

Choreographies as Macros

Alexander Bohosian

Department of Computer Science and Engineering
University at Buffalo, SUNY
Buffalo, NY, USA
asbohosi@buffalo.edu

Andrew K. Hirsch

Department of Computer Science and Engineering
University at Buffalo, SUNY
Buffalo, NY, USA
akhirsch@buffalo.edu

Concurrent programming often entails meticulous pairing of sends and receives between participants to avoid deadlock. Choreographic programming alleviates this burden by specifying the system as a single program. However, there are more applications than implementations of choreographies, and developing new implementations takes a lot of time and effort. Our work uses Racket to expedite building a new choreographic language called Choret. Racket has a powerful macro system which allows Choret to reuse much of its infrastructure for greater functionality and correctness.

1 Introduction

Choreographic programming [see e.g., 15, 18, 20] is an emerging paradigm for designing and implementing concurrent systems. Traditional languages require writing a program for each participant, while avoiding mismatched sends and receives that may cause deadlock. This task grows more difficult with program size and the number of participants. In contrast, choreographic programs—or *choreographies*—encode the system’s pattern of communication in a single program, rather than interweaving several programs. This global view directly encodes the correct pairing of sends and receives, ensuring deadlock freedom.

Since choreographic programming is constantly evolving, rapid prototyping is often desirable. Building a compiler from scratch—even a transpiler—takes time away from higher-level language decisions. Ideally, one would reuse the existing infrastructure of a host language to provide a more robust implementation. Recently, there has been significant interest in doing so via *choreographic libraries* [16, 17, 21, 22]. However, every current choreographic library either has nontraditional semantics or uses a nontraditional language design. These nontraditional design choices allow the library to avoid issues arising from *knowledge of choice*, a technical aspect of choreographic programming. Ideally, we’d be able to implement the traditional design and semantics of choreographies in a choreographic library. To do this, a clever *scheme* is needed.

Such an implementation relies on the metaprogramming capabilities of the host language; in particular, the LISP family of languages offer a powerful metaprogramming system in the form of macros. In many LISPs macros are user-defined functions which run during *compilation*, specifically in a *macro-expansion phase*. These functions are written *in* LISP, akin to normal functions, and can perform arbitrary computation. Most importantly, macros can be imported as part of a module, allowing language extensions to be used like normal libraries.

Intuitively, a choreographic language could be implemented as a set of LISP macros, using their power to provide a choreographic library with traditional design and semantics. We prove this intuition correct by building Choret, an embedded choreographic language in Racket. Racket is a LISP (specifically, an offshoot of Scheme) which offers a particularly sophisticated macro system for metaprogramming, which is key for our implementation of knowledge of choice.

We provide the following contributions: (a) in Section 2, we describe Choret via an example and describe its syntax; (b) in Section 3 we describe the network (target) language of Choret, and describe Choret’s compile-time semantics via *Endpoint Projection*; and (c) in Section 4 we describe how Racket’s macro system allows us to implement Choret while maintaining traditional choreographic-language design and semantics. Finally, in Section 5 we survey related work, and in Section 6 we conclude.

2 Choreographies and Choret

Let us begin by considering an example concurrent application, and see how we would implement it both as a traditional Racket program and in Choret. We use a traditional example: the bookseller. Here, a Buyer wants to buy a book from a Seller. To do so, Buyer sends Seller the title of the book and then Seller looks up the title in a catalog and sends back a price. Buyer then determines whether the price is within their budget. If it is, then Buyer informs Seller of this choice and sends their address, and Seller sends Buyer a date by which to expect the book. If the book is not within Buyer’s budget, then they inform the Seller of this and the protocol ends.

