ChoRus: Library-Level Choreographic Programming in Rust

SHUN KASHIWA, University of California, Santa Cruz, USA
LINDSEY KUPER, University of California, Santa Cruz, USA

In the *choreographic programming* (CP) paradigm, a single program called a *choreography* describes the global behavior of a distributed system of nodes. The choreography is then compiled to a collection of node-local programs via *endpoint projection*. The recent development of *library-level* choreographic programming, in which choreographies and the projection mechanism are expressed as programs in an existing host language, has the potential to improve the accessibility and practicality of CP by bringing it to mainstream languages.

In this paper, we present *ChoRus*, a new library-level CP framework for the Rust programming language. ChoRus implements endpoint projection by dynamically injecting customized implementations of choreographic operators depending on the target node. We give a tour of the ChoRus API, discuss the ChoRus approach to endpoint projection, and outline future work for ChoRus and library-level choreographic programming in general.

1 INTRODUCTION

In a distributed system, the *local* behaviors of independent nodes — sending and receiving messages, and taking internal actions — must together amount to the desired *global* behavior of the entire system. But it is difficult to reason about implicit global behavior while writing only local code. The paradigm of *choreographic programming* (CP) [Carbone and Montesi 2013; Montesi 2013; Cruz-Filipe and Montesi 2020; Giallorenzo et al. 2020; Hirsch and Garg 2022; Shen et al. 2023; Montesi 2023] offers an enticing way forward by making the desired global behavior of the system *explicit*. In CP, instead of programming each node individually, the programmer writes a *choreography*: a coordination plan that defines how all the participants work together in terms of *choreographic operators*. Choreographies may be transformed into collections of executable node-local programs via a compilation step called *endpoint projection* (EPP) [Qiu et al. 2007; Carbone et al. 2007, 2012].

In the last decade, several CP languages have been proposed [Carbone and Montesi 2013; Montesi 2013; Dalla Preda et al. 2014, 2017; Giallorenzo et al. 2020; Hirsch and Garg 2022]. 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. In contrast to standalone CP languages, library-level CP has the potential to immensely 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. Library-level CP could also aid the integration of choreographies into larger systems, without any need for the programmer to change languages just to implement certain components choreographically.

To our knowledge, the first implementation of library-level CP is the recently proposed HasChor framework [Shen et al. 2023], which implements support for CP by means of a domain-specific language embedded in Haskell. In HasChor, choreographies are monadic computations in which choreographic operators may be used, and EPP is accomplished by means of dynamic interpretation of *freer monads* [Kiselyov and Ishii 2015]. The HasChor framework represents the current state of the art of library-level CP, but its implementation approach is quite Haskell-specific and not straightforwardly 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?

Authors' addresses: Shun Kashiwa, University of California, Santa Cruz, USA, Santa Cruz, shkashiw@ucsc.edu; Lindsey Kuper, University of California, Santa Cruz, USA, Santa Cruz, lkuper@ucsc.edu.

A central concept of choreographic programming is that a choreography exhibits different behaviors depending on the location to which it is projected. We observe that one way to accomplish this *location-aware* behavior in library-level CP is to represent 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 simple idea, which we call *endpoint projection as dependency injection*, or EPP-as-DI, can serve as an implementation strategy for library-level CP in any host language that supports higher-order functions.

In this paper, we present ChoRus, a new library-level CP framework for the Rust programming language, implemented using the EPP-as-DI technique. We give a tour of choreographic programming with ChoRus (Section 2) and demonstrate how ChoRus implements endpoint projection using the EPP-as-DI technique (Section 3). We conclude in Section 4 with a discussion of next steps.

2 A BRIEF TOUR OF CHORUS

In this section, we describe some essential features of the user-facing API of ChoRus. We will use the bookseller protocol [Carbone et al. 2007, 2012; Honda et al. 2008; World Wide Web Consortium 2006] as a running example. In this protocol, a buyer sends a title to a seller, who looks up the price and sends it back to the buyer. The buyer then decides whether to buy the book based on the price, and the seller provides the delivery date if the buyer decides to buy. The complete code for the bookseller protocol is available in the ChoRus GitHub repository. For a detailed discussion of the complete API, we encourage readers to consult the official ChoRus documentation.[1]

2.1 Locations

ChoRus represents each participant in a choreography as a distinct type. Users define these types by creating a new struct for each participant and deriving the `ChoreographyLocation` trait. For example, the following code defines two participant types, `Buyer` and `Seller`:

```
#[derive(ChoreographyLocation)]
struct Buyer;
#[derive(ChoreographyLocation)]
struct Seller;
```

In ChoRus, each choreography is associated with a fixed set of locations. `LocationSet` is a special type that represents a set of locations. For example, the following code defines a `LocationSet` with two locations defined above:

```
type L = LocationSet!(Seller, Buyer);
```

Rust does not have a built-in set type, so ChoRus internally uses a heterogeneous list to represent a set of locations. Because writing a list of locations by hand is verbose, `LocationSet!` is a macro to concisely define a set of locations.

