Implementing, Extending and Comparing Choreographic Programming Languages and Session Types

Master thesis in the department of Computer Science by Daniel Stricker
Date of submission: January 28, 2026

1. Review: Prof. Dr.-Ing. Mira Mezini
2. Review: Simon Daniel
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department

Software Technology

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB TU Darmstadt

Hiermit erkläre ich, Daniel Stricker, dass ich die vorliegende Arbeit gemäß § 22 Abs. 7 APB der TU Darmstadt selbstständig, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt habe. Ich habe mit Ausnahme der zitierten Literatur und anderer in der Arbeit genannter Quellen keine fremden Hilfsmittel benutzt. Die von mir bei der Anfertigung dieser wissenschaftlichen Arbeit wörtlich oder inhaltlich benutzte Literatur und alle anderen Quellen habe ich im Text deutlich gekennzeichnet und gesondert aufgeführt. Dies gilt auch für Quellen oder Hilfsmittel aus dem Internet.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§ 38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 28. Januar 2026

Daniel Stricker

Contents

Abstract

In this work we explore programming languages and their features in the context of distributed programming. We first explore session types which promise to proof correct communication between peers. Another paradigm which promises a similar feat is choreographic programming. We inspect ChorLean, a choreographic programming language which uses a dependent type system to add even more guarantees than other choreographic programming languages. We try to extend it with more features and trying to prove its deadlock freedom. We explore the fundamental differences between choreographic programming languages and multitier programming. Furthermore we look into which alternative choreographic programming languages exist and we find an interesting language λ^{χ} which promises to build a common ground for all state of the art choreographic programming languages. The only missing piece: A mechanization, which we try to implement and to even extend it using a dependent type system.

1 Introduction

Distributed systems are important and hard to master. Choreographic programming was established to help and an alternative are session types.

As it turns out, many goals we set ourselves or possible solutions we tried to find, turned out to be either impossible or unfeasible. We had no other option than to shift our focus and goals as needed. Therefore, this work also represents all our attempts and shifts. All of our code is published publicly on GitHub[23]. Even though we could unfortunately not find a solution to any single big problem, we think that the main contribution of our work is to show the feasibility or infeasibility of multiple adjacent smaller problems. This could serve as a base to inspire further research especially because we try to serve as an introduction into choreographic programming, session types, ChorLean and multiple other distinct programming languages in this space, all with unique viewpoints. Another goal of To get a broader view of choreographic languages and to have better comparison.

2 Session Types

The original goal of this work was to find a way to combine session types and choreographic programming. To possibly achieve this, one has to understand the in and outs of both. This is made much harder because both are state-of-the-art research topics each with its own variations, challenges and other intricacies. As you will find out we had to shift goals regularly while doing this work. In our opinion this work is a perfect starting point for further research

Session types[12, 20, 7] usually require your programming language of choice to support a linear type system where every variable can only be used once. It is possible to implement session types in any programming language but checking linearity at compile time is hard and therefore often omitted. Checking it at runtime defeats the purpose. This is problematic for adoption in general programming languages. Thiemann[24] presents a functional programming language which uses a novel way to represent session types. They use callbacks just like in web programming. After every function you call you also have to provide the continuation of your program which decides what happens with the result value of the function. By nesting this procedure, you can write arbitrary programs.

We will implement Thiemann's[24] session types using Lean to possibly combine them with the existing choreographic programming language ChorLean:

```
inductive Type'
  | int : Type'
  | bool : Type'

inductive Session
  | _! : Type' -> Session -> Session
  | _? : Type' -> Session -> Session
  | end' : Session
  | internalChoice : (Fin k -> Session) -> Session
  | externalChoice : (Fin k -> Session) -> Session
open Session
```

First we have to define some primitive data types for our language to be based on (lines 1–3). The main part is the inductive `Session` definition. With session types we want to define a protocal of communication which is statically always fulfilled. One could think about which communication primitives exist. The initial logical choice is send and receive.

We will start with exactly them, `_!` (send) in line 6 and `_?` (receive) in line 7. They have two parameters each. First the type of the data to be sent/received and another session which acts as a continuation. Thiemann's main achievement is to abuse continuations. The thing is that for session types you have to guarantee that every session is only accessed once which can also be called a linear type system. Designing session types with a linear type system is hard and some languages don't even support linear type systems. Turns out continuations fullfill the same goal and are more often supported than linear type systems.

This is also called continuation passing style and means that after every operation we have to pass the next opeartion the function we called. The function usually does not return normally but instead calls the continuation with its result.

To end this unending cycle we introduce `end'` (line 8) which is used as a continuation when we do not want futher operations to happen.

The problem is we cannot change flow of our program yet we can only statically send or receive as often as we want. Therefore, we add `internalChoice` (line 9) and `externalChoice` (line 10). `internalChoice` means that we do some computation which produces a number of type `Fin k` which we then send to our communication partner. He then executes the expression with the `Session` type that is returned by the function which is passed to its `externalChoice` counterpart.

Let us look at a small example:

```
1 def server := _? int $ _? int $ _! int end'
2 def client := dual server
3
4 def equivalentClient := _! int $ _! int $ _? int end'
```

`binary` is a session type which represents a server which receives an `int` and then another `int`. Then it (no necessarily) does some operation on those two arguments and sends the result back. Then the session ends.

A direct communication parter's session can easily be generated automatically with a `dual` function which just recursively flips sends/receives and `internalChoice/externalChoice`:

```
def dual
    | _! t s => _? t $ dual s
    | _? t s => _! t $ dual s
    | end' => end'
    | internalChoice sessionSelector =>
        externalChoice (fun label => dual $ sessionSelector label)
    | externalChoice sessionSelector =>
        internalChoice (fun label => dual $ sessionSelector label)
```

We also added loops and recursion which makes everything more complicate. Unfortunately we could not reach our goal of combining session types with ChorLean. ChorLean's viewpoint of its distributed code is just too different. ChorLean takes a global viewpoint from the top where we only need a `com` primitive at one place which handles sending and receiving at the same time.

Session types are fundamentally built to model one side of a communication. It is possible to generate the other side by using functions like `dual` but fundamentally a session type only represents one side which is not compatible with choreographic programming languages.

