Buyer title_buyer = UI@Buyer.input();
2  String@Seller title_seller = c.<String>com(title_buyer);
3  Integer@Buyer price = c.<Integer>com(catalog.quote(title_seller));
4  boolean@Buyer decision = price <= budget;
5  if(decision) {
6      c.<EnumBoolean>select(EnumBoolean@Buyer.True);
7      String@Seller delivery = catalog.get_delivery(title_seller);
8      String@Buyer delivery2 = c.<String>com(delivery);
9  } else {
10     c.<EnumBoolean>select(EnumBoolean@Buyer.False);
11 }
```

Figure 2.2: The bookseller protocol implemented in the Choral choreographic language

The programmer might also use different encodings for the delivery date in the buyer and seller programs, in which case the buyer will not be able to parse the delivery date sent by the seller, causing a type error.

Choreographic programming addresses these problems by letting the programmer implement a protocol as a single, unified program, called a choreography. Figure 2.2 shows an implementation of the bookseller protocol as a choreography in Choral [11], a standalone CP language. Taking Figure 2.2's implementation of the bookseller protocol as an example, let us consider four key elements of CP:

- **Located data and computation.** The bookseller protocol involves two locations, `Buyer` and `Seller`, and data and computation reside at one of these locations. On line 1 of Figure 2.2, the call to the function `input` happens at the buyer, as indicated by the `@Buyer` annotation on the function call. Likewise, the value returned by `input` is located at the buyer, which we see in its type, `String@Buyer`. Choral's type system ensures that data at one location cannot be accessed at a

different location without explicit communication.[1]

- **A unified language construct for communication.** Choreographies replace explicit calls to `send` and `receive` with a single language construct representing communication between a sender and a receiver. In the bookseller protocol, the buyer sends the title of the book to the seller (and the seller receives it) on line 2 of Figure 2.2, using the `com` method. Here, `c` is a channel object that defines how two locations communicate and `com` takes a string located at the buyer and returns a string located at the seller. Additional calls to `com` on lines 3 and 8 express communications from the seller to the buyer.

- **Propagation of "knowledge of choice".** On line 4 of Figure 2.2, the buyer checks whether the price of the book is within their budget, and depending on the decision, the choreography takes different branches. Conditionals in choreographic programming are challenging because of the problem known as "knowledge of choice" [5]. When the branches of a conditional expression encode different communication patterns, all affected locations must be notified of the outcome of evaluating the conditional. In Choral, the `select` method is used to express *selections*, which indicates that the choreography has taken a particular branch and propagated the information to relevant locations. On line 6, the buyer uses `select` to send `True` to the seller if the price is within the budget; otherwise, the buyer sends `False` to the seller on line 10. The seller will receive `True` or `False`

[1] While choreographic languages often represent locations at the type level, the notion of located data and computation is always present in choreographic programming, whether or not locations are made explicit in a type system like Choral's.

and take the appropriate branch.

- **Endpoint projection.** By itself, a choreography is useful as a global specification of the behavior of a protocol. If we wish to have a runnable implementation, however, we need a way to perform *endpoint projection* (EPP).[2] EPP transforms a choreography into an individual program for each target node. For the bookseller protocol, the Choral compiler carries out EPP and generates Java programs similar to those in Figure 2.1.

So far, we have been using Choral to illustrate the key concepts of CP. Choral exemplifies *language-level* CP, where choreographies are programs in a standalone language with their own syntax, type system, compiler, and so on. Nearly all existing choreographic programming languages are implemented as standalone languages. Let us now turn our attention to *library-level* CP, where CP is implemented as a library in an existing general-purpose programming language.

2.2 CP implemented as a Library

As discussed in Chapter 1, the only existing library-level CP implementation is the HasChor framework [22]. It implements CP as a Haskell library, and it represents the current state of the art of library-level CP. Figure 2.3 shows an implementation of the bookseller protocol as a Haskell program using HasChor.

[2]Unlike in the literature on multiparty session types [15], in which endpoint projection refers to projecting a *global type* to a collection of *local types*, in choreographic programming we are concerned with projecting a *global program* (that is, a choreography) to a collection of *local programs*.

```
1  bookseller :: Choreo IO (Maybe Day @ "buyer")
2  bookseller = do
3    title'   <- (buyer, title) ~> seller
4    price    <- seller `locally` \un -> return (priceOf (un title'))
5    price'   <- (seller, price) ~> buyer
6    decision <- buyer `locally` \un -> return (un price' <= budget)
7
8    cond (buyer, decision) \case
9      True -> do
10       date  <- seller `locally` \un -> return (deliveryDate (un title'))
11       date' <- (seller, date) ~> buyer
12       buyer `locally` \un -> return $ Just (un date')
13     False -> do
14       buyer `locally` \_ -> return Nothing
```

Figure 2.3: The bookseller protocol implemented in Haskell using HasChor [22]

With HasChor, choreographies are written as computations that run in the `Choreo` monad provided by the library. The `bookseller` choreography's type signature on line 1 of Figure 2.3 shows that it returns a value of type `Maybe Day` at the `"buyer"` location. In general, HasChor supports *located values* of type `a @ l`, implemented using GHC Haskell's support for type-level symbols.

The `(~>)` operator, seen on lines 3, 5, and 11 of Figure 2.3, implements communication between a sender and a receiver and is HasChor's counterpart of the `com` method in Choral. The `locally` operator, on lines 4, 6, 10, 12, and 14 of Figure 2.3, implements local computation at a particular node — for instance, looking up the book's price on the seller's node (line 4), and computing whether the book is in budget on the buyer's node (line 6). A located value of type `a @ l` may be "unwrapped" and used at the specified location `l` using the special `un` function passed to `locally`. Finally, the `cond` operator on line 8 of Figure 2.3 implements a choreographic conditional expression. Unlike with the Choral bookseller implementation in Figure 2.2, the HasChor

programmer does not need to use anything like `select` to solve the knowledge-of-choice problem. Instead, in HasChor, `cond` *automatically* inserts the necessary communication to propagate knowledge of choice. While this design choice saves the programmer the tedium of writing calls to `select`, it has unfortunate consequences for efficiency, as we will discuss in Chapter 4.

To run `Choreo` computations, the HasChor framework provides a `runChoreography` function that performs endpoint projection. Given a choreography of `Choreo` type (such as `bookseller`) and a location name (such as `"buyer"`), `runChoreography` acts something like a just-in-time compiler: it dynamically generates (and runs) a node-local program at the specified location, by dynamically interpreting the choreography. This approach to library-level EPP is possible because HasChor implements `Choreo` as a *freer monad* [17], whose operations can be given different semantics depending on the location at which they are run. For instance, in HasChor the `~>` operator is interpreted as `send` for the sender, `receive` for the receiver, and as a no-op for other participants in a choreography.

Library-level CP has the usual advantages of embedding a DSL in an existing host language, including ability to piggyback on the host language's ecosystem and tooling, a gentle learning curve for host-language users, and seamless integration with existing host-language code. HasChor enjoys all of these advantages. It is therefore tempting to directly port the HasChor library to lots of languages in which programmers might benefit from CP. A world with PyChor, JSChor, JavaChor and RustChor libraries would surely make CP more practical and accessible than it is today. Unfortunately,

this "port HasChor to your favorite language" plan has some flaws:

- *Tight coupling with Haskell and monads.* HasChor's monadic implementation approach relies on Haskell-specific language features. While these implementation choices are appropriate (and elegant) in the context of Haskell, doing the same in other languages without proper support for monads (e.g., the do-notation) would hurt the ergonomics of the library. To make choreographic programming more widely accessible, a more general approach to implementing library-level CP is called for.

- *Inefficient conditionals.* HasChor's implementation of conditionals in choreographies involves broadcasting the value of the condition expression to *all* nodes participating in the choreography, even those nodes that are not involved in the execution of the conditional. Implementing conditionals efficiently is a particular challenge for library-level CP: while standalone choreographic languages can statically analyze choreographies to insert only the minimum ammount of inter-node communication needed, such an analysis would be difficult (if not impossible) to accomplish in HasChor, given its implementation approach that relies on dynamic interpretation of free monads. Therefore, HasChor's implementation of conditionals is unlikely to scale well to systems with large numbers of nodes, or those where network bandwidth is a bottleneck.

- *Lack of support for located arguments and return values.* One of the biggest advantages of library-level CP is that it can easily be integrated with existing host-

language code. However, HasChor does not support providing located arguments to choreographies or returning located values from choreographies. This limitation makes it difficult to use HasChor as part of a larger application.

In summary, HasChor aims to make CP easy to *use*, and it succeeds at that goal — provided that the user is a Haskell programmer. But the HasChor design does not make CP easy to *implement* in one's language of choice, and it suffers from efficiency and practicality drawbacks. Our aim in the rest of this study is to democratize the *implementation* of library-level choreographic programming while improving its efficiency and practicality.

Chapter 3

Endpoint Projection as Dependency Injection

A central concept of CP is that a single choreography exhibits different behaviors depending on the location to which it is projected. Each local computation may or may not be executed, and each communication becomes a send, a receive, or a no-op. Allowing a caller (in this case, endpoint projection) to modify the behavior of the callee (in this case, a choreography) is a common pattern in software engineering to improve code reusability and testability. One technique to achieve this is through dependency injection (DI) [10]. In DI, the callee receives its dependencies from the caller, who can alter the callee's behavior by providing different dependencies.

In this section, we present endpoint projection as dependency injection (EPP-as-DI), a new technique for implementing library-level choreographic programming. The

$$
\begin{aligned}
a &: \mathsf{Type} \\
l &: \mathsf{Location} \\
a \mathbin{@} l &= \mathsf{Local}\ a + \mathsf{Remote} && \text{(Located Values)} \\
\mathsf{Unwrap}\ l &= a \mathbin{@} l \to a && \text{(Unwrap)} \\
\mathsf{Choreo}\ a &= \mathsf{Ops} \to a && \text{(Choreography)} \\
\mathsf{Ops} &= \mathsf{Locally} \times \mathsf{Comm} \times \mathsf{Bcast} && \text{(Choreographic Operators)} \\
\mathsf{Locally} &= \forall a.\ \ (l : \mathsf{Location}) \to (\mathsf{Unwrap}\ l \to a) \to a \mathbin{@} l && \text{(Local Computation)} \\
\mathsf{Comm} &= \forall a.\ \ (s\ r : \mathsf{Location}) \to a \mathbin{@} s \to a \mathbin{@} r && \text{(Communication)} \\
\mathsf{Bcast} &= \forall a.\ \ (l : \mathsf{Location}) \to a \mathbin{@} l \to a && \text{(Broadcast)}
\end{aligned}
$$

Figure 3.1: The interface provided by the host-language library for expressing choreographies.

key idea of EPP-as-DI is that we can implement CP by representing a choreography as a host-language function that takes *choreographic operators* as arguments. Then, endpoint projection can change the behavior of the choreography by injecting specialized implementations of the choreographic operators, depending on the projection target. This technique can be used in any host language that supports higher-order functions, enabling the straightforward implementation of choreographic programming libraries in a wide variety of languages. We introduce a simple host language as a stand-in for an arbitrary host language in Section 3.1, then show how EPP-as-DI is implemented in Section 3.2.

3.1 Choreographies as Host-Language Programs

To introduce EPP-as-DI, we assume a simple ML-like host language that supports higher-order functions. For ease of exposition in this section, our host language

is typed; however, types are not essential to implement EPP-as-DI. Choreographies are expressed as host-language functions using the interface presented in Figure 3.1, which we now describe.

3.1.1 Located Values

We assume a set of Locations with decidable equality and write them as l. A *located value*, written $a\,@\,l$, is a value of type a at location l. A located value can either be a Local a, meaning the value is at the current location, or a Remote, meaning the value is at some remote location. We maintain the invariant that, when doing endpoint projection for l, $a\,@\,l$ is always a Local. To use a located value at l, it needs to be *unwrapped* first. Since it does not make sense to unwrap a remote value, we provide an Unwrap l function that can only unwrap values at l. Given a value of type $a\,@\,l$, Unwrap l produces a value of type a.

3.1.2 Choreographies

A choreography Choreo a is a function that takes a set of choreographic operators Ops as dependencies and returns some result of type a. The host-language library interface provides three choreographic operators that are sufficient to realize the key elements of CP described in Section 2.1. We will use lower-case locally, comm, and bcast as the names of operators that have types Locally, Comm, and Bcast, respectively:

- locally performs a local computation: it takes a location and a function and runs the function locally at the location.