Bellow, we can see a Racket program which implements this protocol in the traditional style. There is one program for each of the Buyer and the Seller. These programs share a channel *ch*, which they use to send and receive messages to and from each other. The sends and receives need to be perfectly matched, and this needs to be done by hand: the programmer must check to make sure that they are following the protocol precisely. We have made it easier to read our example by adding subscripts to the sends and receives: *send_n* matches with *recv_n*.

```
;; Code at Buyer                                ;; Code at Seller
(send1 ch title)                                (define title (recv1 ch))
(recv2 ch cost)                                (send2 ch (catalog title))
(if (<= cost budget)                            (define response (recv3 ch))
  (block                                       (if (eq? response "buy")
    (send3 ch "buy")                          (block
    (send4a ch address)                        (define address (recv4a ch))
    (define date (recv5a ch)))                (send5a ch (ship title
  (block                                       address))
    (send3 ch "nevermind")                    (block
    (define response (recv4b ch))              (send4b ch "goodbye"))))
  )))
```

Note the painstaking way that the sends and receives need to be matched. In particular, note that there are two copies of *send₃*, each matched with the same *recv₃*: because they are in different branches of an *if* expression, only one of these sends will run. However, it further complicates the job of matching these sends and receives. While this matching is easily possible to perform in this case, in larger programs it can become very difficult.

This is where Choret (and choreographic programming more broadly) comes in. Rather than writing a program for each of the participants, in Choret we would instead write one canonical program for the entire system and then compile that single program into the two separate programs above. By doing so, we match sends and receives syntactically, making the matching automatic. Thus, we would write *(define/<~ (at P x) (at Q e))* to say “evaluate the expression *e* on process *Q*, and then send the

Racket Expressions	e	
Binding Forms	B	$::= X \mid (\text{at } P \ x)$
Choret Programs	P	$::= (\text{chor } (P \ \dots) \ T \ \dots)$
Choret Expressions	E	$::= (\text{at } P \ e \ \dots) \mid (\sim\> (\text{at } P \ e) \ Q)$ $\mid (\text{if } (\text{at } P \ e) \ E_1 \ E_2) \mid (\text{sel}\sim\> P \ ([l \ Q] \ \dots) \ E)$ $\mid (\text{let } ([B \ E_1] \ \dots) \ E) \mid (\text{let* } ([B \ E_1] \ \dots) \ E)$ $\mid (\text{set! } (\text{at } P \ x) \ E)$
Choret Terms	T	$::= (\text{define } B \ E) \mid (\text{define}/\sim (\text{at } P \ x) (\text{at } Q \ e))$

Figure 1: Choret Syntax

resulting value to P , where it should be called x .” Here, $(\text{at } Q \ e)$ just means “evaluate the expression e on process Q .”

There is another kind of communication that occurs in Choret: communication that is used to propagate *knowledge of choice*. We can see this above: when the Buyer determines whether the cost of the book is within their budget, the Seller has no idea which choice the Buyer made. Instead, we see the Buyer informing the Seller of their choice using the two copies of `send3`. The seller then branches on this by asking if the result of `recv3` is “buy”, and behaving appropriately in each case. The Choret program $(\text{sel}\sim\> P \ ([l \ Q]) \ E)$ means “ P informs Q that they are taking the branch labeled l , and then the entire system continues as E .” Combining this with the basic communication primitives above, we can rewrite the example above into a single Choret program:

```
(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])
      (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"))))))
```

Not only is the choreography shorter and more concise, but it’s no longer possible to mismatch the pairs of sends and receives. Because of this, choreographies offer an exciting property: deadlock freedom by construction [5]. Thus, users of Choret can write their code without fear of deadlocks.

Formally, the syntax of Choret is given in Figure 1. Like Racket more generally, Choret is split into terms (i.e., top-level definitions) and expressions (which return a value). Definitions, whether top-level or local, can either bind global variables or local variables at some process P . For instance, the Choret term $(\text{define}/\sim (\text{at } P \ x) (\text{at } Q \ e))$ binds the local variable x at process P .

By contrast, $(\text{at } P \ e)$ and $(\text{sel}\sim\> P \ ([l \ Q]) \ E)$ are both expressions. Selection is more powerful than previously suggested: P can send any number of labels to (distinct) processes, informing them all of the taken branch. The expression $(\sim\> (\text{at } P \ e) \ Q)$ computes the value of e at P and then sends the result to Q . Finally, we include traditional Racket expressions as Choret expressions, often extending them to describe where computation is taking place via the `at` syntax.

