Choreographies as Macros

by

Alexander Bohosian

August 31, 2025

A thesis submitted to the
faculty of the Graduate School of
the University at Buffalo, The State University of New York
in partial fulfillment of the requirements for the
degree of

Master of Science
Department of Computer Science and Engineering

1 Abstract

Concurrent programming often involves writing separate programs for each participant in the system. However, this entails meticulous matching of sends and receives between participants and makes reasoning about the order of communications difficult. Choreographic programming takes a different approach, where the system is implemented by a single program that directly describes the pattern of communications between participants, ensuring deadlock freedom by construction.

Since choreographies are an actively evolving research area, it is desirable be able to rapidly prototype new ideas and features. Yet developing a new choreographic language from the ground to do this would be time consuming. Instead, a new choreographic language could be implemented as a library, allowing reuse of the host language's features for greater functionality and correctness. To test this approach, Choret, a library for functional choreographic programming is implemented.

Traditionally, choreographic semantics are described via projection and merging. Since projection and merging are operations on the syntax of programs, a host language needs support for manipulating syntax. LISP macros immediately come to mind for this task; many LISPs support procedural macros which permit arbitrary operations on syntax. Racket, a descendant of LISP, has particular features in its macro system that make it a good choice for implementing Choret.

Table of Contents

2 Introduction – Choreographies as Macros

Traditionally, to implement a system with multiple participants working concurrently, a programmer must write separate programs for each participant. However, since communication is coordinated manually by the programmer, it is all too easy to forget a send or receive, or to encode a pattern of communications that leads to deadlock.

Choreographic programming offers a different approach. A single program, called a *choreography*, describes the behavior of the system as a whole, explicitly stating where data is located and the order of communications among participants [21]. Choreographic programs are not only easier to understand, but are also provably deadlock-free by construction [14, 18, 21].

This work showcases *Choret*, a language for writing choreographic programs, which is implemented entirely as a library in the Racket programming language. Implementing a choreographic language as a library allows for significant reuse of the host language's features, saving time and constructing in a more stable implementation. This is a major benefit since choreographic programming is an evolving research area and being able to rapidly prototype new choreographic languages is desirable.

However, implementing choreographic language as a library is not as straightforward as it sounds, due to issues of syntax. In the choreographic literature, the transformation of choreographies into local programs is typically in terms of *select-and-merge endpoint projection*, which describes transformations in terms of the syntax of programs [18, 20, 21]. This is a hurdle since it demands a language with sufficiently expressive and flexible tools for manipulating the syntactic terms of the host language.

To address the issue of syntax, we turn to LISP, a family of languages that are well known for their use of macros, which allow arbitrary functions to be written that can manipulate the syntax of expressions [8, 11, 12]. We chose Racket, a descendant of LISP, thanks to particular features of its macro and library systems, which made it convenient to implement Choret.

The rest of this thesis examines the design and implementation details of Choret. Section 3 is an overview of choreographic programming using Choret. Section 4 goes in-depth on the syntax and semantics of Choret. Section 5 gives details about how Choret was implemented on top of Racket's macro system. Section 6 discusses future work in relation to parallelism and typing. Section 7 mentions related work and Section 8 concludes.

3 Choreographies and Choret

Concurrent systems are traditionally implemented by writing separate programs for each participant, which is tedious and error-prone. Choreographic programming's approach is different. Choreographic programs are a "global view" of the system that gets split into separate programs for each participant.

Take, for example, an online bookseller, with a buyer who wants to buy a book from a seller. The seller has a catalog of books and prices for those books, and the buyer has a book they want to buy and a budget they must abide by. First, the buyer sends a book title to the seller. The seller then looks up the book in the catalog, gets its price, and sends it back to the buyer. The buyer then determines if the cost of the book is within their budget; if it is, the buyer purchases the book, otherwise the order is canceled. Figure 1 shows this