3 ChorLean

Let us now focus on choreographic programming[22]. Our base is the choreographic programming language ChorLean[2, 6] because we have convenient first-hand access to the developers of it. They have a deep understanding of the source code which was a massive help to kick-start further research on ChorLean specifically and the topic at large. This is one of the reasons why our first goal was to find a way to extend ChorLean with session types. As it will turn out later, according to our knowledge this is impossible. This work also serves as an introduction to ChorLean because as of our knowledge there currently exists no other public documentation or introduction for it. We think that an introduction especially to ChorLean is paramount for the further development of choreographic programming languages and to showcase more real life use cases of dependent programming languages. We hope this could inspire further developments of existing solutions or possibly completely new ideas by getting a new perspective on the solution space of dependent programming languages.

3.1 Choreographic programming

Distributed programming poses unique challenges. Specifically deadlocks are a unique threat when communication between peers has to be managed. One way to (usually provably) get rid of this whole class of problems is by using a choreographic programming language. Instead of writing a program for each peer, you instead write one central program where the placement of tasks or data to specific peers is part of the programming language. With this you implicitly define a communication protocol. Because your code defines the protocol, it trivially fulfills the protocol. Your central code is then automatically compiled into separate programs for each peer. If you can prove that your choreographic programming language of choosing is correct then you are guaranteed to never get deadlocks. But this proof even though you only have to do it once is not a trivial endeavour.

In fact, many choreographic programming languages are just implemented without any backing proof or even a proof sketch.

3.2 Lean and dependent types

Manually writing a proof for a choreographic programming language would surely be the best solution, but we can use another tool which gets us multiple advantages. This tool is a dependent type system. Most widely known choreographic programming languages use normal type systems. Turns out that normal type systems are more limited and limit the possible guarantees communication safety. You know what kind of data to expect but modelling from whom you get the data at type level is just impossible. Therefore, you have to implement this logic by yourself. This will usually happen at runtime. Except if you prove your code rigorously then you won't get any guarantees that your logic is correct and as everyone knows, bugs can happen. For some tasks want to have these better guarantees. With a dependent type system, we can encode the location of data into our types. As long as the type system is to be trusted we then have a very good guarantee that we always know who our next communication partner is. Unfortunately, this also isn't a full guarantee because what if the types we wrote are faulty? But it is already a big step in the correct direction.

3.3 Implementation

Let us look at an example where we implement an online rock paper scissors game using ChorLean. We will use this as an introduction to ChorLean specifically but also to the programming language Lean and moreover how choreographic programming languages work generally. Instead of writing two separate programs, we can write one program which contains the logic for both players (one of the biggest advantages of choreographic programming languages). Player 1 will act as a kind of server because the winner is decided on his side. We could have easily added an intermediary server, but we omitted it for simplicity's sake. The following listing contains the most important part of the implementation, the `play` function, with auxiliary code omitted:

```
1 def play: Choreo [Player1, Player2] target proof_target_in_census (Option
      Role) :=
2   do
```

```
3     let handPlayer1 <- (
4       locally Player1 readInput
5     )
6     let handPlayer2 <- (
7       locally Player2 readInput
8     )
9     let handPlayer2Shared <- com Player1 handPlayer2
10
11    let gameResultLocated <- locally Player1 (
12      do
13        pure $ decideWinner handPlayer1.un handPlayer2Shared.un
14    ) >>= com Player2
15    let gameResult := gameResultLocated.un
16
17    let _ <- locally Player1 $ printResultMessageFor Player1 gameResult
18    let _ <- locally Player2 $ printResultMessageFor Player2 gameResult
19
20    pure gameResult
```

In line 2 we have to use `do` which is common syntax for functional programming languages to simplify writing code which uses monads. In this case `Choreo` is the monad we want to return as you can see in line 1. Because of this we have to use `do`. A `Role` in this context is a location where the code runs. We will use the term location and role interchangeably. In this example this is either the constant `Player1` or the constant `Player2`. We will discuss the further details of the return type later. Therefore, let us as first ignore the return type of `play` in line 1.

In lines 4–5 we simply read the input of player 1 which decides which hand he plays and give it the name `handPlayer1`. Analogously we do this for player 2 in lines 7–8. As you can see, we cannot call the function `readInput` directly because the compiler would not know on which location to execute this. We use the function `locally` to specify a single role where the function should be executed. ChorLean and other choreographic programming languages have multiple more auxiliary functions which for example would allow us to execute functions on multiple roles.

`handPlayer1` (and `handPlayer2` respectively) are of type `Located (List Role) String` (we simplified the type parameters for simplicity's sake). In this case, the list of roles consists of only `Player1` (and `Player2` respectively) which means each input hand (of type `String`) is located at its respective role. This type guarantees that we cannot accidentally write code where Player 1 accesses a value which it can't because it is located at Player 2. Writing such a piece of code would directly show us a type error before even starting the program. Because of this, in line 9 we explicitly transfer `handPlayer2` to `Player1`.

`handPlayer2Shared` is now located at both `Player1` and `Player2`. The compiler keeps track of this and allows us to use this value at both locations locally.

Now we have to decide who is the winner. We do this, as stated before, at player 1 which was chosen arbitrarily. In line 11 we can see that we call `locally` with `Player1`. For understanding this code snippet it is not necessary to understand what `do` and `pure` in line 12 and line 13 are. So we glance over them here. In line 13, we call the function `decideWinner` which takes two arguments of type `String`. As we said before, `handPlayer1` and `handPlayer2Shared` are of type `Located` which is not a compatible type to `String`. Therefore, we have to call the `un` function. `x.un` is just syntax sugar for `un x` where `x` is any expression. `un` unpacks the value of the `Located` container. The important thing here is that the `un` function can only be called if the compiler can proof that the role which runs this code has access to that value. In this case `Player1` has access to both `handPlayer1` and `handPlayer2Shared` as lined out before.

Player 1 now knows who won the game. The winner is saved in `gameResultLocated`. This value would again be located only at `Player1`. Therefore, we share it with `Player2` using the operator `>>=` in line 14 which is just a shorter way to write `com` (as in line 9). `gameResultLocated` is located at both players and then unpacked at line 15 so that both players can access the value.

The last step in lines 17–18 is to print a message informing each play if they have won or lost. The result of these two lines is not needed. Therefore, we save them into `_` which is a throwaway name. We have to select a name because else it would not be integrated into the wrapping monad. But these details are not important for this section. At last, we return the game result to the calling function in line 20. We do this to possibly call `play` again if this game resulted in a draw. This also shows that choreographies like the function `play` are just pure functions and can be reused or take parameters like usual to change behaviour. This is possible because we are using monads to wrap any side effects that occur in this code. In fact, Lean is a pure functional programming language which means that this is the only way to represent side effects. With this we have seen how choreographic programming languages are usually structured from the perspective of a library user but there are multiple characteristics which are unique for pure functional languages and even more specifically for dependent programming languages. We have already mentioned some guarantees we get.