2.2 Located Values

ChoRus also provides the `Located` type to represent a value located at a specific location. `Located<V, L>` represents a value of type `V` that is located at a set of locations `L`. Located values can only be constructed or deconstructed using the choreographic operators provided by the `ChoreoOp` trait, which we will discuss in the next section. This restriction ensures that located values can only be accessed at the correct locations.

[1] Link removed for anonymity; please refer to the submitted supplementary material.

2.3 Choreographies

To define a choreography, users implement the `Choreography` trait. To do so, users must provide an associated type `L` that represents the set of locations involved in the choreography, and a method `run` in which users write the choreography. The `run` method takes a reference to an object that implements the `ChoreoOp` trait, which provides the operators for writing the choreography. The following code creates a choreography.

```
struct BooksellerChoreography;
impl Choreography for BooksellerChoreography {
    type L = LocationSet!(Seller, Buyer);
    fn run(self, op: &impl ChoreoOp<Self::L>) {
        // ...
    }
}
```

2.4 Choreographic Operators

ChoRus users write a choreography using the operators provided by the `ChoreoOp` trait, which we describe in this section. The code here is simplified for brevity; in the actual implementation, the operators have additional type constraints to ensure that the operators only use locations that are part of the choreography.

2.4.1 locally. The `locally` operator is used to perform a computation at a specific location and has the following signature:

```
fn locally<V, L1: ChoreographyLocation>(
    &self,
    location: L1,
    computation: impl Fn(Unwrapper<L1>) -> V,
) -> Located<V, L1>;
```

`locally` takes a location and a computation, and returns the result of the computation located at the given location. For example, the following code reads a title from the standard input at the buyer, and stores the result in a located value `title_at_buyer`:

```
let title_at_buyer: Located<String, Buyer> = op.locally(Buyer, |_| {
    let mut title = String::new();
    io::stdin().read_line(&mut title).unwrap();
    title
});
```

Local computations can access the located values at the same location using the `Unwrapper` struct provided as an argument. Using its `unwrap` method, users can access the located values. For example, the buyer can print the title as follows:

```
op.locally(Buyer, |un| {
    println!("Title: {}", un.unwrap(&title_at_buyer));
});
```

`Unwrap` is parameterized by the location, and if the user tries to access a located value at a different location, the Rust compiler will raise a type error.

2.4.2 comm. The `comm` operator is used to move a located value from one location to another. It takes a sender location, a receiver location, and a value located at the sender, and returns the value located at the receiver. The signature of `comm` is as follows:

```
fn comm<L1: ChoreographyLocation, L2: ChoreographyLocation, V>(
    &self,
    sender: L1,
    receiver: L2,
```

```
    data: &Located<V, L1>,
) -> Located<V, L2>;
```

In the bookseller protocol, we use `comm` to express data movement between the buyer and the seller. First, we move the title from the buyer to the seller. The seller then accesses the title, looks up the price, and moves the price back to the buyer.

```
let title_at_seller: Located<String, Seller> = op.comm(Buyer, Seller, &title_at_buyer);
let price_at_seller: Located<Option<i32>, Seller> = op.locally(Seller, |un| {
    let title = un.unwrap(&title_at_seller);
    get_book(&title).map(|entry| entry.price)
});
let price_at_buyer: Located<Option<i32>, Buyer> = op.comm(Seller, Buyer, &price_at_seller);
```

`get_book` is a function that looks up the price of a book given its title. The function returns an `Option` of book information, and we use `map` to extract the price.

2.4.3 broadcast. One of the challenges in CP is the *knowledge of choice* problem [Castagna et al. 2011]. A CP language must ensure that choreographies propagate knowledge of the outcome of evaluating a conditional expression to all locations that are affected by the choice. In ChoRus, programmers can use the `broadcast` operator to handle propagation of knowledge of choice. The `broadcast` operator is similar to `comm`, but it sends a located value to *all* locations involved in the choreography. The signature of `broadcast` is as follows:

```
fn broadcast<L1: ChoreographyLocation, V>(
    &self,
    sender: L1,
    data: Located<V, L1>,
) -> V;
```

Notice that `broadcast` returns the value itself, not a located value. This is because the value is sent to all locations, so the value is no longer associated with a specific location. `broadcast` allows the programmer to access a located value at the choreography level.