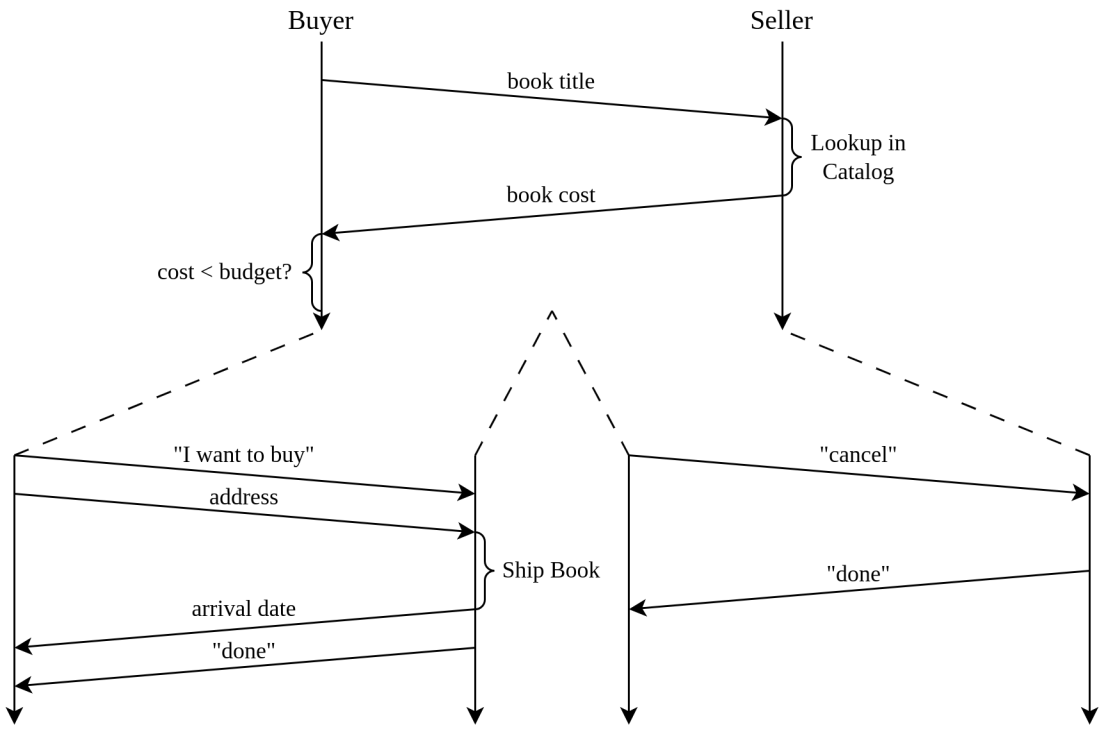

Figure 1: Diagram of the bookseller example

transaction in detail.

Note how in either case the buyer must let the seller know which choice was made. In the choreographic literature, this is known as *knowledge of choice* [21], and ensuring it is communicated correctly is an important issue that choreographies must contend with.

Traditionally, to implement the bookseller example, separate programs for the buyer and seller would be written, as shown in Figure 2.

Writing separate programs makes it difficult to reason about the correctness of the system. Notice how sends and receives must be carefully matched up (annotated by subscripts) to avoid introducing deadlocks. It is also difficult to reason about the ordering of communications in the system. Choreographies instead exploit the notion of a global view of the system, as in Figure 1, by encoding the system's pattern of communications as a single

```
;; Code at Buyer
(send_1 ch title)
(recv_2 ch cost)
(if (<= cost budget)
  (block
    (send_3 ch "buy")
    (send_4a ch address)
    (define date (recv_5a ch)))
  (block
    (send_3 ch "nevermind")
    (define response (recv_4b ch)
   )))
```

```
;; Code at Seller
(define title (recv_1 ch))
(send_2 ch (catalog title))
(define response (recv_3 ch))
(if (eq? response "buy")
  (block
    (define address (recv_4a ch))
    (send_5a ch (ship title
     address))
  (block
    (send_4b ch "goodbye")))
```

Figure 2: Traditional implementation of bookseller in Racket

```
(chor (S B)
  (define/<~_1 (at S title) (at B title))
  (define/<~_2 (at B cost) (at S (catalog title)))
  (if (at B (<= cost budget))
    (sel~>_3 B ([S 'buy])
      (let ()
        (define/<~_4a (at S address) (at B address))
        (define/<~_5a (at B date) (at S (ship title address)))))
    (sel~>_3 B ([S 'do-not-buy])
      (define/<~_4b (at B response) (at S "goodbye")))))
```

Figure 3: Implementation of bookseller in Choret

program.

The bookseller example from Figure 2 can be rewritten in Choret, as seen in Figure 3. The `chor` expression at the top acts as the entry point to Choret. It declares the names of the participants that are in the choreography, seller `S` and buyer `B`, and its body is where Choret programs are written. The next two lines define variables that are located at specific participants. For example, the expression `(define/<~ (at S title) (at B title))` sends the value of `title` evaluated at the buyer and binds it to a new variable `title` located

at the seller. Next, the `if` statement allows a single participant, `B`, to decide one of two paths to take depending on whether the cost of the book is within the buyer's budget. However, since the choice is made by just one participant, the other participants do not have any information about which branch needs to be taken. Thus, in each branch, there is a `sel~>` expression that manually specifies how knowledge of choice is to be communicated. Looking at the `'buy` branch, `(sel~> B ([S 'buy]) E)`, a `'buy` label is sent to participants `S` from participant `B`, then the choreography continues as choreographic expression `E`. The selections are important since they inform the seller of how to interact with the buyer going forward.

Writing the bookseller system as a choreography, rather than as separate programs, makes it much easier to understand and reason about the pattern of communications in the system. And, thanks to the use of selections to keep track of knowledge of choice, choreographic programs are deadlock-free by construction [14, 18, 21].

3.1 Higher-Order Choreographies

The choreographic constructs of Choret are inspired by Pirouette, a functional, higher-order, choreographic programming language [18]. This means that, in addition to the aforementioned constructs, like conditional and communication forms, Choret also has *choreographic functions*. Like normal functions, choreographic functions are a form of abstraction that allow a piece of choreographic code to be invoked later [18, 21].