Getting deep into the implementation of ChorLean is out of scope for this work, but we will showcase the central type `Choreo` because looking at its declaration already gives us deep insight into how some unique features of `ChorLean` work. Here is the declaration (code from ChorLean[6]):

```
1 inductive Choreo:
2   (census: List Role) ->
3   (target: Hidden Role) ->
4   (proof_target_in_census: target ∈ cen) ->
5   (expressionType: Type) ->
6   Type 1
```

The type definition directly shows us the most important parts of ChorLean and how it uses Lean's dependent type system. Every choreography (i.e. mostly all the code) is of type `Choreo`. Let us go through all type parameters one by one. `census` (line 2) is a very important one here. It is a list of `Role`. `Role` is strictly speaking an arbitrary type but for the sake of simplicity let us think of it as a simple enum with one value for every possible participant. Note that we have to statically know all possible participants in advance which turns out is an interesting and important restriction.

The two executables which are generated by `ChorLean` for both locations are equivalent. The difference in behaviour is distinguished by identifying who the `target` (line 3) or in other words the executor of this program is. This is usually done by passing a parameter to the executable. For example let `rock-paper-scissors` be our executable, and we want to start the program for player 1:

```
1 ./rock-paper-scissors Player1
```

This is, so the program can decide which role to assume in the code. `target` is possibly the most important parameter. One could think that a simple global variable would suffice. It is essential that `target` is part of the `Choreo` type because it allows us to have checks at compile time. Simply think about the power we get if we are able to write something like the following code. This is just (a little misrepresenting) pseudocode but gets the point access well enough in our opinion:

```
1 if value.owner != target then
2     throwTypeError
```

This is possible because Lean supports dependent types. We say `Choreo` depends on the value `target`. Think of it like `target` being just another value of your program and not a type but `Choreo` can access this value at compile time and alter its behaviour accordingly. The whole implementation of ChorLean is based on this fact. In line 1 we can check if the target (i.e. the executor of the program) as provably access to the value. Remember that this code is executed at compile time and throws a type error before the program is even started. Getting this kind of power is nowhere near free. *Provably* is the key term here. The compiler cannot always proof everything we want from it by itself. This

is the reason why throughout the implementation code of ChorLean the authors often have to show the compiler manually that some fact are always true by writing proofs. Luckily, Lean is also a very capable theorem prover which helps a lot in this process. Being obligated to understand and write sometimes non-trivial proofs is in our opinion the biggest disadvantage of ChorLean when you want to write non-trivial programs or have to extend some additional functionality in ChorLean.

The next parameter `proof_target_in_census` (line 4) is not exactly very complicated to understand but in our opinion shows the disadvantages of needing to always have a proof for everything. This parameter represents a proof that shows that `target` is indeed part of the `census`. Proving this is not especially hard, but it is cumbersome that such a parameter has to exist in the first place. This is very representative for the multitude of proofs one has to pass and keep around in the implementation code of ChorLean.

`expressionType` (line 5) just declares which kind of expression is returned by this specific `Choreo`. This is to guarantee that not only the location of every value is known at every time but also the expression type like we are used to. Think of a type like `String` at that place for example. Line 6 basically just says that `Choreo` is a type which depends on other types but this is not essential for our understanding here.

3.4 Exceptions

We also tried to add exceptions to ChorLean. For this we have researched multiple solutions[10, 16, 15] but none were a fit or were out of scope. Instead of exceptions, we implemented a 2-phase-commit protocol[19] by hardcoding a `Coordinator` role which acts as a middleman for all communication and communicates rollbacks to all peers. This was not successful either. Even if successful it would have only solved a limited amount of failures. If the coordinator crashes no rollbacks are possiblel.

3.5 Group chat

In our opinion to understand the ins and outs of a library or topic in general we also have to understand alternatives which solve the same problems. A solution never exists in a vacuum. Another solution which also makes it possible to write one program for multiple locations is called multitier programming. We specifically take a look at ScalaLoci[1].

We want to give a comparison to choreographic programming languages and specifically ChorLean. Keep in mind that we are comparing from the perspective of ChorLean and therefore can give more details on ChorLean and give more surface level intuition about ScalaLoci to keep an acceptable scope.

We implemented some examples in ScalaLoci by ourselves similar to our rock paper scissors with ChorLean. Another example program we were interested in was a group chat application where participants can freely start a program to join and leave group chats to send messages. At first, there was no specific reason why we wanted to implement this. This was also a rather trivial example to implement[23] with ScalaLoci. Nonetheless, we found this example rather interesting because it was so nicely interactive.

Hence, we tried implementing it in ChorLean. We quickly found out that the previously trivial problem was now a big task. At first, we thought that because of the different exposed API of ScalaLoci and ChorLean this problem was just harder to express in ChorLean. The main problem was that the `Choreo` type of ChorLean has a type parameter `census` as we have seen before. This parameter has to be set to a list of all possible communication members. This has to be known statically which is the main problem. With a group chat it must be possible that an indefinite amount of users can be added or left. Moreover, when starting the ChorLean executable you have to decide which specific role out of the list of possible roles you are. But how would you even know which roles are already taken?

We found one very hacky solution for this problem. By defining an arbitrary limit how many chatters can connect at the same time, we were able to define the roles like in the following listing. For this example we limited the amount of concurrent chatters to 4, but we could have selected a higher limit:

```
1 inductive Role where | Server | Chatter1 | Chatter2 | Chatter3 | Chatter4
```

But how does a chatter know which role is available. One solution is to iterate over every possibility and check if the program crashes when trying to connect. The port which is taken by each role also has to be statically assigned and because of this the program would instantly crash when trying to take the same port as another running program. In fact the current implementation of ChorLean is limited by the amount of network ports which are available, i.e. 65536 (16 bit) possible ports and therefore this is currently a hard limit on possible concurrent chatters. We have not done any further inspections if this port limitation can be easily circumvented.