Our treatment of knowledge of choice via selection is the tradition in choreographic programming-

$$\begin{array}{lcl}
\text{Network } N & ::= & \dots (\text{All other Racket forms}) \\
& | & (\text{send } P \ e) \mid (\text{recv } P) \\
& | & (\text{choose! } P \ l \ N) \mid (\text{branch? } P \ ([l \ N] \ \dots))
\end{array}$$

Figure 2: Network Language Syntax

language design [15, 18, 20]. However, it leads to significant difficulties in developing choreographic libraries. As we will see in Section 3, the use of selections means that the process of splitting a choreography into different programs for each process requires multiple passes. However, most metaprogramming systems do not make writing multipass transformations possible, much less easy. Therefore, most choreographic libraries either change how they handle knowledge of choice or they perform the splitting at runtime. Either choice allows them to avoid the multiple passes required at compile time. Racket's macro system allows us to uniquely provide the traditional design with its traditional semantics (see Section 4).

3 EPP and the Network Language

Choreographies give a global view of the system. However, in order to execute a choreography, we must spit it into separate programs, one for each participant. In the choreographic literature, this transformation is referred to as *Endpoint Projection (EPP)*. In this section, we describe the design of our network language (that is, the target of endpoint projection) as well as the definition of endpoint projection itself. We discuss their Racket implementation in Section 4.

The syntax of our network language can be found in Figure 2. Unlike Choret itself, which manually reimplements the core Racket forms, our network language is described as four additions on top of Racket itself (implemented as simple macros). The form `(send P e)` evaluates the Racket expression `e` and sends the result to the process `P`, which is assumed to be different from the current process. Similarly, the form `(recv P)` receives a value from the process `P`, returning that value. We propagate knowledge of choice via the forms `(choose! P l E)` and `(branch? P ([l E] ...))`. The former informs `P` about which branch was taken, while the latter allows `P` to tell the current process which branch to take.

We now almost have enough information to formally define EPP. However, one difficulty arises, which is best described by example. Consider the following Choret program:

```

(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"))))

```

Here, we look at `A`'s boolean value and, if it's true, `B` returns "Left", otherwise `B` returns "Right". In order to allow this difference in `B`'s behavior, `A` informs `B` of which branch to take. Now, imagine trying to project a program for `B` from this choreography. Projecting each branch of the `if` is easy: the true branch projects to `(branch? A [l "Left"])`, the false to `(branch? A [r "Right"])`. These each wait for a message from `A` and either return "Left" if the message is `l` (for the former) or return "Right" if the message is `r` (for the latter), doing nothing otherwise. In order to give a single program

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

Figure 3: Definition of Endpoint Projection (Selected Parts)

for B , we need a program which receives a message from A and takes both behaviors, doing nothing only if the message is neither l nor r . We do this via *merging*. We define merging, written $N_1 \sqcup N_2$, as follows:

$$N_1 \sqcup N_2 = \begin{cases} \text{recursively merge} & \text{if } N_1 \text{ and } N_2 \text{ are matching Racket forms} \\
(\text{send } P \ e) & \text{if } N_1 = N_2 = (\text{send } P \ e) \\
(\text{recv } P) & \text{if } N_1 = N_2 = (\text{recv } P) \\
(\text{choose! } P \ l \ N'_1 \sqcup N'_2) & \text{if } N_1 = (\text{choose! } P \ l \ N'_1) \\
& \text{and } N_2 = (\text{choose! } P \ l \ N'_2) \\
(\text{branch? } P \ ([l_{1i} \ N_{1i} \sqcup N_{2j}] \dots & \text{if } N_1 = (\text{branch? } P \ ([l_{11} \ N_{11}] \ \dots)) \\
[l_{1k} \ N_{1k}] \dots & \text{and } N_2 = (\text{branch? } P \ ([l_{21} \ N_{21}] \ \dots)) \\
[l_{2k} \ N_{2k}])) & \text{and } l_{1i} = l_{2j} \\
& \text{and } \forall k, k'. l_{1k} \neq l_{2k'} \\
\perp & \text{otherwise}
\end{cases}$$