For example, in the following "Ping-Pong" program in Figure 4, the participants `Ping` and `Pong` endlessly pass an ever-increasing integer back and forth to each other. To do this, there are two choreographic functions, `ping` and `pong`, that each take a local integer as an

argument, increment the integer, send that value to the other participant, and then calls the other choreographic function:

```
(chor (Ping Pong)
  (define (ping (at Ping x))
    (let ([(at Pong y) (~> (at Ping (add1 x)) Pong)])
      (pong (at Pong y))))
  (define (pong (at Pong x))
    (let ([(at Ping y) (~> (at Pong (add1 x)) Ping)])
      (ping (at Ping y))))

  ;; Start ping-pong
  (ping (at Ping 0)))
```

Figure 4: Ping-Pong Example in Choret

Additionally, the choreographic functions of Choret are higher-order. Like normal higher-order functions, choreographic functions can be passed to other choreographic functions as parameters and return choreographic functions [18].

For example, the program in Figure 5 defines the function `bob-delegates` which takes, as a parameter, another choreographic function, `F`, which delegates doing some computation to Alice. Additionally, Figure 5 defines two choreographic functions, `alice-add` and `alice-sub`, each has Alice receive two numbers, perform a calculation with those numbers, and send the result back to Bob; either of these functions may be passed as a parameter to the `bob-delegates` function.

3.2 Extensible Choreographies via Macros

In addition to higher-order functions, Choret also allows abstractions to be created using plain Racket macros. Choret leans heavily on Racket's macro expander to compile

```
(chor (Alice Bob)
  (define (bob-delegates F)
    (let ([(at Bob x) (F (at Alice 7) (at Alice 3))])
      (at Bob (printf "Bob got ~a" x))))

  (define (alice-add (at Alice x) (at Alice y))
    (~> (at Alice (+ x y)) Bob))
  (define (alice-sub (at Alice x) (at Alice x))
    (~> (at Alice (- x y)) Bob))

  ; Call bob-receives with higher-order parameter
  (bob-delegates alice-add) ; Prints "Bob got 10"
  (bob-delegates alice-sub) ; Prints "Bob got 4"
  )
```

Figure 5: Higher-Order Choreography

Choret programs into separate programs for each participant. As a consequence, Racket macros can be used directly in the language of Choret.

```
(define-syntax-rule (gather~> [(at P E) ...] Q)
  (list (~> (at P E) Q) ...))

(chor (A B C)
  (define (at A x) (at A 10))
  (define (at B x) (at B 10))
  (define (at C lst) (at C (gather~> [(at A x) (at B y)])))
  (define (at C sum) (at C (foldl + 0 lst)))
```

Figure 6: Using macros in Choret

For example, in Figure 6, a new macro of the form `(gather~> [(at P E) ...] Q)` can be defined which gathers values from multiple participants and places them in a list on participant `Q`. Racket's `define-syntax-rule` form is used to introduce a new pattern macro `gather~>`, which includes a pattern for how to apply the macro: `(gather~> [(at P E) ...] Q)`, and what the macro should expand into: `(list (~>(at P E) Q) ...)`. Then in

the Choret program below, the `gather~>` macro is used to get the values from participants `A` and `B`, and collect them in a list on participant `C`.

Defining new macros, like `gather~>` from above, can reduce boilerplate code and allows new compile time abstractions to be added to Choret.

4 Syntax and Semantics

Choret's syntax and semantics are inspired by Pirouette, the first functional choreographic language with formalized semantics [18]. One of its attractive design features is its distinction between expressions of the choreographic language and expressions of a *local language*. Pirouette does not prescribe any particular local language and instead treats a language generically; as long as the local language obeys a certain set of requirements it may be used as a local language for Pirouette [18]. Following the design of Pirouette, Choret's local language is Racket. As for Choret's choreographic syntax, by using Racket macros Choret is an extension of Racket's normal S-expression syntax, with a few changes to the structure of some of Racket's built-in forms.

4.1 Choreographic Syntax

Writing a choreographic program begins with using the `chor` macro, which takes a list of participant names that are to participate in the choreography, as shown in Figure 7.

Inside the body of a `chor` form, Choret syntax may be used. For example, the `~>` macro, which mentions the participant names P and Q, and says "evaluate local expression e at the participant P and send the result to participant Q".

Racket Expressions	e		
Binding Forms	B	::=	X | `(at` P x`)`
Choret Programs	P	::=	`(chor (`P …`)` T …`)`
Choret Expressions	E	::=	`(at` P e …`)` | `(~>` `(at` P e`)` Q`)`
		|	`(if (at` P e`)` E_1 E_2`)` | `(sel~>` P `([`Q l`]` …`)` E`)`
		|	`(let ([`B E_1`]` …`)` E`)` | `(let*` `([`B E_1`]` …`)` E`)`
		|	`(set! (at` P x`)` E`)` |
Choret Terms	T	::=	`(define` B E`)` | `(define/<~ (at` P x`)` `(at` Q e`))`

Figure 7: Choret Syntax

The `(at` P e `...)` form is used to evaluate the expressions e `...` at participant P. Like Racket's `begin` form, `at` is a splicing form, so multiple definitions may be placed in e `...` and they will be "spliced" (made available) to the surrounding internal-definition context ("splicing" and internal-definition contexts are Racket-specific terms, see the documentation on internal definitions for more details [5]).

Binding expressions such as `let` and `define` are "overloaded" so they may be used in both choreographic and local expressions. When used in choreographic expressions, `let` and `define` can introduce two kinds of bindings: local bindings and global bindings. For example, in a local binding of the form `(define (at P x) (at P 5)`, `x` is a local variable for participant P and can only be referenced from a local expression at participant `P`. In a global binding of the form `(define X (at P 5)`, `X` is a global variable and can be used directly as a choreographic expression. Note that a global variable may still hold a value that is located at a particular participant; due to Choret's lack of a type system the programmer is responsible for ensuring that the against any runtime errors that might arise from a value in global variable being at the wrong participant. Global variables can also be bound to choreographic functions.