But we tried to change ChorLean to allow dynamic roles by using `String` as identifiers of roles instead of `inductive` definitions. For this case think of `inductive` definitions

as enums from other programming languages. This does unfortunately not solve a big problem. To still have proven guarantees of behaviour like that chat messages are only sent to people in the same group we have to assign groups to chatters. This assignment has to be done dynamically because as a chatter we want to be able to switch groups. This dynamicism unfortunately counteracts any static proofs.

After further investigation, it turns out that a group chat application is probably impossible with ChorLean[9]. In theory, it could be possible to implement such features in choreographic programming languages generally but for this we would have to implement all expressions which talk about roles with quantifiers to abstract over the specific roles. But as far as we can tell no one has tried such a solutions. The generalization of this problem is that with choreographic programming languages you are generally not able to define arbitrary topologies like usual peer-to-peer or ring networks. Using multitier languages like ScalaLoci topology definition is a primitive endeavour.

This is not the only difference between choreographic programming languages and multitier languages. Distributed data structures are (usually) trivial with choreographic programming languages because you can generally annotate any type with a location. For example (pseudocode):

```
linkedListLocal       = 1 :: 2 :: 3 :: nil
linkedListDistributed = 1@A :: 2@B :: 3@A :: nil
```

This represents a linked list with numeric data elements 1, 2 and 3. By annotating each value with its specific location (in this example A or B) we can easily build a distributed linked list with syntax very similar to a usual local implementation.

Another advantage of choreographic programming languages is that it is usually possible to easily compose multiple choreographies or abstract over specifics to easily reuse choreographies. Specifically in ChorLean each choreography is a pure monadic value which can be called multiple times, can be passed around. In some choreographic programming languages like Klor (more on it later) choreographies can be created dynamically at runtime and even sent over the wire to another role who can modify and execute it. In multitier languages this is not possible because it is missing a similar abstraction of a distributed program in of itself. An easy way to imagine the main difference why those differences arise is that choreographic programming languages have a more global or top down viewpoint and multitier languages are in a way more from the viewpoint of individual locations.

further understand the context of choreographic programming languages and to see which alternatives exist

3.6 Proving Deadlock Freedom

An open task of ChorLean is to proof that it is free of deadlocks. The authors of ChorLean already sketched a proof but a full proof could act as full convincement that dependent types and Lean are viable to solve real world distributed software problems.

Thiemann[24] suggested that it could be proven using interaction trees[25]. We investigated this and found cgraphs[11] which sounded promising. Unfortunately it is written in the programming language Coq which we are not exactly familiar with it. Furthermore, it turns out that a simple example proof using cgraphs is 1500 lines of code long and consists of *very* dense code. Other[11, 25, 13] possibilities turned out to have similar problems.

4 Other choreographic programming languages

Having found out that session types and choreographic programming are incompatible our goal shifted specifically to choreographic programming languages. ChorLean is very advanced because it uses dependent types to guarantee deadlock freedom. Nonetheless, much can be learned by investigating other choreographic languages with vastly different implementations and goals. For example Klor's[17] main goal is ease of use and pragmatism. They sacrifice rigor but try to be more appealing to the industry. Inspecting multiple choreographic programming languages to find a useful influence for ChorLean was another goal of ours.

By comparing multiple choreographic programming languages we can further appreciate features or unique quirks of ChorLean but could also find disadvantages. Before comparing it with multiple other languages it was not trivially clear to us why authors of other choreographic programming languages chose their specific implementation. At first look it seemed to us that dependent types are superior because of more power. Let us inspect if we can learn about hard tradeoffs or maybe some also some elegant inspirations to make ChorLean better. We implemented similar smaller example programs in multiple choreographic programming languages to have baseline to compare them with each other.

4.1 Klor

ChorLean uses the powerful type system of Lean to implement its static guarantees and uses pure functional programming to make errors less likely. These features do come at a cost though. The language is complicated and one always has to think about conforming to strict types and sometimes one even has to provide proofs for facts which are clearly true in the head of the programmer.

In our opinion this added complexity hinders mainstream acceptance. For the usual programmer the added guarantees are not enough reason to get out of their comfort zone and to learn those whole new concepts. Even for the mathematically inclined programmer there is also glaring issue with ChorLean specifically: Even though it seems to be an elegant solution we do not in fact have formal proof to back it up. Another strategy in the complete opposite direction would be Klor[17].

Klor is a choreographic programming language implemented in Clojure. Clojure is part of a bigger family of programming languages called Lisps[5]. One could call Lisps the antithesis to pure functional programming languages with dependent types like Lean.

The first lisp was developed by John McCarthy in 1960[18] which started a family of similar languages but the concepts are usually the same. The main premise of Lisp is that every program can be represented by a very simple syntax. Every program and therefore every expression is just a simple linked list with symbols as data in each cell. At first glance a more simple syntax seems nonessential and can even seem unreadable for the untrained eye because of many explicit parentheses.

These extremely limited syntactic forms have a big advantage. Writing code and values uses the same syntax because data is also based on linked lists in Lisp. One thing we can easily do with data is modify it. Every programmer can trivially take a list apart, rearrange it without much mental overhead.

The key insight here is that every code expression is easily modifiable just like normal linked lists. This leads to a trivial style of writing macros. In other languages like C you have to use complicated pre-processors which are in turn fully turing complete second language you have to understand every intricate detail about.

Macros are unlike in other languages not an afterthought but an essential tool to implement features. Usually a lisps consist of a small set of language primitives and often more interesting features are implemented using macros.

In comparison to Lean, Lisps are usually dynamically typed and support multiple programming paradigms equally, like functional programming, object-oriented programming or procedural programming. In fact, Clojure is a rather modern Lisp and goes a more opinionated way. It has a big focus and strongly prefers functional programming and immutable data structures.

Lean also has macros, but we have not explored them further. In our opinion, because the elegant design of Klor is very promising it could be an interesting idea to inspect how macros could be used to implement a more simple syntax for ChorLean which would

reduce some boilerplate or even maybe allow something like more dynamically selecting the executor role.

Type checking with macros makes it possible to have custom error messages. In Lean, we profit from a rigorous type system which is provably correct. The disadvantage because of its generality is that we have to make due with obscure error messages. We know that something is wrong with our implementation but knowing what is wrong is another entire task in itself. If we instead write our (comparatively very limited) type system by ourselves, we can write custom error messages. For example “Role X is required here, but you provided Y” would be very clear and uses domain language to communicate the error. With our domain being choreographies in this example. Lean does not know of our domain and therefore can only provide general type errors.