```
bookseller : Choreo (Option Date @ buyer)
bookseller(locally, comm, bcast) =
  let title_buyer = locally(buyer, λ(un) → input()) in
  let title_seller = comm(buyer, seller, title_buyer) in
  let price_seller = locally(seller, λ(un) → catalog.get_price(un(title_seller))) in
  let price_buyer = comm(seller, buyer, price_seller) in
  let decision_buyer = locally(buyer, λ(un) → un(price_buyer) ≤ budget) in
  let decision = bcast(buyer, decision_buyer) in
  if decision then
    let delivery_seller = locally(seller, λ(un) → catalog.get_delivery(un(title_seller))) in
    let delivery_buyer = comm(seller, buyer, delivery_seller) in
    locally(buyer, λ(un) → Some(un(delivery_buyer)))
  else
    locally(buyer, λ(un) → None)
```

Figure 3.2: Bookseller Choreography

- comm communicates a value between two locations: it takes a sender and a receiver location, a value at the sender, and returns the same value at the receiver.

- bcast broadcasts a value to the group of locations involved in the interaction: it takes a sender location, a value at the sender, and returns a value at all locations.

Note that, for the sake of simplicity, the host language of Figure 3.3 is dependent-typed to encode the constraints on the locations at the type level, such as Unwrap l and a @ s. In practice, we can encode the same constraints without dependent types using generics and something comparable to singleton types. In Chapter 5 and Chapter 6, we show how to implement EPP-as-DI in Rust and TypeScript, respectively, which do not support dependent types.

We can write choreographies as functions of type Choreo a by using the pro-

