Parkour: Parallel Library-Level Choreographic Programming

GAN SHEN, University of California, Santa Cruz, USA
GRANT VANDOMELEN, Portland State University, USA
JONATHAN CASTELLO, University of California, Santa Cruz, USA
YAO LI, Portland State University, USA
LINDSEY KUPER, University of California, Santa Cruz, USA

Choreographic programming (CP) is a paradigm for implementing distributed systems in which one writes a global description of a system (a *choreography*), which is then automatically projected (*endpoint projection*) to a collection of local programs that are guaranteed to run without deadlocks. Existing approaches to CP support sequential compositions of choreographies and rely on various forms of *out-of-order semantics* to express parallel behaviors. These out-of-order mechanisms are often cumbersome to use or insufficiently expressive. In this paper, we introduce *Parkour*, a new CP framework that features a dedicated parallel operator for composing choreographies, enabling first-class support for expressing parallelism while avoiding the limitations and anomalies of out-of-order semantics. We implement Parkour as an extension to HasChor, a Haskell-based library-level CP implementation. This library-based approach lets Parkour choreographies leverage the extensive Haskell ecosystem, as we demonstrate in our case studies. Parkour retains the simplicity and concision of the original HasChor implementation, while also improving on it with a new type-safe implementation of endpoint projection that removes the need for HasChor's unsafe unwrap function.

1 Introduction

Choreographic programming (CP) [Montesi 2013, 2023] is a paradigm for developing distributed software systems in which multiple nodes interact. Rather than programming the nodes independently, in CP the programmer writes a single, unified program, called a *choreography*, which specifies the behaviors of each node and their interactions from a global perspective. A technique known as *endpoint projection* then transforms the global choreography into a set of node-local programs that, when executed together (over some message-passing substrate), implement the specified behavior. CP simplifies the development of distributed systems by consolidating control flows involving more than one participant that would otherwise be fragmented across multiple programs. Moreover, its global perspective ensures that nodes' patterns of message sending and reception are matched up with each other, ensuring *deadlock-freedom by design* [Carbone and Montesi 2013].

In previous work on CP, a choreography is structured as a sequence of commands—what we call a *sequential choreography*. To model parallel behaviors, sequential choreographies then rely on various forms of *out-of-order* semantics to relax the ordering constraints in the source program. Standard out-of-order semantics [Carbone and Montesi 2013; Cruz-Filipe and Montesi 2017, 2020; Cruz-Filipe et al. 2022; Hirsch and Garg 2022; Montesi 2023] allows certain reorderings of commands occurring at different nodes, capturing the inherent parallelism of distributed systems but falling short of admitting parallelism within each node.

To address this limitation, *fully out-of-order* semantics for choreographies [Plyukhin et al. 2024] has been proposed. Fully out-of-order semantics relaxes all ordering constraints and executes commands solely based on the data dependencies evidenced by the textual inputs and outputs of each command in a choreography. As such, fully out-of-order semantics is similar in spirit to classical auto-parallelizing compilers, which look for parallelism opportunities in sequential

Authors' Contact Information: Gan Shen, gshen42@ucsc.edu, University of California, Santa Cruz, USA; Grant VanDomelen, gsv@pdx.edu, Portland State University, USA; Jonathan Castello, jcaste14@ucsc.edu, University of California, Santa Cruz, USA; Yao Li, liyao@pdx.edu, Portland State University, USA; Lindsey Kuper, lkuper@ucsc.edu, University of California, Santa Cruz, USA.

programs via dependence analysis. The idea is to take away the spurious ordering constraints that the programmer introduced by writing sequential code in the first place [Allen and Kennedy 2001].

Two issues with the fully out-of-order approach deserve consideration. First, it expects programmers to express programs sequentially even when they are conceptually parallel, relying on the choreographic language implementation to make up for the language's expressivity limitation. Second, some ordering constraints in sequential choreographies are *not* spurious, yet fully out-of-order semantics relaxes them anyway, leading to unexpected nondeterministic bugs [Plyukhin et al. 2024, §7]. We dive into these two issues in more detail in Section 2.