4.2 Semantics — Endpoint Projection (EPP) and Merging

There are different ways to implement a choreographic program. We treat each participant as an independent and concurrent thread of execution, each with its own separate state (e.g. local variables) and a set of communication primitives for sending and receiving messages synchronously with other participants. To achieve this, we produce a separate body of code for each participant (which we will refer to as a *program*). The collection of participant programs also need to be *compliant* with the choreography, that is, the behavior of each program is in accordance with the what the choreography says it should do [21]. The process of generating a program for a participant is known as *projection* and the generated program is known as a *projection* [21]. Performing projection for all the participants of a choreographic program is known as *EndPoint Projection* (EPP) and the collection of projections from performing EPP, is known as a *network* and implements the behavior of the choreographic program [21]. The projection of a choreographic expression E of process `P` has the notation $[\![E]\!]_{\texttt{P}}$ [20, 21].

For many choreographic forms, projection is relatively straightforward. For example, for the form $E =$ `(~> (at A 5) B)` has the projections `(send B 5)` for A, `(recv A)` for B, and `(void)` otherwise. However, conditional expressions, like the `if` form, present a problem due to knowledge of choice. Consider the Choret programs in Figure 9 where participant `A` uses `x` to decide which branch to take.

$$
[\![\texttt{E}]\!]_{\texttt{A}} = \begin{cases}
e\ \ldots & \text{if } E = \texttt{(at } A\ e\ \ldots\texttt{)} \\
\texttt{(void)} & \text{if } E = \texttt{(at } P\ e\ \ldots\texttt{)} \text{ where } P \neq A \\
\texttt{(send } Q\ e\texttt{)} & \text{if } E = \texttt{(\~{}> (at } A\ e\texttt{) } Q\texttt{)} \\
\texttt{(recv } P\texttt{)} & \text{if } E = \texttt{(\~{}> (at } P\ e\texttt{) } A\texttt{)} \\
\texttt{(void)} & \text{if } E = \texttt{(\~{}> (at } P\ e\texttt{) } Q\texttt{)} \text{ where } P \neq A \text{ and } Q \neq A \\
\texttt{(if } e\ [\![E_1]\!]_{\texttt{A}}\ [\![E_2]\!]_{\texttt{A}}\texttt{)} & \text{if } E = \texttt{(if (at } A\ e\texttt{) } E_1\ E_2\texttt{)} \\
[\![E_1]\!]_{\texttt{A}} \sqcup [\![E_2]\!]_{\texttt{A}} & \text{if } E = \texttt{(if (at } P\ e\texttt{) } E_1\ E_2\texttt{)} \text{ where } P \neq A \\
\texttt{(let ([}X\ [\![E_1]\!]_{\texttt{A}}\texttt{] \ldots) } [\![E]\!]_{\texttt{A}}\texttt{)} & \text{if } E = \texttt{(let ([}X\ E_1\texttt{] \ldots) } E\texttt{)} \\
\texttt{(let ([}x\ [\![E_1]\!]_{\texttt{A}}\texttt{] \ldots) } [\![E]\!]_{\texttt{A}}\texttt{)} & \text{if } E = \texttt{(let ([(at } A\ x\texttt{) } E_1\texttt{] \ldots) } E\texttt{)} \\
\texttt{(let ([\_ } [\![E_1]\!]_{\texttt{A}}\texttt{] \ldots) } [\![E]\!]_{\texttt{A}}\texttt{)} & \text{if } E = \texttt{ (let ([(at } P\ x\texttt{) } E_1\texttt{] \ldots) } E\texttt{)} \\
 & \quad \text{where } P \neq A \\
\texttt{ (choose! } Q_1\ l_1 & \text{if } E = \texttt{ (sel\~{}> } A\ \texttt{([}Q_1\ l_1\texttt{] [}Q_2\ l_2\texttt{] \ldots) } E\texttt{)} \\
\quad [\![\texttt{(sel\~{}> } A\ \texttt{([}Q_2\ l_2\texttt{] \ldots) } E\texttt{)}]\!]_{\texttt{A}}\texttt{)} & \\
\texttt{ (branch? } P & \text{if } E = \texttt{ (sel\~{}> } P\ \texttt{([}A\ l_1\texttt{] [}Q_2\ l_2\texttt{] \ldots) } E\texttt{)} \\
\quad \texttt{([}l_1\ [\![\texttt{(sel\~{}> } P\ \texttt{([}Q_2\ l_2\texttt{] \ldots) } E\texttt{)}]\!]_{\texttt{A}}\texttt{]))} & \\
[\![\texttt{(sel\~{}> } P\ \texttt{([}Q_2\ l_2\texttt{] \ldots) } E\texttt{)}]\!]_{\texttt{A}} & \text{if } E = \texttt{ (sel\~{}> } P\ \texttt{([}Q_1\ l_1\texttt{] [}Q_2\ l_2\texttt{] \ldots) } E\texttt{)} \\
 & \quad \text{where } P \neq A \text{ and } Q_1 \neq A
\end{cases}
$$