```
epp : Choreo a → [Location] → Location → a
epp(c, ls, l) =
  let unwrap(v) = if let Local(a) = v then a else error("impossible") in
  let locally(l′, f) = if l == l′ then Local(f(unwrap)) else Remote in
  let comm(s, r, a) =
    if l == s then send(unwrap(a), r); Remote else if l == r then Local(recv(s)) else Remote in
  let bcast(s, a) = if l == s then ∀r ∈ ls. send(unwrap(a), r); unwrap(a) else recv(s) in
  c(locally, comm, bcast)
```

Figure 3.3: Endpoint Projection as Injecting Dependencies

vided choreographic operators in the body of the function. To illustrate, Figure 3.2 shows the bookseller protocol implemented in our notional host language using the API of Figure 3.1. We assume that the host language supports standard language constructs such as **let ... in** and **if ... then ... else**. The bookseller choreography uses bcast to propagate knowledge of choice and implement conditionals. When the buyer makes a decision ($\mathsf{decision}_{\mathsf{buyer}}$), it is broadcasted to all locations (decision). Since all locations have the same data, it is safe to use the control-flow constructs of the host language, such as **if**, to implement conditionals in choreographies.

3.2 Endpoint Projection as Injecting Dependencies

Since a choreography is a function that takes choreographic operators as dependencies, we can determine the meaning of these operators by injecting specialized implementations of them, leading to the definition of endpoint projection as a host-language function epp, shown in Figure 3.3. We assume the existence of send and recv functions in the host language that implement message transport, for instance, by

calling into a host-language networking library. epp takes a choreography c, a list of locations participating in the choreography ls, and a target location l, then projects the choreography to a node-local program for the target location. Inside epp, we construct the three choreographic operators from the viewpoint of l and supply them to c:

- For operator $\mathsf{locally}(l', f)$, if l is the same as l', we perform the local computation f; otherwise, no action is taken.
- For operator $\mathsf{comm}(s, r, a)$, if l is the same as the sender location s, we perform a send of a to the receiver; or if l is the same as the receiver location r, we perform a recv from the sender; otherwise, no action is taken.
- For operator $\mathsf{bcast}(s, a)$, if l is the same as the sender location s, we perform a series of sends of a to all the locations participating in the interaction; otherwise, no action is taken.

We implemented two library-level choreographic programming libraries using EPP-as-DI: ChoRus for Rust and Choreography.ts for TypeScript. Implementing a CP library in a real-world host language requires additional considerations, such as type-level encoding of located values, ergonomics of the user-facing API, and integration with the host language's ecosystem. We describe their design and implementation in Chapter 5 and Chapter 6.

Chapter 4

Efficient Conditionals with Choreographic Enclaves

As discussed in Section 2.1, implementing conditionals in choreographic programming is challenging because of the "knowledge of choice" problem [5]. A CP language must ensure — either statically or dynamically — that choreographies propagate knowledge of the outcome of evaluating a conditional expression to all locations that are affected by the choice. If CP is implemented as a standalone language, then the compiler can perform static analysis to check this property, and a choreography that fails to propagate knowledge of choice is deemed *unprojectable*. Standalone CP languages can even support choreography *amendment* [7, 18, 1, 6], a procedure that determines if a choreography is unprojectable as-is and then automatically inserts the minimum necessary communication to make it projectable.

Without access to the full AST of programs in the CP language, however, static analysis becomes infeasible. In particular, with both the EPP-as-DI approach of Chapter 3 and HasChor's freer-monad-based approach, it is not trivial to perform static analysis on choreographies to determine how knowledge of choice needs to be propagated. With static analysis off the table as an option, the propagation of knowledge of choice needs to be handled some other way. In Chapter 3, we solved the problem in a naive way by implementing conditionals with broadcast, which ensures that *all* locations receive the knowledge of choice, whether they are affected by the choice or not. HasChor's `cond` operator internally uses broadcast as well. Not only does this naive approach introduce unnecessary communication, it may cause an undesired leak of information to locations that should not have it.

Alternatively, we could do without static analysis another way: by requiring the programmer to provide annotations to convey their intent. In fact, in the absence of choreography amendment, this is the typical approach even in standalone CP languages: the programmer must annotate the branches of a conditional with selection annotations that indicate to the compiler that knowledge of choice must be propagated, as we see in the Choral code in Figure 2.2 that uses the `select` method. Yet the approach of adding selection annotations is somewhat unsatisfying because we must add annotations to make our code *correct* (that is, projectable). If we must annotate our code for the benefit of the compiler, it would be preferable if we could begin with a choreography that is *correct, but inefficient*, and then add annotations to make it *efficient*.

In this section, we address this design challenge with *choreographic enclaves*[1], a novel CP language feature. Enclaves are sub-choreographies that execute at a specified subset of the locations involved in a larger choreography. One may broadcast within an enclave, just like in any other choreography, but the broadcast will only go to those locations that are in the specified subset. Enclaves allow finer control over the propagation of knowledge of choice, enabling an efficient implementation of conditionals in library-level CP without static analysis.

In Section 4.1, we present a variant of the bookseller protocol to motivate the need for fine-grained control over the propagation of knowledge of choice. Then, in Section 4.2 we introduce choreographic enclaves. We define an enclave operator and present its type signature and implementation, and we show how to implement the two-buyer protocol with enclave and compare it with the naive approach and the selection-annotation approach.

4.1 The Two-Buyer Protocol

To illustrate the problem of inefficient conditionals, let us consider a variant of the bookseller protocol: the *two-buyer* protocol [15, 13]. In this protocol, there are two buyers who wish to collectively buy a book from the seller. First, buyer1 sends the title to the seller, and the seller sends the price to both buyers. Then, buyer2 tells buyer1 how much they can contribute, and buyer1 decides whether to buy the book by comparing

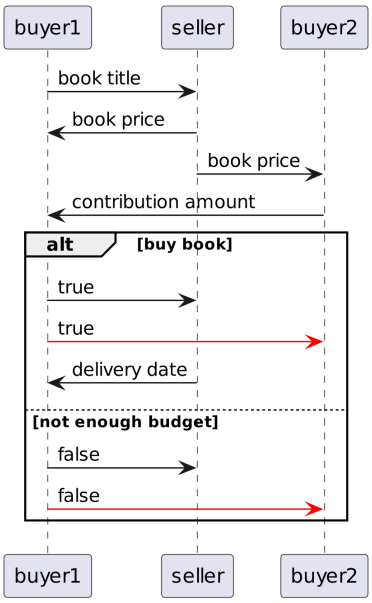
[1] Choreographic enclaves bear no direct relation to secure enclaves like Intel's SGX technology, despite both concepts involving the concealment of information within a smaller component of the overall system.

```
two_buyer : Choreo (Option Date @ buyer1)
two_buyer(locally, comm, bcast) =
  let title_buyer1 = locally(buyer1, λ(un) → input()) in
  let title_seller = comm(buyer1, seller, title_buyer1) in
  let price_seller = locally(seller, λ(un) → catalog.get_price(un(title_seller))) in
  let price_buyer1 = comm(seller, buyer1, price_seller) in
  let price_buyer2 = comm(seller, buyer2, price_seller) in
  let contribution = comm(buyer2, buyer1, buyer2_budget) in
  let decision_buyer1 = locally(buyer1, λ(un) → un(price_buyer1) ≤ buyer1_budget + contribution) in
  let decision = bcast(buyer1, decision_buyer1) in
  if decision then
    let delivery_seller = locally(seller, λ(un) → catalog.get_delivery(un(title_seller))) in
    let delivery_buyer1 = comm(seller, buyer1, delivery_seller) in
    locally(buyer1, λ(un) → Some(un(delivery_buyer1)))
  else
    locally(buyer1, λ(un) → None)
```

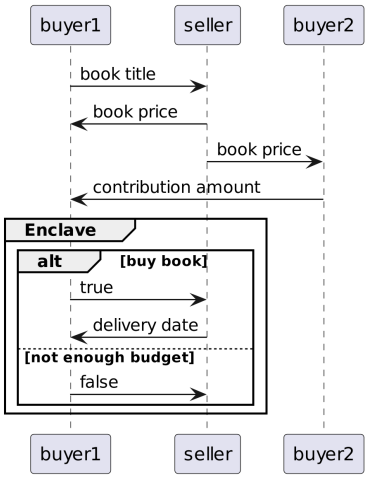
Figure 4.1: A naive version of the two-buyer protocol with bcast

the price with the buyers' combined budget. If buyer1 decides to buy the book, they send their intent to buy to the seller, and the seller sends the delivery date to buyer1. Otherwise, buyer1 tells the seller that they will not buy the book.

Using the bcast choreographic operator that we introduced in Section 3.1, we can implement the two-buyer protocol in our notional host language, as shown in Figure 4.1. Figure 4.2a shows a sequence diagram of the execution of the protocol. After buyer1 makes a decision, to perform the conditional, it broadcasts the decision to all locations, i.e., the seller and buyer2. It is important that the seller receives the decision because the seller needs to know whether to send a delivery date to buyer1, but buyer2 does not need to receive the decision, as its subsequent behavior does not depend on it.



(a) Using broadcast naively



(b) With enclave

Figure 4.2: Sequence diagrams of the two-buyer protocol with and without enclave

Nonetheless, because of the broadcast, buyer2 receives the decision, causing unnecessary communication between buyer1 and buyer2, shown in red in Figure 4.2a. While this communication does not affect the correctness of the choreography, it is inefficient and can be problematic in more complex choreographies with many participants. Moreover, it leaks information about choice, which can be a security concern. For example, buyer2 might infer the budget of buyer1 by observing the decision, and this type of information leakage might be undesirable in some applications.

4.2 The Enclave Operator

To prevent unnecessary communication, we introduce the enclave choreographic operator. The enclave operator executes a sub-choreography at a specified set of loca-

tions. Inside the sub-choreography, the broadcast operator sends data only to locations in the specified set. This allows us to perform conditionals without sending data to unaffected locations.

We modify the interface of our host-language library from Figure 3.1 to add support for an enclave operator with the type Enclave, specified below:

$$\mathsf{Choreo}\ a = \mathsf{Ops} \to a \qquad \text{(Choreography)}$$

$$\mathsf{Ops} = \mathsf{Locally} \times \mathsf{Comm} \times \mathsf{Bcast} \times \mathsf{Enclave} \qquad \text{(Choreographic Operators)}$$

$$\mathsf{Enclave} = \forall a, l.\ [\mathsf{Location}] \to \mathsf{Choreo}\ a@l \to a@l \qquad \text{(Enclave)}$$

The first argument to enclave is a list of locations where the sub-choreography is to be executed, and the second argument is the sub-choreography. It returns the result of running the sub-choreography. To implement endpoint projection for enclave, we update the definition of epp from Figure 3.3 as follows:

$$\mathsf{epp} : \mathsf{Choreo}\ a \to [\mathsf{Location}] \to \mathsf{Location} \to a$$

$$\mathsf{epp}(c, ls, l) =$$

$$\quad \ldots$$

$$\quad \textbf{let}\ \mathsf{enclave}(ls', c') = \textbf{if}\ l \in ls'\ \textbf{then}\ \mathsf{epp}(c', ls', l)\ \textbf{else}\ \mathsf{Remote}\ \textbf{in}$$

$$\quad c(\mathsf{locally}, \mathsf{comm}, \mathsf{bcast}, \mathsf{enclave})$$

The enclave operator recursively calls the sub-choreography by calling epp with the sub-choreography and the list of locations where the sub-choreography is executed if the projection target is one of the specified locations. The behavior of bcast inside the