Also worth mentioning is that almost all language features can be used completely seamlessly as usual. When using type systems for this task, you always end up with data which has some complicated type. This data then can’t be simply used by some pre-existing functions. You usually have to at least wrap/unpack your data manually or use monads.

Let us look at an example:

```
1 (defchor rps-start [A B] (-> #{A B}) []
2   (let [a (A (string->hand (read-line)))
3         b (B (string->hand (read-line)))
4         winner (A (select-winner [a (B->A b)]))]
5     (A (prn "Winner:" winner))))
```

We define a choroegraphy by calling `defchor` (line 1) and providing the types. We say that `A` and `B` are possible communication partners. In line 1 we also define that this choreography returns a value which is located at both A and B denoted by the `#` with curly braces which denote a set in Clojure. Note that the location is typed (and statically checked by a macro) but the actual value stays dynamic because Clojure is dynamic.

In lines 2 we read the input from the console of `A`, convert the string into a hand enum and assign it to variable `a`. The `A` in this line is necessary to tell the compiler where to execute the functions, in this case at `A`. `a` is therefore only located at `A`. This is analogous for line 3 and `B`.

The interesting part about line 4 is `(B->A b)`. There we can see the interesting syntax sugar a macro implementation can give us. The macro detects that `A` and `B` are possible roles and allows this syntax with a arrow to send a value from one peer to another. After printing the result (here only at `A`) we are finished.

4.2 Mechanization of λ^{χ}

As it turned out, specifically λ^{χ}[21] piqued our interest. It is a very recent attempt to take the (among researchers) widely established features and semantics of choreographic programming languages and to break them down to their cores. λ^{χ} simplifies choreographic languages to a relatively simple set of rules with the lambda calculus as its base. Furthermore, they even provided a full mathematical proof for its deadlock-freedom. This is a major feat in of itself. Such a proof is cumbersome and has to be done manually and for each choreographic programming language individually. Because of this, many only provide a proof sketch or omit the proof for their language entirely. Such a proof is also missing from ChorLean. λ^{χ} was specifically designed to be a basis for other languages. If we could translate ChorLean into the basic building blocks of λ^{χ} we would get a proof for free. Exactly this was one of the goals of λ^{χ}. The only missing part of λ^{χ} is that it is missing an implementation. The theoretical work was done rigorously but there exists no implementation. Providing an implementation of λ^{χ} using Lean4 (which is the same language ChorLean uses) was therefore another goal of ours.

A key insight of λ^{χ} is the semantics which all choreographic programming languages follow. That is, they are neither strict nor lazy. According to Plyukhin et al.[21] λ^{χ} this seems to be the main reason why no one could find a simple core set of rules for a choreographic lambda calculus. Chorλ had found a set of working rules but λ^{χ} could reduce the number significantly.

A major feat of λ^{χ} is its simplicity. By developing the language from the ground up with a fresh perspective and using different abstractions λ^{χ} found out that multiple rules used by Chorλ[4] were in fact unnecessary because they could be replaced by other rules. They even found out that Chorλ was missing some rules. They are very specific and rare cases but nonetheless it is a sign that λ^{χ} uses a very structured and sensible approach if they can find that so easily.

In this section we will give an overview over our mechanization (or implementation) of λ^{χ}. As far as we can tell, there currently does not exist any mechanization of λ^{χ}.

First we have to define some auxiliary types. That is `HList` where the "H" stands for homogenous. This is an example taken from the official Lean language website[8]:

```
1 inductive HList {α : Type v} (β : α -> Type u) : List α -> Type (max u v)
2 | nil : HList β []
3 | cons : β i -> HList β is -> HList β (i :: is)
4
```

```
5 infix:67 " :: " => HList.cons
6
7 notation "[" "]" => HList.nil
```

We don't have to understand all of this code but the broad understanding is that we define a new kind of list with its usual constructor functions `nil` (line 2) and `cons` (line 3). The special part about this list is that it can contain values of multiple different types. Most importantly, it also knows which type is at which position.

We also define some shortcut notations. Normal lists in Lean are usually constructed with `::` or with `[` and `]`. We define operators with the same names in lines 5–7. They construct a `HList` now.

We want to implement (or embed) a new language in Lean. Therefore, we have to make a distinction between types in our language (λ^{χ}) and the types of our host language (Lean) which implement the functionality of λ^{χ}. Let us take a look at the following type definitions:

```
1  inductive NType : Type 1
2    | func : (In : NType) -> (Out : NType) -> NType
3    | arbitrary : (α : Type) -> NType
4    | unit
5
6  abbrev NType.de (nType : NType) : Type :=
7    match nType with
8      | .func In Out => In.de -> Out.de
9      | .arbitrary α => α
10     | .unit => Unit
```

In line 1, we define a new type `NType`. In λ^{χ} this part of the language is called the network language, therefore we use the prefix "N" for all types and functions because they are very similar to later code we will introduce. `NType` represents all possible types in λ^{χ}. Those are functions (`func` (line 2)), an `arbitrary` (line 3) type which represents any atomic value like a boolean, string or number. The atomic data type is represented by α Finally, we also provide a `unit` (line 4) type.

Now we know which types can exist in λ^{χ}. In line 6, we define a function (`abbrev` basically just inlines this function) which is called `de`. It translates from the λ^{χ} type world to Lean types. `func` is translated to a common Lean function (line 8). An `arbitrary` α type is translated to just a α type (line 9). This is possible because α is of type `Type` (see line 3). `Type` is the type of Lean types which explains why this is possible. This is an elegant solution to abstract over a big amount (in fact, infinite) amount of types. This is possible

because of Lean's very expressive type system and would be harder or impossible with other host languages. Finally, `unit` is just Lean's `Unit` (line 10).

Let us look at the following listing:

```
1  abbrev NProcessName := String
2
3  inductive NProcedureName
4    | f
5    | g
6    | h
7
8  abbrev NProcedureName.type : NProcedureName -> NType
9    | f => NType.arbitrary String
10   | g => NType.func (NType.arbitrary String) NType.unit
11   | h => NType.arbitrary Nat
12
13 abbrev NProcedureName.argumentCount : NProcedureName -> Nat
14   | f => 0
15   | g => 1
16   | h => 2
```

In line 1, we first define `NProcessName` as a simple alias for `String`. Process names are what we called roles in ChorLean and are just distinct names for each communication member.