Figure 8: Definition of Endpoint Projection (Selected Parts)

```
(chor (A B)
  (define (at A x) ...)
  (if (at A x)
    (at B "Left")
    (at B "Right")))
```

```
(chor (A B)
  (define (at A x) ...)
  (if (at A x)
    (sel~> A [B 'l]
      (at B "Left"))
    (sel~> A [B 'r]
      (at B "Right"))))
```

Figure 9: Incorrect (left) and correct (right) use of Knowledge of Choice

On the left in Figure 9, it is not immediately clear how to project the `if` form for `B`, since the projection of `B` does not know what choice `A` makes. A naive solution would be to always communicate knowledge of choice from `A` to `B`. However, this is not a scalable solution when many more participants are involved, since, for n participants, $n-1$ communications would be sent even if only a small subset of the participants actually need to know of `A`'s choice. Thus, knowledge of choice is traditionally handled by the programmer

$$N_1 \sqcup N_2 = \begin{cases} \text{recursively merge} & \text{if } N_1 \text{ and } N_2 \text{ are matching Racket forms} \\ \texttt{(send } P\ e\texttt{)} & \text{if } N_1 = N_2 = \texttt{(send } P\ e\texttt{)} \\ \texttt{(recv } P\texttt{)} & \text{if } N_1 = N_2 = \texttt{(recv } P\texttt{)} \\ \texttt{(choose! } P\ l\ N_1' \sqcup N_2'\texttt{)} & \text{if } N_1 = \texttt{(choose! } P\ l\ N_1'\texttt{)} \\ & \text{and } N_2 = \texttt{(choose! } P\ l\ N_2'\texttt{)} \\ \texttt{(branch? } P\ \texttt{(} [l_{1i}\ \ N_{1i} \sqcup N_{2j}] \ldots & \text{if } N_1 = \texttt{(branch? } P\ \texttt{([}l_{11}\ \ N_{11}\texttt{] …))} \\ \qquad [l_{1k}\ \ N_{1k}] \ldots & \text{and } N_2 = \texttt{(branch? } P\ \texttt{([}l_{21}\ \ N_{21}\texttt{] …))} \\ \qquad [l_{2k}\ \ N_{2k}]\texttt{))} & \text{and } l_{1i} = l_{2j} \\ & \text{and } \forall k, k'.l_{1k} \neq l_{2k'} \\ \bot & \text{otherwise} \end{cases}$$

Figure 10: Definition of Merging

manually specifying knowledge of choice communications using selections (via the `sel~>` form in Choret) [20, 21]. To ensure that selections are used to communicate knowledge of choice where necessary, an operation known as *merging* is performed, which, in essence, requires the projected code of each branch of a conditional form to be the same unless there is a selection which provides information about which branch to take when the code differs.

For example, when an expression of the form `(if (at P x)` E_1 E_2`)` is projected for any participant other than `P`, say `Q` for example, the subexpressions E_1 and E_2 are first projected to $[\![E_1]\!]_{\texttt{Q}}$ and $[\![E_2]\!]_{\texttt{Q}}$. Then the projections are compared syntactically for differences. If there are no differences, then nothing needs to be done since participant `Q` does the same thing regardless of what `P` decides to do. On the other hand, if there are differing expressions, then it depends on the labels of each branch. If neither the expressions are `branch?` forms, then merging immediately fails. Otherwise, if both expressions are `branch?` forms it depends on the set of labels of each `branch?` form. If the labels are compatible, then merging succeeds; otherwise merging still fails.

Network Language N ::= *... (All other Racket forms)*
| `(send` P e`)` | `(recv` P`)`
| `(choose!` P l N`)` | `(branch?` P `([`l N`] …))`

Figure 11: Network Language Syntax

4.3 Network Language

The projection of each participant is a program that encodes the behavior of a participant with respect to the choreography. This work refers to the language of these projections as the *network language*, which is analogous to the control language of Pirouette [18]. In the case of Choret, the network language is simply an extension of Racket, with four new forms: `send`, `recv`, `choose!`, and `branch?`, as shown in Figure 11. These forms reflect the message-passing primitives used in Pirouette's control language [18] and those used by Montesi [21]. The `send` and `recv` forms, as the names suggest, describe how information is transmitted from one participant to another. The `choose!` and `branch?` forms also send and receive data, but are implemented as distinct forms both for simplicity and, in the case of `branch?`, for performing merging.

5 Implementation

Choret is embedded in Racket as a library. Individual Choret forms, such as `at` and `~>`, are implemented as individual macro definitions, whose resulting expansion is a projection of the form for a specific participant. The advantage of this approach is twofold. First, it enables a relatively simple and robust implementation of choreographic programming by reusing much of the language infrastructure from Racket, enabling Choret's implementation to focus

on the details of choreographic programming. Second, it enables the use of select-and-merge endpoint projection (S&M EPP), while still allowing for a library-based implementation.

On the latter point, S&M EPP can be difficult to implement at as a library since it requires manipulating syntax of both the choreographic and local languages. This means that a host language needs appropriate compile time features for inspecting and manipulating syntax. Racket's macro system provides enough power to be able to do compile-time S&M EPP entirely as a library.