Continuing our previous example, given a price, the buyer can decide whether to buy the book:

```
let decision_at_buyer: Located<bool, Buyer> = op.locally(Buyer, |un| {
    un.unwrap(&price_at_buyer).map(|price| price < BUDGET).unwrap_or(false)
});
```

This decision affects the control flow of the choreography, and both the buyer and the seller must know the decision. We use `broadcast` to send the decision to them both. Using the broadcasted decision, we can use any of Rust's control flow constructs to express the choreography. In the following, we use `if` to perform different actions depending on the buyer's decision.

```
let decision: bool = op.broadcast(Buyer, decision_at_buyer);
if decision {
    let delivery_date_at_seller = op.locally(Seller, |un| {
        get_book(&un.unwrap(&title_at_seller)).unwrap().delivery_date
    });
    let delivery_date_at_buyer = op.comm(Seller, Buyer, &delivery_date_at_seller);
    op.locally(Buyer, |un| {
        println!("The book will be delivered on {}", un.unwrap(&delivery_date_at_buyer));
    });
}
```

2.4.4 enclave. The `broadcast` operator is a heavy-handed solution to the knowledge-of-choice problem: it sends the value to *all* locations, which may be overkill. For instance, suppose that the seller needs to communicate with a third location — a publisher — to look up the latest MSRP of a book before the seller can sell a book. The publisher must be included in the choreography, so

that the seller can communicate with the publisher. However, the buyer's decision to buy the book does not affect the publisher, so the publisher does not need to know the buyer's decision. If we naively use `broadcast` to send the buyer's decision to the seller, the publisher will also receive the buyer's decision, incurring unnecessary communication.

To ameliorate this problem, Chorus provides an `enclave` operator that allows the programmer to define a *sub-choreography* that involves a subset of locations. `broadcasts` inside sub-choreographies are only sent to the locations involved in the sub-choreography, eliminating unnecessary communications. With `enclave`, we can define a sub-choreography that only involves the buyer and the seller. In the sub-choreography, the buyer can broadcast the decision and the two participants can continue based on the decision. The publisher is not involved in the sub-choreography, so it does not receive the broadcasted decision.

2.5 Transport and Endpoint Projection

To execute a choreography, users must perform endpoint projection to transform the choreography into a node-local program. ChoRus is designed to be agnostic to the specific message-passing mechanism and provides a `Transport` trait to abstract the transport layer. The library is equipped with two built-in implementations of `Transport`: `LocalTransport` for inter-thread communication, and `HttpTransport` for HTTP communication. Users can also implement their own transport layers by implementing the `Transport` trait.

Endpoint projection is performed with the `Projector` struct. Users define a `Projector` with a projection target and a transport and call the `epp_and_run` method to run the choreography at the specified location.

The following code snippet shows how to run a ChoRus choreography using the HTTP transport at the buyer's location. We construct a `HttpTransport` with the buyer's and seller's addresses and create a `Projector` with the buyer's location and the transport. We then call the `epp_and_run` method to run the choreography.

```
let config = HttpTransportConfigBuilder::for_target(Buyer, ("0.0.0.0", 8080))
    .with(Seller, ("seller.example.com", 8081))
    .build();
let transport = HttpTransport::new(config);
let projector = Projector::new(Buyer, transport);
projector.epp_and_run(BooksellerChoreography);
```

3 ENDPOINT PROJECTION AS DEPENDENCY INJECTION

When a choreography is projected, the behavior of the choreographic operators presented in Section 2.4 changes depending on the projection target. For example, the `locally` operator only performs a computation if the current projection target matches the location specified as an operand, and becomes noop otherwise. Similarly, the `comm` operator sends a value if the projection target is the sender, and receives a value if the projection target is the receiver.

ChoRus implements this location-aware behavior using *dependency injection* (DI) [Fowler 2004]. DI is a design pattern that customizes the behavior of a component by providing its dependencies as arguments. In the context of endpoint projection, we use DI to change the behavior of the choreographic operators based on the projection target, an approach we call *EPP-as-DI*.

The `run` method of the `Choreography` trait takes a reference to an object that implements the `ChoreoOp` trait. Since it is a trait, `ChoreoOp` is not tied to a specific implementation. This lets us dynamically construct different implementations of `ChoreoOp` that provide different behaviors for the choreographic operators given a projection target.

```
impl<...> Projector<...> {
    pub fn epp_and_run<...>(&'a self, choreo: C) -> V {
        // define implementations of choreographic operators for the projection target
        struct EppOp<...> {...}
        impl<...> ChoreoOp for EppOp<...>
        {
            fn locally<V, L: ChoreographyLocation>(
                &self,
                location: L,
                computation: impl Fn(Unwrapper<L>) -> V,
            ) -> Located<V, L> {
                if L::name() == Target::name() {
                    Located::local(computation(Unwrapper::new()))
                } else {
                    // returns a placeholder value that cannot be accessed
                    Located::remote()
                }
            }
            // comm, and other operators
        }
        // run the choreography by injecting the operators
        choreo.run(&EppOp {...})
    }
}
```

Fig. 1. The `epp_and_run` method of the `Projector` struct.