To address these issues, we propose an alternative approach in which programmers can explicitly say what is parallel and what is not. Rather than modeling a choreography as a sequential program and retrofitting parallelism *implicitly* through out-of-order semantics, we advocate for modeling parallelism directly in the syntax of choreographies, with ordering—or lack thereof—specified explicitly by the programmer. This paradigm, which we call *explicitly parallel choreographic programming*, represents a choreography as a parallel composition of processes running across multiple nodes. We call such a choreography a *parallel choreography*. Explicitly parallel choreographic programming is sufficiently expressive to support parallel behaviors that previously relied on fully out-of-order semantics, while avoiding the anomalies associated with fully out-of-order semantics.

We present Parkour[1], a language for explicitly parallel choreographic programming, implemented as an embedded domain-specific language in Haskell by means of a monadic interpreter (building on a long tradition of embedded monadic DSLs [Hudak 1996]). The key feature of Parkour is a new operator `par` for explicit composition of choreographies in parallel. In the presence of parallel choreographies, it becomes necessary to distinguish messages originating from concurrently executing choreographies at run time. To this end, we extend endpoint projection to assign a unique *session id* to each choreography to be excuted in parallel.

Parkour is implemented as an extension to HasChor [Shen et al. 2023], an existing library-level framework for choreographic programming in Haskell. As such, Parkour's choreographic language constructs and endpoint projection are implemented entirely as a library, with no special compiler or runtime support required, and Parkour choreographies can leverage the host language's extensive ecosystem. For example, programmers can freely use advanced concurrency abstractions, such as those provided by Haskell's support for software transactional memory, in Parkour computations when writing choreographies, as we will see in Section 3.5.3. Furthermore, while explicit parallelism is the default in Parkour, we will see in Section 4 how Parkour choreographies can also support implicit parallelism in the style of Plyukhin et al. [2024]'s Ozone library by way of GHC's `ApplicativeDo` language extension.

Parkour is "mostly backwards-compatible" with HasChor; that is, HasChor programs are legal Parkour programs, modulo some ergonomic improvements to the HasChor choreographic operators that make choreographies more concise (Section 3.3). In addition, we have implemented an alternative approach to library-level endpoint projection that replace HasChor's unsafe, partial `unwrap` function with a safe, total function (Section 6). We achieve this by, essentially, making the runtime representation of a located type dependent on the endpoint to which the type is projected. This approach mostly impacts Parkour internals; but since it does implicate the existing HasChor interface in a backwards-imcompatible way, Parkour does not take this approach by default.

In summary, we make the following contributions:

[1]The name Parkour is short for *par*allel *chor*eography, as well as a nod to the athletic movement practice "in which a Traceur (French for bullet and term for a practitioner of Parkour) moves through their environment as efficiently as possible. In this pursuit for efficient movement, Traceurs re-negotiate obstacles which may slow them down or divert them from an optimised course in un-conventional ways, moving over, through or under them." [Rawlinson and Guaralda 2011]

- We present Parkour, a library-level framework for parallel choreographic programming. Parkour implements a lightweight interface for *explicit* parallelism based on abstractions provided by the host language, Haskell. Parkour is designed to work with the existing features and ecosystem of its host language. We present Parkour's interface, show how parallel programs can be implemented using Parkour, and show how Parkour can interact with the `STM` monad, Haskell's implementation of software transactional memory (Section 3).
- Even though Parkour is based on explicit parallelism, we show we can recover *implicit* parallelism in the setting of Parkour by leveraging the Haskell ecosystem. In particular, we can use GHC's `ApplicativeDo` language extension to obtain Ozone-like behavior when desired (Section 4).
- We show how Parkour can be implemented as an extension to HasChor [Shen et al. 2023] without changing the underlying HasChor implementation by using `CTree`s, a data structure that represents computation trees with both sequential and parallel compositions. A key challenge to implementing Parkour is ensuring noninterfering communications among parallel "forks" at different locations; we address this challenge using a zipper-like datatype for tracking session ids in `CTree`s (Section 5).
- We describe an alternative approach to type-safe library-level endpoint projection that improves on HasChor's design by removing its reliance on an unsafe `unwrap` function. Instead, we make the runtime representation of a located type dependent on the endpoint to which the type is projected (Section 6).