```
two_buyer : Choreo (Option Date @ buyer1)
two_buyer(locally, comm, bcast, enclave) =
  ...
  let decision_buyer1 = locally(buyer1, λ(un) → un(price_buyer1) ≤ buyer1_budget + contribution) in
  let c(locally, comm, bcast, enclave) =
    let decision = bcast(buyer1, decision_buyer1) in
    if decision then
      let delivery_seller = locally(seller, λ(un) → catalog.get_delivery(un(title_seller))) in
      let delivery_buyer1 = comm(seller, buyer1, delivery_seller) in
      locally(buyer1, λ(un) → Some(un(delivery_buyer1)))
    else
      locally(buyer1, λ(un) → None)
  in
  enclave([buyer1, seller], c)
```

Figure 4.3: A more efficient version of the two-buyer protocol using an enclave

sub-choreography depends on the ls' argument to the recursive call to epp, so bcast inside the sub-choreography will only send data to the specified locations. Using enclave, we can rewrite the last part of the two-buyer protocol, as shown in Figure 4.3. After buyer1 makes a decision, we define a sub-choreography c that uses bcast to perform conditionals. Then, we call the sub-choreography at buyer1 and seller using enclave. Because the sub-choreography is not executed at buyer2, bcast does not send the decision to buyer2, as shown in Figure 4.2b.

While we have shown how to implement endpoint projection for enclave using the EPP-as-DI technique, the use of choreographic enclaves is orthogonal to the use of EPP-as-DI. For instance, one could extend HasChor with an enclave operator without departing from HasChor's freer-monad-based implementation of EPP.

Chapter 5

ChoRus

This chapter presents *ChoRus*, a choreographic programming library for Rust. ChoRus is implemented using EPP-as-DI, supports choreographic enclaves, and has other features that make it a practical choice for distributed programming in Rust. We describe how we encode EPP-as-DI in Rust Section 5.1, and give a brief tour of ChoRus features Section 5.2. The code shown in this section is simplified for presentational purposes. ChoRus is open source, and its implementation, case studies and benchmarking code, and documentation are available at `https://github.com/lsd-ucsc/ChoRus`.

5.1 EPP-as-DI in ChoRus

5.1.1 Locations

ChoRus represents each location at which node-local code runs as a distinct type. In Rust, we can create a new type by defining a struct. Locations must be

comparable for equality to perform endpoint projection. To that end, ChoRus defines the `ChoreographyLocation` trait, which all location types must implement:

```
trait ChoreographyLocation: Copy {
    fn name() -> &'static str;
}
```

The `name` method returns the string representation of the location, which is used to compare locations for equality. Thanks to Rust's macro system, `ChoreographyLocation` can be derived automatically. For example, the following code defines a location named `Alice`:

```
#[derive(ChoreographyLocation)]
struct Alice;
```

5.1.2 Located Values

Located values are values that reside at a specific location. ChoRus defines the `Located<V, L1>` struct to represent a located value of type `V` at location `L1`:

```
struct Located<V, L1: ChoreographyLocation> {
    value: Option<V>,
    phantom: PhantomData<L1>,
}
```

The `value` field holds a value of type `Option<V>`; it is `Some` if the current projection target is `L1` and `None` otherwise. We use `std::marker::PhantomData` to indicate to the compiler that the `L1` parameter is not used at run time.

5.1.3 Choreography Trait

In Chapter 3, we represented choreographies as functions. To provide a more ergonomic API, ChoRus represents choreographies as structs that implement the `Choreography` trait. The `Choreography` trait is defined as follows:

```
trait Choreography<R = ()> {
    fn run(self, op: &impl ChoreoOp) -> R;
}
```

The `R` type parameter represents the return type of the choreography. The `run` method takes a reference to an object that implements the `ChoreoOp` trait, which provides the choreographic operators.

5.1.4 ChoreoOp Trait

ChoRus supports the four choreographic operators `locally`, `comm`, `broadcast`, and `enclave`, as described in Chapter 3 and Chapter 4.

Figure 5.1 shows an excerpt of the `ChoreoOp` trait that implements the `locally` operator. The `locally` method takes a location `location` and a function `computation` and returns a `Located` value. The `computation` function takes an argument of type `Unwrapper<L1>`, which it can use to unwrap located values at location `L1`. Other choreographic operators are defined similarly as methods of the `ChoreoOp` trait.

```rust
trait ChoreoOp {
    fn locally<V, L1: ChoreographyLocation>(
        &self,
        location: L1,
        computation: impl Fn(Unwrapper<L1>) -> V,
    ) -> Located<V, L1>;
    // ...
}
```

Figure 5.1: The `ChoreoOp` trait (excerpt).

5.1.5 Transport

The `Transport` trait represents the message transport layer. Users can implement the `Transport` trait by providing the `send` and `receive` methods. ChoRus has two built-in transport implementations: `LocalTransport` and `HttpTransport`. The `LocalTransport` implementation models each location as a thread and uses an inter-thread channel to send messages. The `HttpTransport` implementation uses HTTP to send messages.

5.1.6 Endpoint Projection

ChoRus provides the `Projector` struct to perform endpoint projection and execute choreographies. First, users construct a `Projector` by passing the projection target and the transport. Then, they can call the `epp_and_run` method to perform endpoint projection and execute the choreography. The `epp_and_run` method takes a choreography, defines `EppOp` — an object that implements `ChoreoOp` for the projection target — and calls the `run` method of the choreography with it. Figure 5.2 shows an excerpt of the `epp_and_run` method and the implementation of `locally`.

```
impl<...> Projector<...> {
    pub fn epp_and_run<...>(&'a self, choreo: C) -> V {
        struct EppOp<...> {...}
        impl<...> ChoreoOp for EppOp<...>
        {
            fn locally<V, L1: ChoreographyLocation>(
                &self,
                location: L1,
                computation: impl Fn(Unwrapper<L1>) -> V,
            ) -> Located<V, L1> {
                if L1::name() == Target::name() {
                    let value = computation(Unwrapper::new());
                    Located::local(value)
                } else {
                    Located::remote()
                }
            }
            // ...
        }
        choreo.run(&EppOp {...})
    }
}
```

Figure 5.2: The `epp_and_run` method of the `Projector` struct.

5.2 Advanced Features

ChoRus supports all the features supported by HasChor [22], such as swappable transport backends, higher-order choreographies, and location polymorphism. In this section, we present two new features of ChoRus: *location sets* and *located input/output*.

5.2.1 Location Sets

In ChoRus, the set of locations at which a choreography runs is represented at the type level. We call this type the *location set* of the choreography. Each choreography has an associated type `L` that represents its location set. `ChoreoOp` is parametrized by the location set of the choreography and prevents users from using locations that are not in the location set. For example, Figure 5.3 defines a choreography `AliceBobChoreography`

```
struct AliceBobChoreography;
impl Choreography for AliceBobChoreography {
    type L = LocationSet!(Alice, Bob);
    fn run(self, op: &impl ChoreoOp<Self::L>) {
        op.locally(Carol, |_| println!("Hello from Carol!"));
    }
}
```

Figure 5.3: Invalid use of location `Carol` in `AliceBobChoreography`.

that runs on locations `Alice` and `Bob`. `LocationSet!` is a macro that constructs a special location set type. Inside the `run` method, we can only use locations `Alice` and `Bob`. If we try to use location `Carol`, the Rust compiler will report an error. Location sets are especially useful when defining choreographic enclaves, as they prevent us from accidentally using locations outside the enclave.

5.2.2 Located Input/Output

When a choreography is used as part of a larger program, it is often useful to be able to pass *located* values to and from the choreography. For example, consider a simple password authentication protocol between a client and a server. The client reads a password from the user and sends it to the server. The server checks the password and sends the result back to the client. The client then prints the result. The inputs to this choreography are (1) the typed password on the client, and (2) the correct password on the server, and the output is the result of the authentication on the client. Morally, these are all *located* values; for example, when running the choreography on the client, we do not have access to the correct password on the server. However, from outside the choreography, we do not have a way to talk about their locations. To solve this

problem, ChoRus provides a *located input/output* feature that provides a convenient and type-safe way to handle located values.

`Projector` plays an important role in the located input/output feature. `Projector` is parameterized by the projection target and can construct (1) local located values at the projection target, and (2) remote located values at other locations. It can also unwrap located values at the projection target, but not at other locations.

Figure 5.4 shows the password authentication choreography and code to execute the choreography as the client and as the server. When running the choreography as the client, we use an instance of `Projector` that is parameterized by the client location. We provide the password attempt as a local located value and the correct password as a remote located value on the server. Conversely, when running the choreography as the server, we provide the password attempt as a remote value and the correct password as a local value. The result can only be unwrapped at the client location using the `unwrap` method of `Projector`.