Figure 1 shows an excerpt of the `epp_and_run` method of the `Projector` struct. When the function is called with a choreography, it dynamically constructs a new implementation of the `ChoreoOp` trait called `EppOp` for the projection target associated with the `Projector` instance. The `locally` operator of `EppOp` checks if the location of the computation matches the projection target. If it does, the computation is performed locally and the result is returned as a located value. Otherwise, the computation is not performed and a remote located value is returned as a placeholder. The other choreographic operators are implemented similarly.

EPP-as-DI is a general technique and is not specific to Rust. ChoRus bundles the choreographic operators into a trait and passes it to the `run` method for an ergonomic API in Rust; however, the essence of the technique is that we are passing functions to another function. Hence, the technique is usable in any language that supports higher-order functions.

4 CONCLUSION AND FUTURE WORK

In this paper, we presented ChoRus, a Rust library for choreographic programming. ChoRus provides a set of abstractions for writing choreographies as Rust programs and implements endpoint projection using the EPP-as-DI technique. As a next step, we plan to conduct a thorough evaluation of ChoRus. We have implemented a few choreographic applications, such as a replicated key-value store and a multiplayer board game. We plan to evaluate the expressiveness and performance of ChoRus by implementing more complex applications and comparing them with non-choreographic Rust implementations. We will also experiment with using ChoRus alongside existing widely-used Rust libraries to evaluate its usability in real-world applications. Second, the current version of ChoRus does not provide any safety or correctness guarantees. In particular, shared mutable references can lead choreographies to deadlock. We plan to investigate how to

provide safety guarantees for choreographies at the library level, both from theoretical and practical perspectives. Finally, we plan to use the EPP-as-DI technique to implement library-level CP frameworks for more languages, and investigate how different host language features can affect the design of CP libraries.

REFERENCES

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2007. Structured Communication-Centred Programming for Web Services. In *Programming Languages and Systems*, Rocco De Nicola (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–17.

Marco Carbone, Kohei Honda, and Nobuko Yoshida. 2012. Structured Communication-Centered Programming for Web Services. *ACM Trans. Program. Lang. Syst.* 34, 2, Article 8 (June 2012), 78 pages. https://doi.org/10.1145/2220365.2220367

Marco Carbone and Fabrizio 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* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/2429069.2429101

Giuseppe Castagna, Mariangiola Dezani-Ciancaglini, and Luca Padovani. 2011. On Global Types and Multi-party Sessions. In *Formal Techniques for Distributed Systems*, Roberto Bruni and Juergen Dingel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–28.

Luís Cruz-Filipe and Fabrizio Montesi. 2020. A core model for choreographic programming. *Theoretical Computer Science* 802 (2020), 38–66. https://doi.org/10.1016/j.tcs.2019.07.005

Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. 2017. Dynamic Choreographies: Theory And Implementation. *Logical Methods in Computer Science* Volume 13, Issue 2 (April 2017). https://doi.org/10.23638/LMCS-13(2:1)2017

Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbrielli. 2014. 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*. Springer, 161–170.

Martin Fowler. 2004. Inversion of control containers and the dependency injection pattern. https://martinfowler.com/articles/injection.html

Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2020. Object-Oriented Choreographic Programming. https://doi.org/10.48550/ARXIV.2005.09520

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

Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2008. Multiparty Asynchronous Session Types. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '08)*. Association for Computing Machinery, New York, NY, USA, 273–284. https://doi.org/10.1145/1328438.1328472

Oleg Kiselyov and Hiromi Ishii. 2015. Freer Monads, More Extensible Effects. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell* (Vancouver, BC, Canada) *(Haskell '15)*. Association for Computing Machinery, New York, NY, USA, 94–105. https://doi.org/10.1145/2804302.2804319

Fabrizio Montesi. 2013. *Choreographic Programming*. Ph.D. Thesis. IT University of Copenhagen. https://www.fabriziomontesi.com/files/choreographic-programming.pdf.

Fabrizio Montesi. 2023. *Introduction to Choreographies*. Cambridge University Press.

Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. 2007. Towards the Theoretical Foundation of Choreography. In *Proceedings of the 16th International Conference on World Wide Web* (Banff, Alberta, Canada) *(WWW '07)*. Association for Computing Machinery, New York, NY, USA, 973–982. https://doi.org/10.1145/1242572.1242704

Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All. *Proc. ACM Program. Lang.* 7, ICFP (Aug. 2023). https://doi.org/10.1145/3607849

The World Wide Web Consortium. 2006. Web Services Choreography Description Language: Primer. https://www.w3.org/TR/ws-cdl-10-primer/