We describe the key ideas of Parkour in Section 2. We discuss related work in Section 7 and conclude in Section 8. The Parkour implementation is available at [link omitted for review; reviewers, please see supplemental material uploaded].

2 Key Ideas

In this section, we describe the key ideas of Parkour. After introducing the basics of choreographic programming (Section 2.1), we illustrate the need for out-of-order semantics to model parallel behaviors in choreographies, as well as the limitations of the out-of-order approach (Section 2.2). Then we motivate the Parkour approach, which uses a dedicated parallel composition operator to model parallel behaviors (Section 2.3). We use the Parkour API in the examples in this section, but defer a more detailed API specification to Section 3.

2.1 Background on Choreographic Programming

In this section, we give a brief introduction to choreographic programming via an example. We refer the reader to Montesi [2023] for a comprehensive introduction.

Consider the distributed data processing pipeline protocol depicted in Figure 1, involving three locations: `alice`, `bob`, and `carol`. The protocol works as follows: input data (of some unspecified type `Data`) is read at `alice` and passed around `alice`, `bob`, and `carol`, with each of them applying their respective processing functions `f`, `g`, and `h` to the data, and the result is output at `carol`. Then, `alice` checks whether there is additional data to process; if so, the protocol recurses, and otherwise it terminates. In traditional distributed programming, the programmer would write independent programs for `alice`, `bob`, and `carol`. Besides obfuscating the control flow of the protocol, this approach is also error-prone, because all three locations must depend on each other to faithfully follow the protocol. If one of them neglects to send a message, for instance, its counterpart will wait forever to receive, leading to a deadlock.

In choregraphic programming [Montesi 2013], the programmer instead writes a single program, called a *choreography*, that expresses the behavior of the entire system from a global point of view.

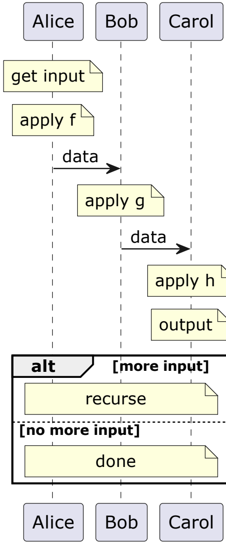

Fig. 1. A distributed data processing pipeline.

```
1  pipeline :: Choreo IO ()
2  pipeline = do
3    x  <- alice `locally` getInput
4    x' <- alice `locally` (f x)
5    y  <- x' ~> bob
6    y' <- bob `locally` (g y)
7    z  <- y' ~> carol
8    z' <- carol `locally` (h z)
9    carol `locally` (output z')
10
11   hasMore <- alice `locally` checkMore
12   cond hasMore \case
13     True  -> pipeline
14     False -> return ()
```

Fig. 2. A choreography for the distributed data processing pipeline

For example, the distributed data processing pipeline protocol might be written as the Parkour choreography in Figure 2.[2] Parkour represents a choreography as a computation in the `Choreo` monad. For readers unfamiliar with monadic programming, the `Choreo` monad is best thought of as a small embedded domain-specific language for expressing choreographic computations within Haskell, where the boundary between DSL code and host-language code is guarded by the Haskell type system. A computation of type `Choreo m a` is parameterized by a local monad `m`, typically `IO` (as is the case in Figure 2 and all other `Choreo` computations shown in this paper), that indicates the kind of local computations that locations can do. Choreographies can also return values, as is typical of functional choreographic programming [Hirsch and Garg 2022]. The `a` type parameter to `Choreo`, indicates the type of value returned by the choreography. The choreography in Figure 2 does not need a meaningful return value, so it simply returns a value of type `()` (the unit type in Haskell).