```
struct PasswordAuthChoreography {
    attempt_password: Located<String, Client>,
    correct_password: Located<String, Server>,
}
impl Choreography<Located<bool, Client>> for PasswordAuthChoreography {
    type L = LocationSet!(Client, Server);
    fn run(self, op: &impl ChoreoOp<Self::L>) -> Located<bool, Client> {
        let password = op.comm(Client, Server, &self.attempt_password);
        let result = op.locally(Server, |un| {
            un.unwrap(&password) == un.unwrap(&self.correct_password)
        });
        op.comm(Server, Client, &result)
    }
}
```

(a) Password authentication choreography

```
let result = client_projector.epp_and_run(PasswordAuthChoreography {
    attempt_password: client_projector.local("1234".to_string()),
    correct_password: client_projector.remote(Server),
});
println!("Result: {}", client_projector.unwrap(result));
```

(b) Client code

```
server_projector.epp_and_run(PasswordAuthChoreography {
    attempt_password: server_projector.remote(Client),
    correct_password: server_projector.local("password".to_string()),
});
```

(c) Server code

Figure 5.4: The password authentication choreography (a), along with node-local code to invoke it on the client (b) and server (c).

Chapter 6

Choreography.ts

EPP-as-DI is a general technique that can be applied to any programming language that supports higher-order functions. In this chapter, we present *Choreography.ts*, a choreographic programming library for TypeScript. TypeScript is a statically typed language that builds on JavaScript by adding static type definitions, enabling better tooling and error-checking capabilities. It has unique type system features that make it an interesting case study for implementing EPP-as-DI. The source code for Choreography.ts is available on GitHub at `https://github.com/shumbo/choreography-ts`.

6.1 Unique Features of TypeScript

When implementing an embedded DSL, the host language's features play a significant role in its design and implementation. TypeScript has several unique features that are useful for implementing EPP-as-DI. We discuss some of these features in this

section.

6.1.1 String Literal Types

TypeScript has a feature called *string literal types* that allows developers to define types that accept only specific string values. For example, the following code defines a type `Hello` that can only be `"hello"`:

```
type Hello = "hello";
```

A variable of type `Hello` can only be assigned the value `"hello"`. Assigning any other value will result in a type error.

```
let hello: Hello = "hello"; // OK

let world: Hello = "world"; // Type error
```

String literal types are useful to constrain the string values that a variable can take. Choreography.ts uses string literal types to represent locations and location sets.

6.1.2 Union Types

TypeScript has a feature called *union types* that allows developers to define types that can be one of several possible types. For example, the following code defines a type `Direction` that can be either `"left"` or `"right"`:

```
type Direction = "left" | "right";
```

`left` and `right` are string literal types, and `Direction` is a union type that can be either `"left"` or `"right"`. A variable of type `Direction` can be assigned either `"left"` or `"right"`.

```
let left: Direction = "left"; // OK
let right: Direction = "right"; // OK
let up: Direction = "up"; // Type error
```

Union types can naturally represent a set of types. Choreography.ts uses union types to represent a set of locations that a choreography can use.

6.1.3 Generic Constraints

TypeScript has a feature called *generic constraints* that allows developers to define constraints on generic type parameters. For example, the following code defines a function `direction_id` that takes a direction and returns the same direction:

```
function direction_id<T extends Direction>(d : T): T {
    return d;
}
```

`direction_id` is a generic function that takes a type parameter `T` that extends the `Direction` type. It takes a value of type `T` and returns the same value. The type parameter `T` is constrained to be a subtype of `Direction`, which means that it can only be either `"left"` or `"right"`. This ensures that the function only accepts valid directions and we can use the type parameter `T` to encode additional constraints. We will use generic constraints to ensure that choreographic operators can only use the correct locations.

```
let left: "left" = direction_id("left"); // OK
let right: "right" = direction_id("right"); // OK
```

```
let err = direction_id("top"); // Type error

let mismatch: "left" = direction_id("right"); // Type error
```

Now that we have discussed some of the unique features of TypeScript, we can proceed to the design and implementation of Choreography.ts.

6.2 Design and Implementation of Choreography.ts

6.2.1 Locations

Choreography.ts uses string literal types and their unions to represent locations and location sets. A location is defined as a string literal type. For example, `type Alice = "Alice";` defines a location `Alice`. A location set is defined as a union of location types. For example, `type L = "Alice" | "Bob";` defines a location set `L` that can be either `Alice` or `Bob`.

6.2.2 Located Values

Located values are implemented in a similar way to ChoRus. Figure 6.1 shows the implementation of located values in Choreography.ts. A located value is a generic class that takes a type parameter `T` and a location type parameter `L1`. It has a protected field `value` of type `T` and an optional field `phantom` of type `L1`. The `phantom` field is there to prevent illegal assignments of located values to different locations. For example, a located value of type `Located<number, "Alice">` cannot be assigned to a variable of type `Located<number, "Bob">`.

```
class Located<T, L1 extends string> {
  protected value: T;
  protected phantom?: L1;
}
```

Figure 6.1: Located values in Choreography.ts

6.2.3 Choreography and Dependencies

Figure 6.2 shows the type definitions for choreography and choreographic operators. A choreography is a function of type `Choreography`. It takes three type parameters: `L` is a location set, `Args` and `Return` are arrays of located values at some location in `L`. The function takes an object of type `Dependencies` that provides choreographic operators and a set of arguments of type `Args`. It returns a promise of an array of located values of type `Return`.

The definitions of choreographic operators, `locally`, `comm`, `enclave`, and `broadcast` are similar to those in ChoRus, with minor differences. For syntax, we use generic constraints to ensure that choreographic operators can only use the correct locations. For example, the `locally` operator has a type parameter `L1` that extends the location set `L`. This ensures that the location passed to the `locally` operator is a valid location in the choreography. Another difference is that the choreographic operators are asynchronous functions that return promises. Since JavaScript is typically executed in a single-threaded environment, most blocking operations, such as IO, are implemented as asynchronous functions that return promises. For this reason, choreographies and choreographic operators in Choreography.ts are designed to work with promises. In most cases, users can use the `await` keyword to wait for promises to resolve and write

```
type Choreography<
    L extends Location,
    Args extends Located<any, L>[] = [],
    Return extends Located<any, L>[] = []
> = (deps: Dependencies<L>, args: Args) => Promise<Return>;
type Dependencies<L extends Location> = {
    locally: Locally<L>;
    comm: Comm<L>;
    broadcast: Broadcast<L>;
    enclave: Enclave<L>;
};
type Locally<L extends Location> = <L1 extends L, T>(
    location: L1,
    callback: (unwrap: Unwrap<L1>) => T | Promise<T>
) => Promise<Located<T, L1>>;
type Unwrap<L1 extends Location> = <T>(located: Located<T, L1>) => T;
type Comm<L extends Location> = <L1 extends L, L2 extends L, T>(
    sender: L1,
    receiver: L2,
    value: Located<T, L1>
) => Promise<Located<T, L2>>;
type Enclave<L extends Location> = <
    LL extends L,
    Args extends Located<any, LL>[],
    Return extends Located<any, LL>[]
>(locations: LL[], choreography: Choreography<LL, Args, Return>, args: Args) =>
    Promise<Return>;
type Broadcast<L extends Location> =
    <L1 extends L, T>(sender: L1, value: Located<T, L1>) => Promise<T>;
```

Figure 6.2: Choreography type definitions in Choreography.ts

asynchronous code in a synchronous style. Figure 6.3 shows an example of a two-buyer protocol implemented in Choreography.ts.

6.2.4 Transport

Similar to ChoRus, Choreography.ts provides a transport layer that allows locations to communicate with each other. Figure 6.4 shows the type definitions for the transport layer in Choreography.ts. The `Parcel` type represents a message that is sent between locations. It contains the sender and receiver locations and the message