The merge function looks intimidating, but the only complicated case is `branch`; every other case merely checks to make sure that N_1 and N_2 are compatible before making a recursive call. In the case of `branch`, we need to combine the possible branches. If both N_1 and N_2 have a branch for some label l , then we recursively call merge on those branches. Any labels that either N_1 or N_2 have, but not both, are simply kept. If we apply this to our example above, we compute

$$(\text{branch? } A \ ([l \ \text{"Left"}])) \sqcup (\text{branch? } A \ ([r \ \text{"Right"}])) = \\
(\text{branch? } A \ ([l \ \text{"Left"}] \ [r \ \text{"Right"}]))$$

which behaves exactly as desired.

We use the definition of merging to define endpoint projection in Figure 3. This describes how to transform each of the forms of Choret into a network-language form. As an example, a choreographic `send ($\sim \rightarrow (\text{at } P \ e) \ Q$)` is transformed into `(send $Q \ e$)` for P and into `(recv P)` for Q . For any

process not involved, a Choret form will turn into `(void)`, a Racket standard-library function which returns “nothing.” Thus, every process only gets the information available to them in the choreography.

Note that Figure 3 only contains selected forms. The other forms are uninteresting; they recursively call EPP on their subforms and then return an “obvious” Racket analog of themselves. This pattern can be seen in the 1st lines of Figure 3.

Now that we have a mathematical definition of EPP, we can begin to implement it in Racket macros. However, this leads to some complications: Racket’s macro system is powerful, but expressing complicated, multi-pass transformations like EPP inside of them is still difficult. In Section 4, we introduce the tricks and tips that the Racket community has put together for this problem and describe how we use those tricks to implement Choret.

4 Racket Macro Expansion and EPP

We now explain the implementation of Choret. This implementation relies on the power of Racket’s macro system to allow us to perform merging and EPP at compile time. Thus, we begin by providing some background on the Racket macro system before finally describing the implementation of EPP using that system.

4.1 Background on Racket Macros

Racket canonizes its own syntax into data called *syntax objects*. These syntax objects are Racket data that represent Racket programs; *macros* are then simply functions that take and return syntax objects. Syntax objects themselves contain not only the abstract-syntax tree of a program, but also information such as scope and source locations. Thus, macros form a powerful metaprogramming facility.

Racket’s macro expansion is performed top-down, outermost to innermost, and fully expands all macros to their *core forms*. Thus, a macro “sees” its arguments in unexpanded form. While this is normally desirable, sometimes a programmer wants a macro to operate on *expanded* output. In order to do this, the macro calls `local-expand`, which invokes the macro expander directly. Calling `local-expand` on a syntax object returns its full expansion, which can then be parsed and analyzed using Racket’s `syntax-case` form.

The Racket core form `quote-syntax` turns data into a syntax object. Syntax objects obtained this way retain most of their lexical information (i.e. scope sets) [1], though certain scopes are pruned. However, a programmer can force `quote-syntax` to preserve all scopes using the `#:local` keyword. Since `quote-syntax` is defined as a core form, the macro expander will not touch it or the data it is transforming. Thus, programmers can use `quote-syntax` to prevent the macro expander from expanding some syntax, preserving it for other macros to see.

Finally, sometimes a programmer wants to communicate information across macros nonlocally; for instance, they may want to allow the expansion of one macro to determine how another expands globally. To do so, they can store that information in a *syntax parameter*. Syntax parameters allow for dynamic macro time bindings, which can be used to update a binding for expansions within an entire branch of the syntax tree.

4.2 The Implementation of EPP

In order to implement select-and-merge EPP as a library during compile time, we take full advantage of the Racket macro system. The top-level `chor` macro creates a syntax parameter representing the process

currently being expanded. It then loops through all of the processes in the choreography, expanding its body once for each process, setting its syntax parameter appropriately each time.