The `pipeline` choreography showcases three *choreographic operators*, `locally`, `(~>)`, and `cond`, that are available inside `Choreo` computations:

- The `locally` operator lets a location perform a local computation. In Parkour, we often surround `locally` with backticks (`) to use it as an infix operator, so that the location appears on the left and the local computation on the right. The result of a `locally` is a located value at the location that performs the computation, and the type system will ensure only that location can access the value. For example, on line 3 of Figure 2, `alice` runs `getInput` to get the initial input for the pipeline and bind the result to a variable `x`, where `x` is of type `Data @ "alice"`.[3]

[2]This choreography does not use Parkour's `par` operator, so it is also a legal HasChor choreography, modulo a couple of ergonomic improvements that we discuss later on in Section 3.3.

[3]The reader might wonder about the difference between `alice` and `"alice"`. They are essentially the same thing: Parkour represents locations as type-level strings, and `alice` is just a term-level proxy to the type-level string and is of type `Proxy "alice"`. Parkour provides a macro that lets the user create a location, e.g., `$(mkLoc "alice")`, which creates both

- (`~>`), or verbally referred to as the communication (or simply "comm") operation, is the centerpiece of choreographic programming. The expression `x ~> l` denotes that a located value `x` is sent to, and received at, location `l`. The result of `x ~> l` is a new located value of the same type as `x`, but at location `l`. For instance, on line 5 of Figure 2, the `x'` at `alice` with type `Data @ "alice"` is sent to (and received by) `bob`, resulting in a new value `y` located at `bob`, with type `Data @ "bob"`.
- Conditional execution in choreographies require extra attention, as we need to inform locations about which branch to take. This information is known as *knowledge of choice* [Castagna et al. 2011]. Parkour provides the `cond` operation for expressing conditional execution. It takes a located value of type `a @ l` as the value we want to branch on, and a function of type `a -> Choreo IO b` that represents the branches. For instance, on line 11 of Figure 2, the choreography branches on the `hasMore` boolean value at `alice`, and depending on it being `True` or `False`, the choreography either recurses or terminates. For simplicity, conditional execution in Parkour (as with HasChor) does a broadcast to all locations under the hood, which can be inefficient in certain cases as not all locations need to learn about the choice. Previous work has proposed various approaches to deal with this inefficiency issue, such as conclaves [Bates et al. 2025] and static analysis. We defer a full discussion of features for efficient conditional execution and how they interact with parallel choreographies to Section 7.

A choreography can be automatically transformed into a collection of independent *network programs* for each location via *endpoint projection* (EPP). Given a communication operation `x ~> l`, where `x` is of type `a @ l'`, EPP transforms it into a send operation for location `l'` and a receive operation for location `l`. For conditional execution `cond x f`, where `x` is of type `a @ l`, `l` performs a broadcast and then continues with `f`, whereas all other locations perform a receive and then continue with `f`. If EPP is correct, every send or broadcast in the resulting collection of network programs is guaranteed to have a corresponding receive in another network program, ensuring deadlock freedom [Carbone and Montesi 2013].

2.2 The Goldilocks Problem of Out-of-Order Semantics

It is very common for distributed systems to exhibit process-local parallel behavior, such as a server interacting with multiple clients at the same time. We would like to model this kind of behavior using choreographies. Consider a web service protocol involving three locations: `server`, `client1`, and `client2`. The server receives requests from both clients, processes the requests, and sends reponses back to both of them. It might be tempting to model this protocol as the sequential choreography shown in Figure 3. In the `serviceSeq` choreography, we use a derived operator (`~~>`) that combines the `locally` and (`~>`) operators we saw in Section 2.3. The expression `(sender, action) ~~> receiver` indicates that `sender` first performs the local computation `action`, and then sends the result to `receiver`, who receives it, resulting in a located value at the receiver.