A distinct feature of λ^{χ} is that there are statically predefined procedures which can be used at any time when implementing a λ^{χ} program. Procedures are normal functions but have the special ability to abstract over process names. This allows us to write programs which can dynamically select which process should be used.

Procedures are in fact not values in λ^{χ}. Because of this, we have do define a specific set of procedures beforehand. We declare 3 functions `f`, `g` and `h` in lines 3–6 but could have defined any number of procedures. In lines 8–11 we define the type of each procedure. The type is written using `NType`. For example `g` in line 10 is a function (`func`) which accepts a `String` (wrapped into `NType.arbitrary`) and returns `unit`.

Lean has to statically know how many arguments a procedure expects. For this reason we have to write a helper function `argumentCount` (line 13–16) to statically define the argument count in advance. Lean cannot extract this information by itself. Currently, we have not implemented these procedures but only declared them and gave them static types. We will talk more later about the implementation of procedures and will use `argumentCount` there.

Let us now get to the more interesting main language definitions of λ^χ. A `NProcess` just defines all syntactical elements of λ^χ and their types. Note that a process *name* in the context of λ^χ talks about the role or location of code and a `NProcess` itself represents the expressions:

```
1 inductive NProcess :
2         (processes : List NProcessName) ->
3         (varTypes : HList NType) ->
4         (exprType : NType) ->
5         Type 1
6     where ...
```

Let us first look at the type parameters which are used as a surrounding context for our language. `processes` (line 2) contains the list of all process names which are part of the current expression. We use this to statically infer if a written expression by the programmer is legal with respect to the selected communication partners.

Keen readers could have realised that this is very similar to the concept of `census` in ChorLean but with the nomenclature of λ^χ. In fact, this type parameter (and the following `varTypes` (line 3)) are additions by us and not just implementations of λ^χ. Using Lean to implement λ^χ gave us the unique opportunity to use concepts which were to day only possible in ChorLean and transfer them to this new territory. Our hope is that we can find a unique tradeoff between the simplicity of λ^χ and the guarantees which ChorLean provides. As far as we can tell, we are the first who combined these two paradigms.

`varTypes` (line 3) serves another static guarantee. To make the implementation simpler, we implemented variables using De Bruijn indices. With De Bruijn indices basically every expression type has the current stack of variable types embedded. For example, if you want to access the innermost variable definition you use index 0 and if you want a variable defined higher up on the stack you use a greater index. `varTypes` represents exactly this stack of types. Lean's type system allows us to have this list embedded into `NProcess` using a homogenous list `HList` which we defined previously. Then later we can statically check which value is at a specific index in this list. We will show that when we talk about variables later.

`exprType` (line 4) just defines which type of value this process returns. Line 5 just says that `NProcess` is itself a type which depends on other types.

Let us now look at the possible inductive definitions of `NProcess` and therefore look which kind of expressions λ^χ consists of and their respective structure and accepted sub-expressions:

```
1  inductive NProcess : ... where
2      | NApply :
3        NProcess processes varTypes (NType.func In Out) ->
4        NProcess processes varTypes In  ->
5        NProcess processes varTypes Out
6      | NIf :
7        NProcess processes varTypes (NType.arbitrary Bool) ->
8        NProcess processes varTypes exprType ->
9        NProcess processes varTypes exprType ->
10       NProcess processes varTypes exprType
11     | NOffer :
12       (chooser ∈ processes) ->
13       (chooser : NProcessName) ->
14       ((chosen : NChoiceLabel) -> NProcess processes varTypes exprType) ->
15       NProcess processes varTypes exprType
16     | NChoose :
17       (offerer ∈ processes) ->
18       (offerer : NProcessName) ->
19       (chosen : NChoiceLabel) ->
20       (continuation : NProcess processes varTypes exprType) ->
21       NProcess processes varTypes exprType
22     | NProcedure :
23       (procedure : NProcedureName) ->
24       (substitutes : Vector NProcessName procedure.argumentCount) ->
25       (substitutes.toList ⊆ processes) ->
26       NProcess processes varTypes procedure.type
27     | NValueExpr :
28       NValue processes varTypes exprType ->
29       NProcess processes varTypes exprType
```

Let us start with `NApply` in lines 2–5. `NApply` represents function application and therefore first expects an `NProcess` expression which returns a `func` type. The second sub-expression (line 4) represents the argument to the function and has no specific restriction by itself. But note that we gave the return type of the second sub-expression the `In` which matches the parameter type of the `func` type in line 3. This ensures that both expressions match. Finally, we specify that an expression of type `NApply` is a `NProcess` with the return type `Out` (line 5) which again matches the `func` type.

`NIf` in lines 6–10 is not particularly interesting. It just requires that the first argument is of type `Bool`.

Our next semantic goal is to have some process decide which next step another process should take. For that we need two expression types, i.e. `NOffer` and `NChoose`.

First let us talk about what the semantics of `NOffer` (lines 11–14) are. `NOffer` defines another process - the `chooser` (line 13) - who decides which next step the current process should take. It is also interesting because it is the first expression which requires a proof (line 12). The proof guarantees that you as a programmer cannot select a process which is not part of the current group of processes. This is again an addition by us which was not specified in λ^χ and provides more guarantees and better error reporting. Line 14 defines a function which provides all possible next expressions to execute for any possible `NChoiceLabel`. `NChoiceLabel` is basically just an enum of possible outcomes.

The counterpart is `NChoose`. It defines an `offerer` (line 18) with corresponding proof (line 17). `chosen` is the lable selected by the current executor. Which next expression follows is selected statically with `continuation` (line 20).