Choret's major implementation details are implemented in two modules. The `threads-network` module provides abstractions for describing networks and synchronous communication that are defined in terms of Racket threads. The `main` module implements choreographic programming in terms of the network abstractions.

5.1 `threads-network` Module

The `threads-network` module provides abstractions for building a network, which are later used when projecting Choret programs. A `(define-network BODY ...)` macro acts as the entry point for building a new network, and inside the `BODY` form(s), uses of a `define-process` macro defines the participants of the network. Two macros, `send` and `recv`, perform synchronous communication and keep track of which pairs of participants communicate with each other so channels only need to be created between participants that actually communicate with each other.

Currently, the `threads-network` module is built on top of Racket threads. Racket threads are a convenient choice since they use synchronous channels to send and receive

messages. However, one drawback of using Racket threads is that they only run concurrently. That is, Racket threads can't exploit parallelism to do work faster than a single-threaded solution [1, 2]; Section 6.1 mentions alternatives to Racket threads that would allow for true parallelism.

5.2 `main` (Choreographic) Module

The `main` module defines the syntax of Choret programs and implements S&M EPP. Choret is implemented as a shallowly-embedded language in Racket, with a separate macro definition for each Choret form. Each of these macros separately defines how the Choret form should be projected. The interesting consequence of this implementation strategy is that projection is almost entirely handled by Racket's normal process of macro expansion (with the exception of merging, covered in Section 5.2.1).

Since multiple projections are needed, to generate one program for per participant the body of a choreographic program is macro-expanded multiple times, once per participant. In Racket, this may be done by using `local-expand`, which is a function that can be called by a macro to fully macro expand all uses of macros in a Racket expression [6]. Choret uses `local-expand` to expand, and thus project, the body of a Choret program multiple times.

During each expansion, Choret macros need to know which participant is being currently projected. We communicate this information via Racket syntax parameters. A syntax parameter is set by using the `syntax-parameterize` macro, which creates a new dynamic binding from an identifier to a value that remains visible to any further macro expansion that occurs in the body of the `syntax-parameterize` macro [7]. When a Choret macro

is invoked, it uses `syntax-parameter-value` to check which participant is currently being projected.

Encoding projection as macro expansion is what also allows Choret to be extensible using regular Racket macros; since the Racket macro expander is doing the heavy-lifting of projection, new forms added by `define-syntax` will automatically be recognized and expanded.

5.2.1 Knowledge of Choice and Merging

Knowledge of choice poses a unique challenge for macro expansion. S&M EPP does some internal bookkeeping using the `branch?` form, which keeps track of what labels have been sent by each participant. When merging two `branch?` forms, the labels attached to each form are checked to ensure knowledge of choice is properly communicated. However, this interferes with how projection is encoded as macro expansion. Since merging operates on projected forms, and projection is performed by macro expansion, our implementation has to perform merging on Racket core forms. This means that merging can't directly look utilize custom syntax defined via macros, since they will be turned into Racket core forms during macro expansion (i.e. projection).

The solution is to hide `branch?` forms inside of an existing core form so they remain intact for merging. We chose the `quote-syntax` form as it is a Racket core form for which Racket doesn't expand the syntax object inside it (Note we could not find explicit mention of this fact in the Racket documentation, but it appears to be implied by the grammar of fully expanded programs in the documentation [4]). For example, when expanding `(sel~> P (['l Q]) E)` for process `Q`, it expands to `(quote-syntax (branch? P ['l E]) #:local)`.

(Note that the expression `E` is also expanded using `local-expand`). Later, when merging is performed, it will specifically look for these hidden `branch?` forms.

After projection and merging has completed, there may still be hidden `branch?` forms left over. This is to be expected, but it would not be correct to leave these hidden forms as-is since they are still wrapped inside of `quote-syntax` forms. Thus we make one more pass on the projections which turns the hidden `branch?` forms into forms that use the Racket `case` macro.

6 Future Work

6.1 Parallelism

Choret currently uses Racket threads to implement concurrency and message passing among participants. One of the drawbacks of using Racket threads, however, is that they only perform concurrent, not parallel, execution [1, 2]. While Choret currently only uses Racket threads, Racket does have two other features, futures and places, that could be used to support parallelism among participants.

Unlike Racket threads, Racket futures are capable of executing a subset of Racket computations in parallel [3]. In theory, implementing participants as Racket futures could enable parallelism in Choret. However, this might be difficult to realize in practice since the evaluation of a Racket future may become halted due to blocking operations; and once a future becomes halted, it will not continue to run until the `touch` function is used to force the future to evaluate to completion [9]. Worse yet, even when `touch` is called on a halted

future, the future will only continue concurrently, not in parallel, with other futures [9]. The limitations of futures would make it difficult to create an implementation that yields meaningful performance benefits.

In contrast, Racket's places are a better candidate to use in Choret's network language since they allow for much more reliable parallelism than futures. Akin to threads and futures, places allow for concurrent execution of code. However, unlike threads and futures, a place is "...effectively a new Racket instance that can run in parallel with other places... " [10]. Since there is a significant degree of separation among Racket places (e.g. each place has its own garbage collector) [23] places can reliably run in parallel and continue to do so after blocking operations, unlike futures.