Under standard out-of-order choreographic semantics [Carbone and Montesi 2013; Cruz-Filipe and Montesi 2017, 2020; Cruz-Filipe et al. 2022; Hirsch and Garg 2022; Montesi 2023], actions at different locations can be performed out of order. This means that the `comp` actions on line 3 and 4 of Figure 3 can be executed out of order, since they occur at different locations, thereby modeling their parallel execution. However, standard out-of-order semantics mandates that other actions must still be executed in the order in which they appear in the choreography. This prevents the server

the term-level proxy and the type-level string at once. For concision, we have omitted these location declarations from the paper's examples.

```
1  serviceSeq :: Choreo IO ()
2  serviceSeq = do
3    req1 <- (client1, comp) ~~> server
4    req2 <- (client2, comp) ~~> server
5    (server, proc req1) ~~> client1
6    (server, proc req2) ~~> client2
7    return ()
```

Fig. 3. A sequential choreography modeling a web service server

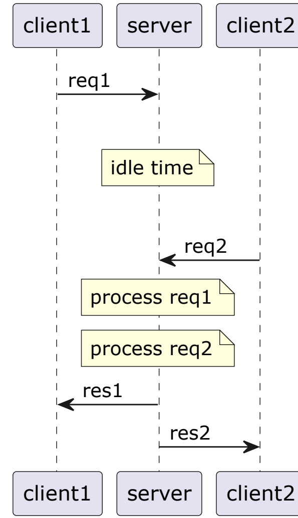

Fig. 4. Demonstration of wasteful waiting in the sequential choreography

from processing the two requests in parallel: if req1 arrives before req2, the server cannot start processing req1 until req2 has arrived, resulting in wasted time. For example, Figure 4 (inspired by a similar diagram in Plyukhin et al. [2024]) depicts an execution of the choreography in which server has to wait for req2 to arrive before it can process req1. Reordering the choreography, for example by swapping lines 4 and 5, does not help either: if req2 happens to arrive first, then it cannot be processed by server until req1 has arrived.

To address this limitation, Plyukhin et al. [2024] proposed *fully* out-of-order semantics for choreographies. Fully out-of-order semantics relaxes all sequential ordering constraints and executes commands in a choreography solely based on their dependencies. While this approach is expressive enough to model the intended parallel behavior, allowing the server to process the two requests in an arbitrary order, it is a double-edged sword: it can also introduce undesired reordering. For instance, say we want to allocate some resource on server for proc to use, and then release the resource after proc is done. Simply adding the highlighted lines to serviceSeq is not sufficient:

```
serviceSeq = do
  ...
  server `locally` allocateResource
  (server, proc req1) ~~> client1
  (server, proc req2) ~~> client2
  server `locally` releaseResource
  ...
```

Because there are no dependencies — or, at least, no dependencies that Plyukhin et al. [2024]'s fully out-of-order semantics has visibility into — between allocateResource, releaseResource, and proc, they may be executed in an arbitrary order under fully out-of-order semantics. This can lead to bugs such as proc accessing a resource that has not been allocated, or the resource being released before it is allocated.

Plyukhin et al. [2024, §7] suggest that this issue might be mitigated by adding support for a hypothetical barrier operation that blocks subsequent operations until preceding ones have finished.

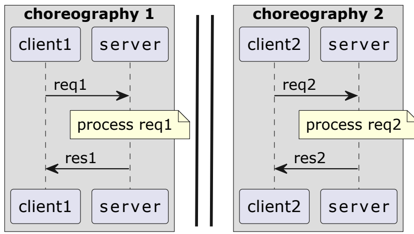

Fig. 5. Sequence diagram for the web service as a parallel choreography

```
1  servicePar :: Choreo IO ()
2  servicePar = do
3    session client1 `par` session client2
4    where
5      session :: Proxy l -> Choreo IO ()
6      session client = do
7      req <- (client, comp) ~~> server
8      (server, proc req) ~~> client1
9      return ()
```

Fig. 6. The web service as a parallel choreography in Parkour

If such a barrier operation existed, we could use it to introduce a synthetic dependency between `allocateResource`, `proc`, and `releaseResource`. However, the need for this kind of workaround shows that out-of-order semantics has a "Goldilocks problem": standard out-of-order semantics does not allow all the reordering we want, while fully out-of-order semantics allows *too much* reordering and would require adding synthetic dependencies to avoid unintended behavior.

One could also argue that this issue could be resolved by baking `allocateResource` and `releaseResource` into `proc`, making it an all-encompassing action. However, this violates the principle of modularity: we can no longer freely compose functionality at function boundaries, but instead must reason about functions' internals.