Let us define all value types `NValue` which follow similar patterns:

```
1  inductive NValue : (processes : List NProcessName) -> (varTypes : List NType)
       -> (exprType : NType) -> Type 1 where
2      | NConst : (a : α) -> NValue processes varTypes (NType.arbitrary α)
3      | NUnit : NValue processes varTypes NType.unit
4      | NVar : NMember exprType varTypes -> NValue processes varTypes exprType
5      | NFun :
6        {In : NType} ->
7        NProcess processes (In :: varTypes) Out ->
8        NValue processes varTypes (NType.func In Out)
9      | NSend :
10       (receiver : NProcessName) ->
11       (receiver ∈ processes) ->
12       NValue processes varTypes (NType.func Payload NType.unit)
13     | NRecv :
14       (sender : NProcessName) ->
15       (sender ∈ processes) ->
16       NValue processes varTypes (NType.func NType.unit Payload)
```

4.3 MultiChor

`MultiChor`[3] is a choreographic programming language implemented with Haskell, another purel functional programming language. Let us look at this example:

```
1  data Winner = Alpha | Beta | Draw | Fail deriving (Eq, Show, Read)
2
3  choreography :: Choreo '["alpha", "beta"] (CLI m) ()
4  choreography = do
```

```
5   let alpha :: Member "alpha" '["alpha", "beta"] = listedFirst
6       beta  :: Member "beta"  '["alpha", "beta"] = listedSecond
7   a <- locally alpha \_ -> getInput @String "Enter hand:"
8   a' <- (alpha, (singleton @"alpha", a)) ~> beta @@ nobody
9   b <- locally beta \_ -> getInput @String "Enter hand:"
10  b' <- (beta, (singleton @"beta", b)) ~> alpha @@ nobody
11
12  winner <- locally alpha \un ->
13    pure
14      if (un singleton a) == (un singleton b') then "Draw"
15      else if ...
16
17  locally_ alpha \un -> putOutput "Winner:" $ winner
18  locally_ beta \un -> putOutput "Winner:" $ winner
```

In lines 7–8 we read the input of peer `alpha`, save it in `a` and share it to `beta` and saving it in `a'`.

In lines 12–15, `alpha` locally decides the winner. Note that we have to unpack the values with `un` here like in ChorLean. Then we just print the winner locally each at `alpha` and `beta`.

4.4 ChoRus

We also have an implementation with `ChoRus`[14] using Rust but we will leave out further explanation because nothing particularly interesting happens here:

```
impl Choreography for RockPaperScissorsChoreography {
    type L = LocationSet!(PlayerX, PlayerO);
    fn run(self, op: &impl ChoreoOp<Self::L>) -> () {
        let hand_x = op.locally(PlayerX, |un| {
            let mut hand_input = String::new();
            println!("Player X: Enter your hand:");
            std::io::stdin().read_line(&mut hand_input).unwrap();
            return hand_input;
        });
        let mut hand_x_shared = op.broadcast(PlayerX, hand_x);

        let winner = op.locally(PlayerO, |un| {
            let mut hand_input = String::new();
            println!("Player O: Enter your hand:");
            std::io::stdin().read_line(&mut hand_input).unwrap();
```

```
16
17              println!("{}", hand_input);
18              println!("{}", hand_x_shared);
19              println!("{}", hand_input.trim() == "rock");
20              println!("{}", hand_x_shared.trim() == "rock");
21
22              let winner = decideWinner(hand_input, hand_x_shared)
23
24              };
25
26              return winner;
27          });
28          let mut winner_shared = op.broadcast(PlayerO, winner);
29
30          op.locally(PlayerX, |un| {
31              match winner_shared {
32                  Winner::X => println!("You win!"),
33                  Winner::O => println!("You loose!"),
34                  Winner::Draw => println!("Draw!"),
35                  Winner::Fail => println!("Fail!"),
36              };
37          });
38
39          op.locally(PlayerO, |un| {
40              match winner_shared {
41                  Winner::X => println!("You loose!"),
42                  Winner::O => println!("You win!"),
43                  Winner::Draw => println!("Draw!"),
44                  Winner::Fail => println!("Fail!"),
45              };
46          });
47
48      }
49  }
```

5 Conclusion

5.1 Evaluation

Going in at this work we knew that the topics we wanted to research were bleeding edge and use very high-level programming languages and concepts. But the challenges were even higher than we assumed. A big reason for this was that to combine multiple high complexity topics we had to first invest much time into learning the single topics first. After learning about each topic in a vacuum we often learned that combining those two specific topics was either very hard or would take an unreasonable amount of time. Therefore, we had to shift focus multiple times.

For example from combining session types with ChorLean to extending ChorLean with features, to comparing ChorLean with other choreographic programming languages to the mechanization of a full choreographic lambda calculus. In the end we think that by doing this work we learned about a big chunk of the research landscape surrounding distributed programming. Because we learned so much, we hope that readers of this work can also learn about this exciting research subject. We hope that we inspire some readers to either continue with a broad experimental analysis of this subject like we did or take some specific we found out and turn it into a very concentrated research journey.

5.2 Future Work

Future work could include looking further into the differences and similarities between multitier programming languages (e.g. ScalaLoci) and choreographic programming languages (e.g. ChorLean). There are multiple open questions if specific problems are fundamentally unsolvable or if limitations are based on specific implementations. For example, one could try to implement interesting topologies with a choreographic programming

language by adding quantifiers and quantifying over roles which could proof that topology definitions are also possible with choreographic programming languages and not only multitier languages.

Another open topic is exception handling in specifically ChorLean but also generally in choreographic programming languages.

We already started the implementation of λ^{χ} in Lean and finishing this would be the first completed mechanization of λ^{χ}. This also could then be used to translate ChorLean into the framework of λ^{χ} to possible get a trivial deadlock freedom proof or at least aid in such a proof. Using another way to proof deadlock freedom for ChorLean or any other choreographic programming languagewould also be an interesting task.

We already compared ChorLean to multiple other choreographic programming languages but there are naturally many more to find. Maybe one could find inspiration how to improve existing choreographic programming languages by cherry-picking specific features from other languages.

Even though we think that session types are not compatible with choreographic programming languages, one could try to apply session types to multitier programming like ScalaLoci or other distributed programming libraries.

In this work we have shown our experience with the complexity and sometimes even impossibility of some goals. Therefore, further work can learn from our journey by having a better understanding of which research directions are feasible and which are not (for a specific time pensum). For example combining session types and choreographic programming languages seems to us to be impossible or at least unfeasible. Proving deadlock freedom of ChorLean turns out to be a huge task when using usual proof techniques.

5.3 Related Work

We compared multiple choreographic programming languages and tried to improve them by combining knowledge of them. We cared about seeing multiple distinct views of every problem and tried to combine the knowledge of seemingly distinct implementations/research. λ^{χ}[21] also looked at multiple existing choreographic programming languagesto improve them by designing a common denominator.

We compared multiple distinct paradigms like choreographic programming languages, multitier programming and session types. Giallorenzo et al.[9] worked specifically on the pair of multitier programming and choreographic programming languages.

By focusing on using Lean to implement most of our code, for example λ^{χ} we tried to have more secure systems with better static guarantees. For example Thiemann[24] does focus on provable communication guarantees by researching session types on a theoretical level.