Most other macros in Choret don't use `local-expand` and instead directly rearrange them according to the EPP specification from Section 3, implicitly relying on the macro expander to expand their subforms. As described in that specification, they produce programs in our network language, which is also a collection of macros.

For most of Choret's expressions, this works beautifully. However, selections create branches, which need to be merged later. If we allowed the `branch` macro to be expanded fully to its core forms, the `merge` macro would not be able to detect when two branches need to be merged. Thus `(branch? P ([l E]))` expands to another `branch?` form wrapped with `quote-syntax`, with the `E` subform expanded, like `(quote-syntax (branch? P ([l $\llbracket E \rrbracket_A$])) #:local)`. The `merge` macro can then look for these hidden branches without fear that they will be expanded away by Racket.

All together, this design allows us to provide a traditional select-and-merge choreographic language design without requiring EPP to be performed at runtime. Other choreographic libraries, such as HasChor [21], mix the semantics of their network languages with EPP, allowing them to project the appropriate branch when required rather than performing full merges. This means two things. First, every process sees the entire choreography, which may not be appropriate in mixed-trust settings. Second, EPP can now block the execution of a program, potentially slowing down a system considerably. However, Racket's type system enables Choret to perform EPP fully at compile time.

5 Related Work

5.1 Choreographic Programming

Choreographic programming emerged from the process-calculus and session-type communities about ten years ago [4, 5, 18, 20]. Since that time, most of the work has been exploring theoretical aspects of EPP in lower-order settings: there was no ability to create subroutines or functions [5, 7–10, 13, 19]. Recently a fair amount of interest has come up in *functional* choreographic programming, which combines choreographic programming languages with λ calculi to allow for program abstraction and reuse [6, 14, 15]. In particular, this work is inspired by Pirouette, the first functional choreographic programming language [15].

As an outgrowth on the work on functional choreographic programming, many people have begun to experiment with *embedded* choreographic languages. This was started by HasChor [21], which embedded a choreographic programming language inside of Haskell using a freer monad. This method made it easy to implement a choreographic language. However, it lead to an unusual situation: endpoint projection was no longer a compile-time activity, but something that happened at runtime whenever a node needed the next line of its instructions. This methodology, with slight variations, has since been adapted by other embedded implementations of choreographic languages [16, 22].

The closest work to this is Klor, which is an embedded implementation of choreographies in Clojure, another Scheme-like language [17]. Like our work, Klor uses macros to implement a choreographic language inside of a LISP-like language with EPP at compile time. However, Klor handles knowledge of choice significantly differently. Whereas we, like most of the choreographic literature [6, 11, 14, 15, 18, 20], use selection messages to encode knowledge of choice, Klor instead uses a relatively new idea: agreement types [3]. These allow any data to be located at a *collection* of processes, and communication adds a process to that collection. When a choreography branches on data, then, any process in that collection knows which path to take. Doing so allows Klor to avoid merging, and therefore avoid the

need for `local-expand`. We, in contrast, choose to implement the traditional approach to knowledge of choice in choreographies. We are thus the only embedded implementation of choreographies with traditional select-and-merge EPP at compile time.

5.2 Embedded Languages via Racket Macros

The defining feature of Racket is its “Languages as Libraries” design, which entails an API for extending the language. A good example is Typed Racket, a sister language of Racket implemented entirely as a normal Racket library [23, 24].

A characteristic feature of many LISPs, like Racket, is the ability to easily embed domain specific languages (DSLs) using macros. However, macros in other LISPs tend to have certain issues. For example, macros in many LISPs use *symbols*—essentially immutable strings—to encode identifiers. However, this allows macros to ignore scope when manipulating identifiers: a macro may introduce identifiers which accidentally capture identifiers in the macro’s body, or it may introduce identifiers which are accidentally captured by bindings in the macro’s body. Such problems, among others, are often referred to as *macro hygiene*. Racket largely solves such issues using scope sets [12], which associates with each identifier a set of scopes. Racket uses such scope sets to determine the correct bindings for the expanded code.