2.3 Explicitly Parallel Choreographic Programming with Parkour

Since standard out-of-order semantics provides insufficient parallelism, and fully out-of-order semantics requires cumbersome (and heretofore hypothetical) explicit barriers to enforce ordering constraints, we believe an alternative approach is needed. We introduce *explicitly parallel choreographic programming*, a new paradigm for expressing parallel behaviors in choreographies. The key idea is to model parallel behaviors explicitly with a parallel composition operator, rather than retroactively introducing it by giving out-of-order semantics to sequential choreographies. For instance, instead of viewing the web service example from Section 2.2 as a single sequential choreography, we can view it as two choreographies running in parallel, as shown in Figure 5. In each parallel choreography, the server interacts with a single client, and the parallel composition the two choreographies makes up the complete choreography involving the server and both clients.

As our implementation of explicitly parallel choreographic programming, Parkour provides a parallel composition operator `par` for expressing multiple choreographies that run in parallel. An expression of the form ``e1 `par` e2`` indicates that the actions in `e1` and `e2` are executed in parallel. Using `par`, we can rewrite the web service example as a parallel choreography as shown in Figure 6. From line 5 to line 9 of Figure 6, we define the interaction between the server and a client as a subchoreography `session`. On line 2, we then run two instances of `session` in parallel.

Unlike fully out-of-order semantics, Parkour preserves all sequential ordering present in the choreography. For example, if we want to allocate and release resources for `proc`, we can simply add ``server `locally` allocateResource`` and ``server `locally` releaseResource`` before and after `proc` on line 8 of Figure 6, and they will be executed in the order in which they are written in the choreography. Parkour also supports mixing sequential and parallel composition. For example, if we want the `server` to log some run-time information after it has finished interacting with the two clients, we can simply add a line after line 3:

```
servicePar = do
  ...
  session client1 `par` session client2
  server `locally` makeLog
  ...
```

One challenge in implementing the parallel composition operator is how to distinguish, at run time, messages from the same sender in different parallel choreographies. If not properly handled, we might have a receiver receive a message that arises in a different choreography, leading to errors. To address this challenge, we extend the network language (that is, the language of network programs to which Parkour programs are projected) to include a *session id* for each send and receive operation to match them up. We also extend endpoint projection to automatically generate session ids, making it stateful. It is crucial that endpoint projection assigns the same session id to each matching send and receive. To achieve this, we once again exploit the global perspective provided by choreographies — session ids resember the parallel structure of parallel choreographies, and since each endpoint gets the same choreography, matching send and receive are guaranteed to be assigned the same session id. We describe the generation and assignment of session ids in more detail in the next section.

We note that the introduction of the `par` operator makes our programming model inherently nondeterministic. Parkour itself neither provides abstractions to ensure determinism, nor does it prevent the use of them. Instead, it is the programmer's responsibility to properly use appropriate synchronization mechanisms to coordinate parallel executing choreographies. Haskell provides a number of such abstractions, such as `MVar`s [Marlow 2011] and software transactional memory [Harris et al. 2005]. We provide an example of using the latter in a Parkour choreography in Section 3.5.3.

Fully out-of-order choreographies are also nondeterministic, as Plyukhin et al. [2024, §7] discuss. However, nondeterminism in a Parkour choreography is *always* the result of a use of `par` in the source program, whereas nondeterminism in Plyukhin et al.'s fully out-of-order choreographies is a result of reordering carried out by the language implementation and is therefore (we argue) not as obvious or predictable to the programmer.

3 The Parkour API

In this section, we present the user-facing API of Parkour. Parkour builds on previous work on HasChor [Shen et al. 2023] and largely preserves its API, while making some ergonomic changes. One key highlight of the Parkour API is that we do not need any additional interface to support explicit parallelism. Instead, we reuse the <*> operator provided by the `Applicative` typeclass, a typeclass defined in Haskell standard library. However, Parkour still includes an operator for explicit parallelism, namely `par`, for convenience, but the operator is not essential—it is defined using <*>.