6.2 Typed Choreographies

Choret is not statically typed, with neither local nor location types. However, adding typing to choreographic expressions in Choret presents some technical challenges. We discuss a couple of potential approaches, both with their own trade-offs.

6.2.1 Embedding Choreographic Types in Typed Racket

One approach would be to embed choreographic types in Typed Racket. Typed Racket is a sister language of Racket that adds static, gradual-typing. The advantage of an embedding to Typed Racket would be that all of the work of type inference and checking would be handled by Typed Racket.

However, there is an issue with how Typed Racket is implemented: it fully expands all the macros in a module before doing type inference [24]. This means that projection,

which is performed during macro expansion, can't leverage the types of expressions when deciding how to project code, since projection happens entirely before type inference.

Take, for example, the interaction between channels in Typed Racket and projection, and the expression `(~> (at P e) Q)` whose projection for participant `Q` is `(recv P)`. The expression `(recv P)` would further expand into `(channel-get ch-P)`, where `channel-get` is a function that receives a value across the channel `ch-P`. In Typed Racket, a channel must have an associated type, but the type can't be inferred by Choret since type inference is done by Typed Racket after all other code for the module has be expanded. This would force us to give it type `Any`, to allow any type through the channel. However this is still problematic, since `(channel-get ch-P)` would have the inferred type `Any`, which would have to be cast to the type of the expression `e`, which Choret does not know. The most feasible solution would be to require that the local type is always manually specified for communication forms such as `~>` so that Choret can insert the necessary type annotations in the projected Typed Racket code.

6.2.2 Embedding Type Checking in Macro Expansion

Another approach would be to implement a from-scratch type system for Choret that performs type checking during macro expansion. There is, in fact, already a Racket library that supports such a paradigm: Turnstile, a Racket library (and DSL) that does type checking during macro expansion [15].

Like the current implementation of Choret, all the choreographic forms would be implemented as separate macros. However, unlike the Typed Racket approach, Turnstile requires that every macro has explicit rules attached to it for how to type check its subforms

and how to infer the type of the resulting expanded form.

The advantage of this approach is that projection can take advantage of the inferred types to determine how to project expressions of the choreographic language. However, one problem with using Turnstile is that all the forms of the choreographic language need to be implemented as Turnstile macros. Turnstile can't be used to infer the types of local expressions, since they are all arbitrary Racket forms. This means that using Turnstile alone can only enforce the location types, and not the local types, of expressions. Additionally, any extensions to Choret could only be made using Turnstile macros, not regular Racket macros.

6.2.3 Tradeoffs

There are two important trade-offs to consider between a Typed Racket embedding and a Turnstile-like solution. Turnstile could use the inferred location types of choreographic expressions to influence projection, whereas Typed Racket embedding seemingly wouldn't be able to. On the other hand, with a Typed Racket embedding it would be much more feasible to typecheck the local types of choreographic expressions, unlike Turnstile. The choice of which to use depends on which aspect of choreographic typing one wishes to emphasize. For local types a Typed Racket embedding would be preferable; for location types a Turnstile-like solution would be preferable.

7 Related Work

Choret's choreographic language features are derived from Pirouette [18]. Pirouette was developed to formalize the semantics of functional choreographies, that is, choreographies with higher-order choreographic functions. Also of note is Chorλ, another functional choreographic programming language [16] that was independently developed at the same time as Pirouette.

Using Pirouette as a foundation, Choret is implemented as a Racket library that uses macros to compile Choret programs into Racket programs. However, other choreographic languages take different approaches to implementing choreographic programming.

For example, one approach is to write a custom parser and compiler to transform the choreographic language to an existing language, such as how Choral compiles to Java code [17]. This saves a lot of time implementing the lower level features of the language, like handling machine/byte code generation, basic data structures, and libraries. However, it still requires considerable effort create a parser and datastructures for representing and manipulating the AST of the choreographic language.

A more direct approach is to directly embed the choreographic language into the host language. For example, HasChor is a library written in Haskell [22]. This allows for reuse the parser of the language itself. However, not all languages have metaprogramming features that make it feasible to implement traditional select-and-merge projection as a library; HasChor, in fact, has such a limitation and does projection entirely at runtime [22].

The closest choreographic language to Choret is Klor, which is implemented as an embedded language in Clojure, another descendant of LISP. Like Choret, Klor performs

compile-time projection. However, Klor does not use S&M EPP and instead uses agreement types to ensure that knowledge of choice is followed [13, 19].

8 Conclusion

Choreographies are of interest because they provide a compelling way of writing concurrent programs that is easier and safer than manually writing separate programs for each participant. However, choreographic language design is still undergoing further development and experimentation.

To this end, being able to rapidly implement new choreographic language features would help with developing and testing such features. There are multiple ways to implement new language features, but, for rapid prototyping, they would ideally be built on top of existing languages as libraries. Racket's macro and library systems make it particularly attractive for implementing choreographic programming with traditional select-and-merge endpoint projection.

Choret demonstrates the feasibility of this approach by embedding choreographic programming in Racket using macros. The resulting implementation has only 370 lines of Racket (excluding comments and tests), while also allowing Choret to easily interact with the features of Racket. Since Choret relies primarily on Racket's macro expander to perform projection, we get extensible choreographies in Choret for free, which may be leveraged to develop new, *syntactic*, abstractions for choreographies.