```
const bookseller: Choreography<L> = async ({ locally, comm, broadcast, enclave }) => {
  const title_at_buyer1 = await locally("buyer1", async () => {
    return await rl.question("Book name?");
  });
  const title_at_seller = await comm("buyer1", "seller", title_at_buyer1);
  const price_at_seller = await locally(
    "seller",
    (unwrap) => lookup(unwrap(title_at_seller)).price
  );
  const price = await broadcast("seller", price_at_seller);
  const contrib_at_buyer2 = await locally("buyer2", async () => {
    return await rl
      .question("How much do you want to contribute for the purchase?")
      .then((x) => parseInt(x));
  });
  const contrib_at_buyer1 = await comm("buyer2", "buyer1", contrib_at_buyer2);
  await enclave(
    ["buyer1", "seller"],
    async ({ locally, broadcast, comm }) => {
      const decision_at_buyer1 = await locally("buyer1", async (unwrap) => {
        const contrib = await rl
          .question("How much do you want to contribute for the purchase?")
          .then((n) => parseInt(n));
        return contrib + unwrap(contrib_at_buyer1) >= price;
      });
      const decision = await broadcast("buyer1", decision_at_buyer1);
      if (decision) {
        const delivery_date_at_seller = await locally(
          "seller",
          (unwrap) => lookup(unwrap(title_at_seller)).delivery_date
        );
        const delivery_date_at_buyer1 = await comm(
          "seller",
          "buyer1",
          delivery_date_at_seller
        );
        await locally("buyer1", (unwrap) => {
          console.log(
            `the book will be delivered on ${unwrap(delivery_date_at_buyer1)}`
          );
        });
      }
      return [];
    },
    []
  );
  return [];
};
```

Figure 6.3: Two-buyer Protocol in Choreography.ts

```
type Parcel<L extends Location> = {
  from: L;
  to: L;
  data: any;
}
abstract class Transport<L extends Location, L1 extends L> {
  public abstract send(parcel: Parcel<L>): Promise<void>;
  public abstract subscribe(cb: (p: Parcel<L>) => void): Subscription;
  private phantom?: L1;
}
```

Figure 6.4: Transport Layer in Choreography.ts

data. The `Transport` class is responsible for sending and receiving messages. The `send` method sends a parcel to the receiver location. The `subscribe` method registers a callback function that is called when a parcel is received.

Choreography.ts comes with three built-in transport implementations: `LocalTransport`, which runs all locations concurrently in the same process; `ExpressTransport`, which uses the Express.js framework [14] to send messages over HTTP; and `SocketIOTransport`, which uses the Socket.IO library [21] to send messages over WebSockets. Users can also implement their own transport layer by extending the `Transport` class and implementing the `send` and `subscribe` methods.

6.2.5 Endpoint Projection

Endpoint projection is accomplished by constructing a `Dependencies` object that provides choreographic operators for the projection target and calls the choreography function with it. Choreography.ts provides the `Projector` class that works similarly to the ChoRus projector but with some differences to better fit TypeScript.

First, because it is easy to work with anonymous functions in TypeScript,

the `epp` method on the `Projector` class takes a choreography and a projection target and returns a new function that takes the arguments of the choreography and returns a promise of the return values. This allows users to potentially reuse the projected choreography multiple times with different arguments.

Second, Choreography.ts takes advantage of TypeScript's type-level programming abilities to provide precise type checking for located arguments and return values. When projecting a choreography with located inputs and outputs, the `epp` method infers the types of arguments and returns values based on the current projection target. Concretely, if a located argument or return value is projected to the same location, the type of the argument or return value is inferred as a normal value. If a located argument or return value is projected to a different location, the type of the argument or return value is inferred as `undefined`. This allows TypeScript to catch type errors when users don't correctly supply the arguments or handle the return values.

Figure 6.5 shows the same password authentication choreography from Section 5.2.2 implemented in Choreography.ts. The argument of the choreography is of type `[Located<string, "client">, Located<string, "server">]`: an attempt password from the client and the correct password from the server. The return value of the choreography is of type `[Located<boolean, "client">]`: a boolean value indicating whether the password is correct. To perform endpoint projection, we instantiate a `Projector` class with a transport and a location. In Figure 6.5b and Figure 6.5c, we assume that `clientTransport` and `serverTransport` are instances of the `Transport` class for the client and server locations, respectively. We create two instances of the

`Projector` class, one for the client and one for the server, and call the `epp` method with the password authentication choreography. The `epp` method returns a function that would behave as the client or server endpoint. In other words, the `epp` method returns a partial application of the choreography with the concrete choreographic operators for the projection target.

Notice that the type of the projected function is inferred based on the projection target. The client endpoint takes an attempt password and `undefined` as a placeholder for the correct password and returns a boolean value. The server endpoint takes `undefined` as a placeholder for the attempt password and the correct password and returns `undefined`. The library automatically wraps/unwraps located values based on the projection target, so users don't have to worry about the details of located values. While the code in Figure 6.5 shows the types of the projected functions, users can omit the types and let TypeScript infer them.

6.2.6 ESLint Plugin

Unlike standalone choreographic programming languages, library-level choreographic programming requires users to follow certain conventions to ensure that the code is correct. The dynamic nature of JavaScript makes it vulnerable to certain types of errors that are difficult to catch with static analysis. To help users write correct choreographies, Choreography.ts comes with an ESLint plugin that enforces best practices for choreographic programming.

For example, one common mistake in Choreography.ts is using choreographic

```
const passwordAuthChoreography: Choreography<
  L,
  [Located<string, "client">, Located<string, "server">],
  [Located<boolean, "client">]
> = async ({ locally, comm }, [attempt_password, correct_password]) => {
  const password = await comm("client", "server", attempt_password);
  const result_at_server = await locally(
    "server",
    (unwrap) => unwrap(password) === unwrap(correct_password)
  );
  const result_at_client = await comm("server", "client", result_at_server);
  return [result_at_client];
};
```

(a) Password authentication choreography

```
const clientProjector = new Projector(clientTransport, "client");
const client:  (args: [string, undefined]) => Promise<[boolean]> =
  clientProjector.epp(passwordAuthChoreography);
const [result] = await client(["attempt_password", undefined]);
console.log("result:", result);
```

(b) Client code