3.1 Locations

Choreographies abstract nodes in a distributed system as locations. Locations are used both at the type and term levels. We need locations at both levels because Haskell is not a fully dependently typed language. Parkour defines `LocTy` and `LocTm` for the type level and term level, respectively:

```
type LocTy = Symbol
type LocTm = String
```

At the term level, a location is just a `String`. At the type level, a location is a type-level string, namely a `Symbol`. When we need to provide a type-level location at the term level, we use the

`Proxy` data type, whose type `Proxy l` carries a phantom type `l` representing the type-level location. For example, the `client1` used in Figure 6 is defined as

```
client1 :: Proxy "client1"
client1 = Proxy
```

We use the `KnownSymbol` type class for converting a type-level string to a term-level one, and require this constraint for all uses of type-level locations.

3.2 Located Values

A *located value* `a @ l` is a value of type `a` located at location `l`, where `l` is of type `LocTy`. To prevent users from using a located value at a location that does not own the value, we keep located values opaque to the user. The opacity also prevents users from writing choreographies that depend on located values, ensuring all locations have a consistent view of the choreography. To use a located value, the user needs to *unwrap* it using an unwrap function of type `Unwrap l`.

```
type Unwrap l = forall a. a @ l -> a
```

Crucially, this function of type `Unwrap l` is only accessible when performing a local action at `l`, ensuring safe access to located values.

3.3 The Choreo Monad

The `Choreo` monad is the centerpiece of Parkour for writing choreographies.

```
type Choreo m a
instance Functor (Choreo m)
instance Applicative (Choreo m)
instance Monad (Choreo m)
```

As in HasChor, `Choreo` is an instance of `Monad`, which gives us the (>>=) operator (pronounced "bind") for writing sequential choreographies, as well as the convenient `do`-notation that we use for all our examples.

```
(>>=) :: (Monad m) => m a -> (a -> m b) -> m b
```

Unlike in HasChor, though, where the `Applicative` instance for `Choreo` is shallowly derived from `Monad`, Parkour uses the <*> operator of `Applicative` to express parallel execution of choreographies.

```
<*> :: (Applicative m) => m (a -> b) -> m a -> m b
```

The <*> operator (which we pronounce "apply") runs two computations and applies the result of the first to the second. Since there are no dependencies between the two computations at the type level, we can exploit parallelism and run the two computations in parallel. For situations where the results of choreographies are not needed and the choreographies are being executed only for their side effects, we provide the simpler operator `par` that ignores the results and returns the unit value `()`. While the user-facing API includes <*>, typical choreographies such as `servicePar` from Section 2, as well as several other examples we will see later on in Section 3.5, need only use `par` rather than <*>.

```
par :: (Applicative m) => m a -> m b -> m ()
par a b = pure (\_ _ -> ()) <*> a <*> b
```

The `locally`, (~>), and `cond` operations in the `Choreo` monad, which we saw earlier in Section 2, are mostly the same as in HasChor, except for some ergonomic changes that Parkour makes to (~>) and `cond`. In particular, they no longer need to take the initiator location (that is, the sender of a (~>) operation, or the location at which the condition of a `cond` operation is evaluated), since that location can be inferred from the type `a @ l` of the located value argument to both operations.

```
locally :: (KnownSymbol l) =>
           Proxy l -> (Unwrap l -> m a) -> Choreo m (a @ l)
(~>)    :: (Show a, Read a, KnownSymbol l, KnownSymbol l') =>
           a @ l -> Proxy l' -> Choreo m (a @ l')
cond    :: (Show a, Read a, KnownSymbol l) =>
           a @ l -> (a -> Choreo m b) -> Choreo m b
```

For example, in Parkour we can simply write `x ~> server` instead of `(client, x) ~> server` as one would need to do in HasChor. This change makes choreographies more concise. The `locally` operation provides an `Unwrap l` function to the user, which can unwrap located values at `l`. For example, the `proc` function in Figure 3 would use the `Unwrap l` to unwrap the request at server:

```
proc :: (Request @ "server") -> Unwrap "server" -> IO Response
proc req un = procPrimitive (un req)
```