9 References

[1] *11.1 Threads*. – URL https://docs.racket-lang.org/reference/threads.html. – Accessed July 16th, 2025

[2] *1.1.12 Threads*. – URL https://docs.racket-lang.org/reference/eval-model.html#(part._thread-model). – Accessed July 16th, 2025

[3] *11.4 Futures*. – URL https://docs.racket-lang.org/reference/futures.html. – Accessed July 20th, 2025

[4] *1.2.3.1 Fully Expanded Programs*. – URL https://docs.racket-lang.org/reference/syntax-model.html#(part._fully-expanded). – Accessed July 30th, 2025

[5] *1.2.3.8 Internal Definitions*. – URL https://docs.racket-lang.org/reference/syntax-model.html#%28part._intdef-body%29. – Accessed July 30th, 2025

[6] *12.4 Syntax Transformers*. – URL https://docs.racket-lang.org/reference/stxtrans.html. – Accessed July 30th, 2025

[7] *12.5 Syntax Parameters*. – URL https://docs.racket-lang.org/reference/stxparam.html. – Accessed July 28th, 2025

[8] *16.2 General Macro Transformers*. – URL https://docs.racket-lang.org/guide/proc-macros.html. – Accessed July 1st, 2025

[9] *20.1 Parallelism with Futures*. – URL https://docs.racket-lang.org/guide/parallelism.html#(part._effective-futures). – Accessed July 20th, 2025

[10] *20.1 Parallelism with Places*. – URL https://docs.racket-lang.org/guide/parallelism.html#(part._effective-places). – Accessed July 21th, 2025

[11] *Clojure - Macros*. – URL https://clojure.org/reference/macros. – Accessed July 1st, 2025

[12] *Common Lisp HyperSpec: Macro DEFMACRO*. – URL https://www.lispworks.com/documentation/HyperSpec/Body/m_defmac.htm. – Accessed July 1st, 2025

[13] Bates, Mako ; Kashiwa, Shun ; Jafri, Syed ; Shen, Gan ; Kuper, Lindsey ; Near, Joseph P.: Efficient, Portable, Census-Polymorphic Choreographic Programming. In: *Proc. ACM Program. Lang.* 9 (2025), Juni, Nr. PLDI. – URL https://doi.org/10.1145/3729296

[14] Carbone, Marco ; Montesi, Fabrizio: Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming. In: Giacobazzi, Roberto (Hrsg.) ; Cousot, Radhia (Hrsg.): *POPL 2013*, ACM, 2013, S. 263–274

[15] Chang, Stephen ; Knauth, Alex ; Greenman, Ben: Type systems as macros. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. New York, NY, USA : Association for Computing Machinery, 2017 (POPL '17), S. 694–705. – URL https://doi.org/10.1145/3009837.3009886. – ISBN 9781450346603

[16] Cruz-Filipe, Luís ; Graversen, Eva ; Lugovic, Lovro ; Montesi, Fabrizio ; Peressotti, Marco: Functional Choreographic Programming. In: Seidl, Helmut (Hrsg.) ; Liu, Zhiming (Hrsg.) ; Pasareanu, Corina S. (Hrsg.): *ICTAC 2022* Bd. 13572, Springer, 2022, S. 212–237

[17] Giallorenzo, Saverio ; Montesi, Fabrizio ; Peressotti, Marco: Choral: Object-Oriented Choreographic Programming. In: *ACM Transactions on Programming Languages and Systems* 46 (2024), Nr. 1, S. 1–59

[18] Hirsch, Andrew K. ; Garg, Deepak: Pirouette: Higher-Order Typed Functional Choreographies. In: *POPL 2022* Bd. 6, 2022, S. 1–27

[19] Logović, Lovro ; Jongmans, Sung-Shik: Klor: Choreographies for the Working Clojurian. In: *Choreographic Programming (CP)*, URL https://pldi24.sigplan.org/details/cp-2024-papers/15/Klor-Choreographies-for-the-Working-Clojurian, 2024

[20] Montesi, Fabrizio: *Choreographic Programming*, IT University of Copenhagen, Dissertation, 2013. – URL https://www.fabriziomontesi.com/files/choreographic_programming.pdf

[21] Montesi, Fabrizio: *Introduction to Choreographies*. Cambridge University Press, 2022. – ISBN 9781108981491

[22] Shen, Gan ; Kashiwa, Shun ; Kuper, Lindsey: HasChor: Functional Choreographic Programming for All (Functional Pearl). In: *International Conference on Functional Programming (ICFP)*, 2023

[23] Tew, Kevin ; Swaine, James ; Flatt, Matthew ; Findler, Robert B. ; pdinda@northwestern.edu, Peter D.: Places: adding message-passing parallelism to racket. In: *Proceedings of the 7th Symposium on Dynamic Languages*. New York, NY, USA : Association for Computing Machinery, 2011 (DLS '11), S. 85–96. – URL https://doi.org/10.1145/2047849.2047860. – ISBN 9781450309394

[24] Tobin-Hochstadt, Sam ; Felleisen, Matthias: The design and implementation of typed scheme. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. New York, NY, USA : Association for Computing Machinery, 2008 (POPL '08), S. 395–406. – URL https://doi.org/10.1145/1328438.1328486. – ISBN 9781595936899