```
const serverProjector = new Projector(serverTransport, "server");
const server: (args: [undefined, string]) => Promise<[undefined]> =
  serverProjector.epp(passwordAuthChoreography);
await server([undefined, "correct_password"]);
```

(c) Server code

Figure 6.5: The password authentication choreography (a), along with node-local code to invoke it on the client (b) and server (c).

operators of outer choreographies in a sub-choreography. For example, suppose that the user forgets to receive choreographic operators in the definition of the sub-choreography on line 19 of Figure 6.3. We can still access `locally`, `comm`, and `broadcast` in the sub-choreography because the same operators for the outer choreography (defined on line 1) are in scope. However, they have different types because the sub-choreography cannot use `buyer2` which is not part of the enclave. This can lead to misuse of locations and possibly deadlocks. The ESLint plugin can catch this error by checking that the choreographic operators in the sub-choreography are received as arguments. When integrated with an IDE, the ESLint plugin can provide real-time feedback to users as they write choreographies. Figure 6.6 shows the ESLint plugin in action, highlighting the error in the sub-choreography in VSCode. The ESLint plugin can fix some of the errors automatically. In this case, by right-clicking on the error and selecting "Fix", the plugin can automatically add the missing choreographic operators as arguments to the sub-choreography and resolve the error.

The ESLint plugin has two more rules: one that checks that all locations in an enclave are used and one that checks that choreographic operators are not renamed to the other checks would function properly.

When implementing choreographic programming at the library level, it is often difficult to prevent all illegal programs only with the type system. Linters can help users write correct choreographies by enforcing conventions and best practices, and I believe that they can play an important role in improving the safety and correctness of library-level choreographic programming.

Figure 6.6: ESLint Plugin showing errors for misused choreographic operators

Chapter 7

Evaluation

In this chapter, we assess the utility and practicality of library-level CP with EPP-as-DI and the Enclave operator. We use ChoRus as a model implementation of library-level CP and evaluate it through two case studies and performance benchmarking. First, to demonstrate that ChoRus indeed brings the advantages of CP to Rust, we present a case study involving a key-value store (Section 7.1). In this case study, we implement a simple replicated key-value store as a choreography and as a traditional Rust program. We compare these two implementations and highlight how choreography helps to track the flow of data and control. Next, to illustrate that library-level CP enables code reuse, we conduct a second case study: a multiplayer tic-tac-toe game (Section 7.2). We begin by showing the code for a tic-tac-toe game that runs locally, then we use ChoRus to modify the program to run across multiple computers over a network with minimal changes. We observe that library-level CP allows a substantial portion of

the local code to be reused for the distributed implementation. Finally, we measure the performance overhead incurred by using ChoRus (Section 7.3). Through benchmarking, we show that ChoRus introduces very minimal overhead, making it sufficiently practical for use.

7.1 Case Study 1: Replicated Key-Value Store

To demonstrate how ChoRus helps developers to implement distributed systems, we consider a simple replicated key-value store. Our key-value store supports two operations: `get` and `put`. The `get` operation takes a key and returns the value associated with the key. The `put` operation takes a key and a value, and associates the key with the value. Our system consists of three nodes: `Client`, `Primary`, and `Backup`. The `Client` node takes a request from the user and sends the request to the `Primary` node. The `Primary` node checks the type of the request. If the request is a `get` request, it looks up the requested key in its local state and returns the response to the client. If the request is a `put` request, it forwards the request to the backup node. The backup node updates its local state and returns the response to the `Primary` node. Once `Primary` receives the response from `Backup`, it applies the update to its local state and returns the response to the client.

While the protocol is simple, implementing it is error-prone. Figure 7.1a shows the implementation of the protocol *without* using choreographic programming. The code defines three functions for each node. The highlight and arrows show the flow of data

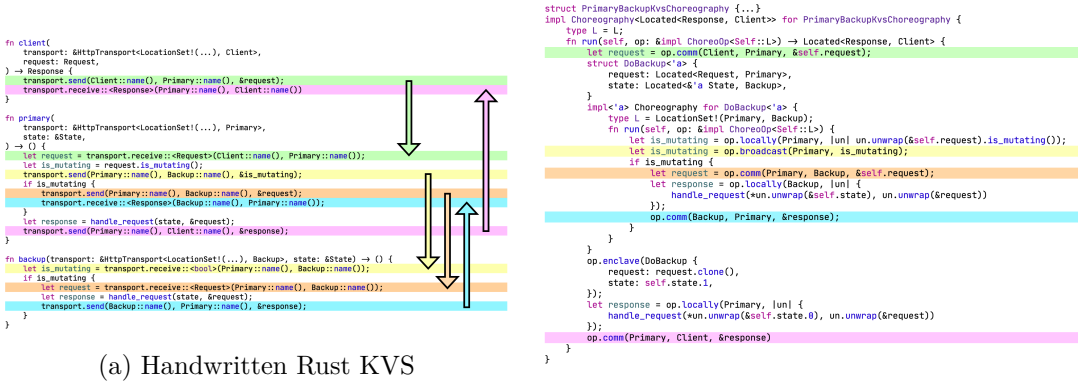
between the nodes. Because sends and receives are interleaved, it is difficult to track the flow of data and control.

Figure 7.1b shows the implementation of the same protocol as a choreography in ChoRus. The choreography communicates the request from `Client` to `Primary` using `comm`. Then, it uses the `enclave` operator to call the `DoBackup` sub-choreography at `Primary` and `Backup`. The sub-choreography branches on the type of the request, and if the request is `put`, it forwards the request to the backup node. After the sub-choreography returns, the primary node processes the request and returns the response to the client. The choreographic version is easier to understand because both data and control naturally flow from top to bottom.

While we could implement the KVS protocol as a choreography in HasChor, the naive implementation of conditionals in HasChor would present a problem. When we branch on the type of the request on the primary node, it broadcasts the type, and in HasChor, this broadcast would also go to the client, leaking an implementation detail. By using enclaves, we can implement the protocol in a more efficient (and secure) manner.

7.2 Case Study 2: Multiplayer Tic-Tac-Toe

An advantage of library-level choreographic programming is that it allows developers to reuse existing code. This is especially useful for implementing a distributed version of an existing local program. In this case study, we implement a distributed ver-



(a) Handwritten Rust KVS

(b) Choreographic ChoRus KVS

Figure 7.1: Comparison of the flow of control between the handwritten and choreographic KVS

sion of a tic-tac-toe game using ChoRus. We start with a local implementation of the game where two players play on the same computer. Then, we use ChoRus to port the local implementation to a distributed version, which lets the players play on different computers over the network, with minimal changes to the code. Finally, we compare the ChoRus implementation with a handwritten distributed version of the game.

Let us start with the local implementation of the game. The left side of Figure 7.2 shows the structure of the main game loop written in Rust. We omit the definitions of the structs and traits that capture the core logic of the game, such as `board`, `brain_for_x`, and `brain_for_o`. The game starts with an empty board. Then, the game enters a loop in which the two players take turns to make a move. After each move, we check the status of the board, and if the game is over, we break out of the loop. Finally, we print the result of the game.

Because ChoRus is a library, we can reuse the existing local Rust code to implement the distributed version of the game. The right side of Figure 7.2 shows the

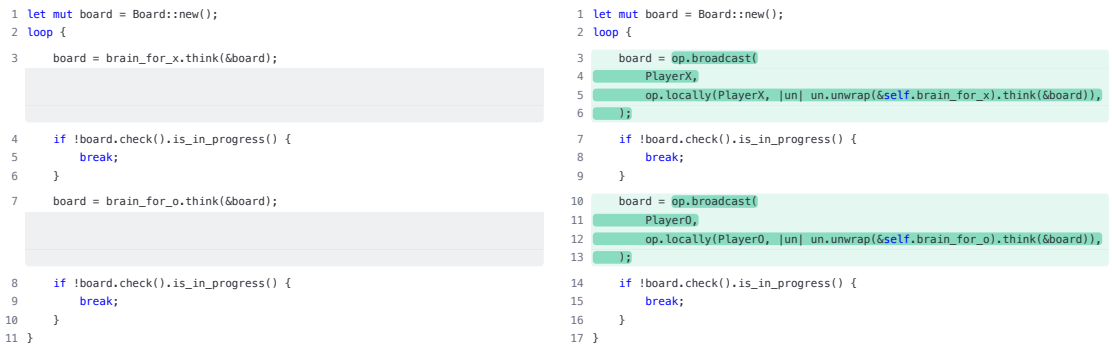


Figure 7.2: Diff between the local Rust and distributed ChoRus implementations of the tic-tac-toe game

distributed implementation of the game written as a choreography in ChoRus. For brevity, we only show the `run` method of the choreography. Just like its local counterpart, the choreography starts with an empty board. Then, the choreography enters a loop in which the two players make a local move and broadcast the new board. After each move, we check the status of the board, and if the game is over, we break out of the loop. Finally, we print the result of the game from the perspective of each player.

As highlighted in Figure 7.2, changing the local implementation to the distributed implementation requires minimal changes to the code. All game logic and control flow are reused, and the only changes are the addition of the `locally` and `broadcast` operators to specify the location of data and computation. This is a significant advantage of library-level CP as opposed to a standalone CP language, because it allows developers to reuse existing code for local computation and focus on the distributed aspects of the program.

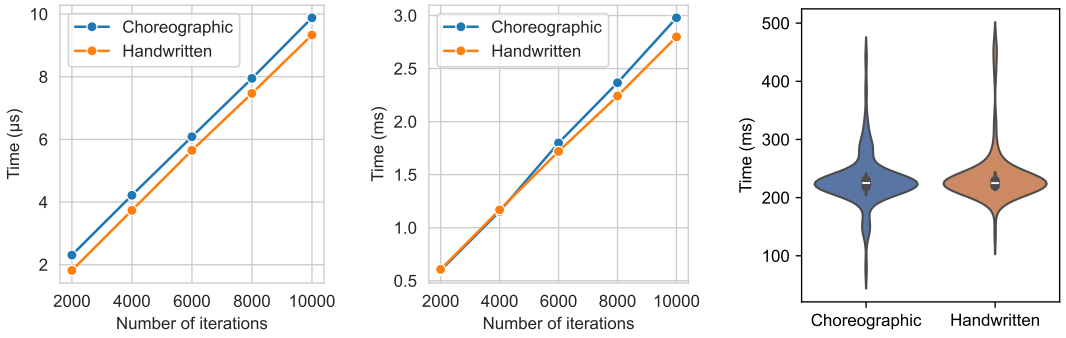
7.3 Performance