While scope sets ensure macro hygiene, it is sometimes desirable to use unhygienic macros. For example, Choret sometimes needs to communicate which participant is currently being projected to the macros that perform EPP. We are able to do this by using Racket’s `syntax-parameter` [2] macro, which, when expanded, updates a compile time binding that is only visible in the body of the macro. Thus, we are able to selectively ignore hygiene when necessary, while writing hygienic macros by default.

6 Conclusion

Choreographies are a promising paradigm for concurrent programming. However, in order for them to live up to their promise, the community needs to rapidly develop and prototype new choreographic-language designs. Choreographic libraries are a promising method for doing so, but they rely on the metaprogramming capabilities of a host language. Because these capabilities tend to be weak, previous choreographic-library designers have developed clever new semantics for choreographic languages which can be implemented with those anemic capabilities. However, that has left the most-common design for choreographic languages—the traditional select-and-merge semantics—without the rapid prototyping advantages of choreographic libraries.

By developing Choret, we have shown that Racket’s macro system is strong enough to bridge this gap. In particular, we have implemented Choret as a choreographic library in Racket that performs EPP at compile time. While Choret uses advanced features of Racket’s macro system, the implementation is quite small—only 370 lines of code (excluding comments and tests). We hope that this small size means that other choreographic-language designers will build on Choret in order to test out their designs.

References

- [1] 3.21 *Syntax Quoting: quote-syntax*. Available at https://docs.racket-lang.org/reference/Syntax_Quoting__quote-syntax.html. Accessed March 3rd, 2025.
- [2] Eli Barzilay, Ryan Culpepper & Matthew Flatt (2011): *Keeping it clean with syntax parameters*. *Proc. Wksp. Scheme and Functional Programming*.

- [3] Mako Bates & Joseph P. Near (2024): *We Know I Know You Know; Choreographic Programming With Multicast and Multiply Located Values*. Available at <https://arxiv.org/abs/2403.05417>.
- [4] Marco Carbone & Fabrizio Montesi (2012): *Merging Multiparty Protocols in Multiparty Choreographies*. In Simon J. Gay & Paul Kelly, editors: *PLACES 2012, EPTCS 109*, pp. 21–27, doi:10.4204/EPTCS.109.4.
- [5] Marco Carbone & Fabrizio Montesi (2013): *Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming*. In Roberto Giacobazzi & Radhia Cousot, editors: *POPL 2013*, ACM, pp. 263–274, doi:10.1145/2429069.2429101.
- [6] Luís Cruz-Filipe, Eva Graversen, Lovro Lugovic, Fabrizio Montesi & Marco Peressotti (2022): *Functional Choreographic Programming*. In Helmut Seidl, Zhiming Liu & Corina S. Pasareanu, editors: *ICTAC 2022, Lecture Notes in Computer Science 13572*, Springer, pp. 212–237, doi:10.1007/978-3-031-17715-6_15.
- [7] Luís Cruz-Filipe, Kim S. Larsen & Fabrizio Montesi (2017): *The Paths to Choreography Extraction*. In Javier Esparza & Andrzej S. Murawski, editors: *FOSSACS 2017, Lecture Notes in Computer Science 10203*, pp. 424–440, doi:10.1007/978-3-662-54458-7_25.
- [8] Luís Cruz-Filipe & Fabrizio Montesi (2016): *Choreographies in Practice*. In Elvira Albert & Ivan Lanese, editors: *FORTE 2016, Lecture Notes in Computer Science 9688*, Springer, pp. 114–123, doi:10.1007/978-3-319-39570-8_8.
- [9] Luís Cruz-Filipe & Fabrizio Montesi (2016): *A Core Model for Choreographic Programming*. In Olga Kouchnarenko & Ramtin Khosravi, editors: *FACS 2016, Lecture Notes in Computer Science 10231*, pp. 17–35, doi:10.1007/978-3-319-57666-4_3.
- [10] Luís Cruz-Filipe & Fabrizio Montesi (2017): *On Asynchrony and Choreographies*. In Massimo Bartoletti, Laura Bocchi, Ludovic Henrio & Sophia Knight, editors: *EPTCS 2017, EPTCS 261*, pp. 76–90, doi:10.4204/EPTCS.261.8.
- [11] Luís Cruz-Filipe & Fabrizio Montesi (2020): *A Core Model for Choreographic Programming*. *Theor. Comp. Science 2020* 802, pp. 38–66, doi:10.1016/j.tcs.2019.07.005.
- [12] Matthew Flatt (2016): *Binding as sets of scopes*. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL ’16*, Association for Computing Machinery, New York, NY, USA, p. 705717, doi:10.1145/2837614.2837620. Available at <https://doi.org/10.1145/2837614.2837620>.
- [13] Saverio Giallorenzo, Fabrizio Montesi & Maurizio Gabbrielli (2018): *Applied Choreographies*. In Christel Baier & Luís Caires, editors: *FORTE 2018, Lecture Notes in Computer Science 10854*, Springer, pp. 21–40, doi:10.1007/978-3-319-92612-4_2.
- [14] Eva Graversen, Andrew K. Hirsch & Fabrizio Montesi (2024): *Alice or Bob?: Process Polymorphism in Choreographies*. *Journal of Functional Programming (JFP)* 34, p. e1, doi:10.1017/S0956796823000114.
- [15] Andrew K. Hirsch & Deepak Garg (2022): *Pirouette: Higher-Order Typed Functional Choreographies*. In: *POPL 2022*, 6, pp. 1–27, doi:10.1145/3498684.
- [16] Shun Kashiwa & Lindsey Kuper (2024): *ChoRus: Library-Level Choreographic Programming in Rust*. In: *Choreographic Programming (CP)*. Available at <https://users.soe.ucsc.edu/~lkuper/papers/chorus-cp24.pdf>.