Bibliography

[1] pweisenburger et al. *scala-loci*. https://github.com/scala-loci/scala-loci. 2024.

[2] Simon Daniel et al. *Private ChorLean article*. Privately shared access to me. 2025.

[3] Mako Bates, Syed Jafri, and Joseph P. Near. *MultiChor: Census Polymorphic Choreographic Programming with Multiply Located Values*. 2024. arXiv: 2406.13716 [cs.PL]. URL: https://arxiv.org/abs/2406.13716.

[4] Luís Cruz-Filipe et al. "Functional Choreographic Programming". In: *Theoretical Aspects of Computing – ICTAC 2022* (2022). Ed. by Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu. Choreographic programming is an emerging programming paradigm for concurrent and distributed systems, where developers write the communications that should be enacted and a compiler then automatically generates a distributed implementation., pp. 212–237.

[5] Pierre-Evariste Dagand and Frederic Peschanski. "The Lisp in the Cellar". In: *Zenodo* (2025). DOI: 10.5281/zenodo.15424968. URL: https://zenodo.org/records/15424968.

[6] Simon Daniel, van-den-Berg, and Daniel Stricker. *Private ChorLean code repository*. Privately shared access to me. 2025.

[7] Simon Fowler et al. "Exceptional asynchronous session types: session types without tiers". In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019). DOI: 10.1145/3290341. URL: https://doi.org/10.1145/3290341.

[8] Lean FRO. *Dependent de Bruijn Indices*. Jan. 28, 2026. URL: https://lean-lang.org/examples/1900-1-1-dependent-de-bruijn-indices/ (visited on 01/28/2026).

[9] Saverio Giallorenzo et al. “Multiparty Languages: The Choreographic and Multitier Cases”. In: *Leibniz International Proceedings in Informatics (LIPIcs)* 194 (2021). Ed. by Anders Møller and Manu Sridharan. 35th European Conference on Object-Oriented Programming (ECOOP 2021), Keywords: Distributed Programming, Choreographies, Multitier Languages, 22:1–22:27. ISSN: 1868-8969. DOI: 10.4230/LIPIcs.ECOOP.2021.22. URL: https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.ECOOP.2021.22.

[10] Eva Graversen, Fabrizio Montesi, and Marco Peressotti. *A Promising Future: Omission Failures in Choreographic Programming*. 2025. arXiv: 1712.05465 [cs.PL]. URL: https://arxiv.org/abs/1712.05465.

[11] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. “Connectivity Graphs: A Method for Proving Deadlock Freedom Based on Separation Logic (Artifact)”. In: *Zenodo* (2021). DOI: 10.5281/zenodo.5675249. URL: https://zenodo.org/records/5675249.

[12] Jules Jacobs, Stephanie Balzer, and Robbert Krebbers. “Multiparty GV: functional multiparty session types with certified deadlock freedom”. In: *Proc. ACM Program. Lang.* 6.ICFP (Aug. 2022). DOI: 10.1145/3547638. URL: https://doi.org/10.1145/3547638.

[13] Jules Jacobs, Jonas Kastberg Hinrichsen, and Robbert Krebbers. “Deadlock-Free Separation Logic: Linearity Yields Progress for Dependent Higher-Order Message Passing”. In: *Proc. ACM Program. Lang.* 8.POPL (Jan. 2024). DOI: 10.1145/3632889. URL: https://doi.org/10.1145/3632889.

[14] Shun Kashiwa et al. *Portable, Efficient, and Practical Library-Level Choreographic Programming*. 2023. arXiv: 2311.11472 [cs.PL]. URL: https://arxiv.org/abs/2311.11472.

[15] Alex C. Keizer. “Implementing a definitional (co)datatype package in Lean 4, based on quotients of polynomial functors”. In: *ILLC Eprints / Master of Logic Thesis Series* (2023). URL: https://eprints.illc.uva.nl/id/eprint/2239/.

[16] Oleg Kiselyov and Hiromi Ishii. “Freer monads, more extensible effects”. In: *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*. Haskell ’15. Vancouver, BC, Canada: Association for Computing Machinery, 2015, pp. 94–105. ISBN: 9781450338080. DOI: 10.1145/2804302.2804319. URL: https://doi.org/10.1145/2804302.2804319.

[17] lovrosdu and sungshik. *klor*. https://github.com/lovrosdu/klor. 2024.

[18] John McCarthy. “Recursive functions of symbolic expressions and their computation by machine, Part I”. In: *Commun. ACM* 3.4 (Apr. 1960), pp. 184–195. ISSN: 0001-0782. DOI: 10.1145/367177.367199. URL: https://doi.org/10.1145/367177.367199.

[19] C. Mohan, B. Lindsay, and R. Obermarck. “Transaction management in the R* distributed database management system”. In: *ACM Trans. Database Syst.* 11.4 (Dec. 1986), pp. 378–396. ISSN: 0362-5915. DOI: 10.1145/7239.7266. URL: https://doi.org/10.1145/7239.7266.

[20] Andreia Mordido et al. “Parameterized Algebraic Protocols”. In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591277. URL: https://doi.org/10.1145/3591277.

[21] Dan Plyukhin, Xueying Qin, and Fabrizio Montesi. “Relax! The Semilenient Core of Choreographic Programming (Functional Pearl)”. In: *Proc. ACM Program. Lang.* 9.ICFP (Aug. 2025). DOI: 10.1145/3747538. URL: https://doi.org/10.1145/3747538.

[22] Gan Shen, Shun Kashiwa, and Lindsey Kuper. “HasChor: Functional Choreographic Programming for All (Functional Pearl)”. In: *Proceedings of the ACM on Programming Languages* 7.ICFP (Aug. 2023), pp. 541–565. ISSN: 2475-1421. DOI: 10.1145/3607849. URL: http://dx.doi.org/10.1145/3607849.

[23] Daniel Stricker. *master-thesis-daniel-stricker-2026*. https://github.com/AraneusRota/master-thesis-daniel-stricker-2026. 2026.

[24] Peter Thiemann. “Intrinsically Typed Sessions With Callbacks”. In: (2023). arXiv: 2303.01278 [cs.PL]. URL: https://arxiv.org/abs/2303.01278.

[25] Li-yao Xia et al. “Interaction trees: representing recursive and impure programs in Coq”. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019). DOI: 10.1145/3371119. URL: https://doi.org/10.1145/3371119.