To employ CP in production, the performance overhead of using CP must be acceptable. Performance is a particular concern for library-level CP, which involves carrying out EPP at runtime. In this section, we measure the performance overhead of using ChoRus compared to traditional distributed programming in Rust. We focus on the overhead of running a choreography with EPP-as-DI. We conducted two experiments. First, we performed microbenchmarking to measure the overhead of EPP-as-DI in isolation. Second, we measured and compared the performance of the two versions of the key-value store from Section 7.1. All experiments in this section were performed on a MacBook Pro 2020 with an Apple M1 chip, 16 GB of RAM, and macOS Sonoma 14.0.

7.3.1 Microbenchmarks

With microbenchmarking, we measured the performance overhead of using two of the choreographic operators in ChoRus: `locally` and `comm`.

To measure the overhead of the `locally` operator, we implemented a simple counter program as a handwritten Rust program and as a ChoRus choreography. The program initializes a counter and repeatedly increments it a given number of times. The ChoRus version is written as a choreography that runs only at one location and uses the `locally` operator to perform initialization and increments. We use endpoint projection to execute the choreography. We measured the runtime of the two versions



(a) locally (b) comm (c) KVS Benchmark Results

Figure 7.3: Benchmark Results

of the program with different numbers of iterations. Figure 7.3a shows the result of the microbenchmark. There is a small, constant overhead of using ChoRus. Because the overhead does not grow with the number of iterations, the overhead is likely due to the cost of endpoint projection, and there is no observable overhead of using the `locally` operator.

We also measured the performance of the `comm` operator. We implemented a simple protocol that moves data from one location to another as a handwritten Rust program and as a ChoRus choreography. In both the handwritten Rust and the ChoRus versions, to isolate the performance overhead of using EPP, we used ChoRus' `LocalTransport` message transport layer to send data between the two locations. Figure 7.3b shows the result. The message passing is dominating the running time in both versions, and the overhead of endpoint projection is not observable. The ChoRus version performed slightly worse for larger iterations, with a difference of <0.5 ms.

7.3.2 Key-Value Store Benchmark

We also benchmarked the two versions of the key-value store from Section 7.1 to measure the system-level performance overhead of using ChoRus . We generated 100 random requests of 50% `get` and 50% `put` requests. We measured the runtime of the two versions of the program. We used `HttpTransport` for communication between nodes in both versions. Figure 7.3c shows a violin plot of 100 runs of the benchmark. The median runtime of the choreographic version was 225.09 ms, while the median runtime of the handwritten version was 224.93 ms. Even though the nodes are running on the same computer, the running time is dominated by the network latency, and we did not observe significant overhead of using ChoRus.

Chapter 8

Conclusion

This thesis has presented two implementation techniques to advance the state of the art of library-level choreographic programming: *endpoint projection as dependency injection* (EPP-as-DI) and *choreographic enclaves*. EPP-as-DI is a language-agnostic technique for implementing endpoint projection at the library level, requiring only support for higher-order functions from the host language. EPP-as-DI can serve as a foundation for library-level choreographic programming in a wide variety of host languages. Choreographic enclaves are a language feature that lets the programmer execute sub-choreographies within a larger choreography only among a subset of locations. Because enclaves narrow the scope of broadcast to only their participants, they give the programmer fine-grained control over the "knowledge of choice" in a choreography, eliminating unnecessary communication while still allowing the use of the host language's conditional constructs.

We also presented ChoRus and Choreography.ts, two library-level choreographic programming implementations for Rust and TypeScript, respectively, that use EPP-as-DI and choreographic enclaves. We discussed how we implemented EPP-as-DI and choreographic enclaves in these two languages. We also evaluated the usability and performance of ChoRus compared to traditional distributed programming in Rust through two case studies and performance benchmarks.

Although ChoRus and Choreography.ts are the first steps toward a more accessible and practical choreographic programming experience, there is still much work to be done. Our evaluation shows that ChoRus can be used to implement a distributed key-value store and tik-tac-toe game, but the scale of these case studies is limited. Evaluation of library-level choreographic programming in larger systems is needed to understand the full potential and limitations of our proposed techniques. Moreover, the presented libraries do not have correctness guarantees, such as deadlock freedom, that are typically associated with choreographic programming. Future work should explore how to provide such guarantees in a library-level choreographic programming setting, both from a theoretical and practical perspective.

Bibliography

[1] Samik Basu and Tevfik Bultan. Automated choreography repair. In Perdita Stevens and Andrzej Wąsowski, editors, *Fundamental Approaches to Software Engineering*, pages 13–30, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[2] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centred programming for web services. In Rocco De Nicola, editor, *Programming Languages and Systems*, pages 2–17, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[3] Marco Carbone, Kohei Honda, and Nobuko Yoshida. Structured communication-centered programming for web services. *ACM Trans. Program. Lang. Syst.*, 34(2), June 2012.

[4] Marco Carbone and Fabrizio Montesi. 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, page 263–274, New York, NY, USA, 2013. Association for Computing Machinery.

[5] Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. On global types and multi-party sessions. In Roberto Bruni and Juergen Dingel, editors, *Formal Techniques for Distributed Systems*, pages 1–28, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[6] Luís Cruz-Filipe and Fabrizio Montesi. Now It Compiles! Certified Automatic Repair of Uncompilable Protocols. In Adam Naumowicz and René Thiemann, editors, *14th International Conference on Interactive Theorem Proving (ITP 2023)*, volume 268 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 11:1–11:19, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[7] Luís Cruz-Filipe and Fabrizio Montesi. A core model for choreographic programming. *Theoretical Computer Science*, 802:38–66, 2020.

[8] Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. Dynamic Choreographies: Theory And Implementation. *Logical Methods in Computer Science*, Volume 13, Issue 2, April 2017.

[9] Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbrielli. Aiocj: A choreographic framework for safe adaptive distributed applications. In *Software Language Engineering: 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings 7*, pages 161–170. Springer, 2014.

[10] Martin Fowler. Inversion of control containers and the dependency injection pattern, Jan 2004.

[11] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Object-oriented choreographic programming, 2020.

[12] Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. Choral: Object-oriented choreographic programming. *ACM Trans. Program. Lang. Syst.*, 46(1), jan 2024.

[13] Andrew K Hirsch and Deepak Garg. Pirouette: higher-order typed functional choreographies. *Proceedings of the ACM on Programming Languages*, 6(POPL):1–27, 2022.

[14] TJ Holowaychuk. Express. `https://expressjs.com`, 2010. Accessed: 2024-05-23.

[15] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 273–284, New York, NY, USA, 2008. Association for Computing Machinery.

[16] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196–es, dec 1996.

[17] Oleg Kiselyov and Hiromi Ishii. Freer monads, more extensible effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, page 94–105, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Amending choreographies. In António Ravara and Josep Silva, editors, *Proceedings 9th International Workshop on Automated Specification and Verification of Web Systems, WWV 2013, Florence, Italy, 6th June 2013*, volume 123 of *EPTCS*, pages 34–48, 2013.

[19] Fabrizio Montesi. *Choreographic Programming.* Ph.D. thesis, IT University of Copenhagen, 2013. `https://www.fabriziomontesi.com/files/choreographic-programming.pdf`.

[20] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. Towards the theoretical foundation of choreography. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, page 973–982, New York, NY, USA, 2007. Association for Computing Machinery.

[21] Guillermo Rauch. Socket.io. `https://socket.io`, 2010. Accessed: 2024-05-23.

[22] Gan Shen, Shun Kashiwa, and Lindsey Kuper. HasChor: Functional Choreographic Programming for All. *Proc. ACM Program. Lang.*, 7(ICFP), August 2023.

[23] The World Wide Web Consortium. Web services choreography description language: Primer, 2006.