- [17] Lovro Logović & Sung-Shik Jongmans (2024): *Klor: Choreographies for the Working Clojurian*. In: *Choreographic Programming (CP)*. Available at <https://pldi24.sigplan.org/details/cp-2024-papers/15/Klor-Choreographies-for-the-Working-Clojurian>.
- [18] Fabrizio Montesi (2013): *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen. Available at https://www.fabriziomontesi.com/files/choreographic_programming.pdf.
- [19] Fabrizio Montesi (2015): *Kickstarting Choreographic Programming*. In Thomas T. Hildebrandt, António Ravara, Jan Martijn E. M. van der Werf & Matthias Weidlich, editors: *LNPSE 2015, Lecture Notes in Computer Science* 9421, Springer, pp. 3–10, doi:10.1007/978-3-319-33612-1_1.
- [20] Fabrizio Montesi (2022): *Introduction to Choreographies*. Cambridge University Press, doi:10.1017/9781108981491.
- [21] Gan Shen, Shun Kashiwa & Lindsey Kuper (2023): *HasChor: Functional Choreographic Programming for All (Functional Pearl)*. In: *International Conference on Functional Programming (ICFP)*, doi:10.1145/3607849.
- [22] Gan Shen & Lindsey Kuper (2024): *Toward Verified Library-Level Choreographic Programming with Algebraic Effects*. In: *Choreographic Programming (CP)*. Available at <https://arxiv.org/abs/2407.06509>.
- [23] Sam Tobin-Hochstadt & Matthias Felleisen (2008): *The design and implementation of typed scheme*. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '08*, Association for Computing Machinery, New York, NY, USA, p. 395406, doi:10.1145/1328438.1328486. Available at <https://doi.org/10.1145/1328438.1328486>.
- [24] Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt & Matthias Felleisen (2011): *Languages as libraries*. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11*, Association for Computing Machinery, New York, NY, USA, p. 132141, doi:10.1145/1993498.1993514. Available at <https://doi.org/10.1145/1993498.1993514>.