3.4 Projecting and Running a Choreography

Given a choreography written in the `Choreo` monad, we can project it to a given location and run the resulting network program using `runChoreography`.

```
runChoreography :: (Backend config) => config -> Choreo IO a -> LocTm -> IO a
```

The `runChoreography` function takes a configuration for a particular network transport backend, which specifies how network programs handle message passing. Parkour provides a built-in HTTP backend that sends and receives HTTP messages. For example, we can run the `server` of the file server choreography in Section 2 with the following call to `runChoreography`:

```
let cfg = mkHttpConfig
  [ ("server",  ("localhost", 4242))
  , ("client1", ("localhost", 4343))
  , ("client2", ("localhost", 4444))
  ]
in runChoreography cfg fileServer server
```

The HTTP backend configuration being supplied here as the first argument to `runChoreography` is an association list that maps each location to a host name and port number. We defer detailed implementation of backend to Section 5.

3.5 Parkour in Action

We illustrate the use of the Parkour API with several examples. First, we reimplement two example programs from Plyukhin et al. [2024] in Parkour to show how Parkour's dedicated parallel operator compares with fully out-of-order semantics (Sections 3.5.1 and 3.5.2). We then present a quorum voting example that combines Parkour's parallel operator with local concurrency, using Haskell's support for software transactional memory (STM) (Section 3.5.3).

3.5.1 Microservices Example. Consider the microservices example from Plyukhin et al. [2024] depicted in Figure 7. A content service `cs` sends a value `txt` to the server `s`, while the key service `ks` sends a value `key`. The server then forwards both values to a client `c`, which displays the text and decrypts the key. Because the two sessions are independent, we would like the server to handle `txt` and `key` independently, regardless of their arrival order. Under fully out-of-order semantics, the choreography in Figure 8 has this behavior despite appearing to be sequential: instructions 1, 3, and 5 may run independently of instructions 2, 4, and 6, subject only to the dependencies within each chain. This formulation is concise, but it makes additional ordering constraints awkward to express. For example, suppose we want the server to log after forwarding each value to the client.

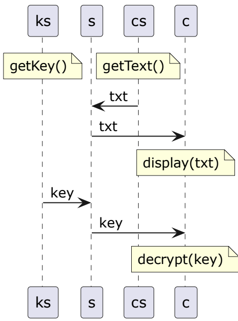

Fig. 7. The microservices example shown in Figure 2 of Plyukhin et al. [2024].

1. cs.$getText()$ ⟶ val s.txt;
2. ks.$getKey()$ ⟶ val s.key;
3. s.txt ⟶ val c.txt;
4. s.key ⟶ val c.key;
5. c.$display(txt)$;
6. c.$decrypt(key)$;

Fig. 8. Choreography for the microservices example shown in Figure 2 of Plyukhin et al. [2024]. Although the choreography appears to be sequential, under fully out-of-order execution, lines 1, 3, and 5 can run independently of lines 2, 4, and 6.

```
1  decryptDisplay :: Choreo IO ()
2  decryptDisplay = contentSession `par` keySession
3    where
4      contentSession :: Choreo IO ()
5      contentSession = do
6        stxt <- (contentService, \_ -> getText) ~~> server
7        ctxt <- stxt ~> client
8        client `locally` (\un -> display (un ctxt))
9        return ()
10
11     keySession :: Choreo IO ()
12     keySession = do
13       skey <- (keyService, \_ -> getKey) ~~> server
14       ckey <- skey ~> client
15       client `locally` (\un -> decrypt (un ckey))
16       return ()
```

Fig. 9. A Parkour implementation of the microservices example as two distinct choreographies that are explicitly composed in parallel with par.

Adding s.$log()$ after lines 3 and 4 would not be sufficient: without an explicit dependency, fully out-of-order semantics may reorder each log action before the communication it is meant to follow. In Parkour, we instead write the two independent sessions as an explicit parallel composition, as shown in Figure 9. The choreography is decomposed into contentSession and keySession, which run in parallel, while each session remains internally sequential. Consequently, a server-side log can be inserted after line 7 or line 14, and Parkour preserves the intended ordering within the corresponding session.

3.5.2 Resource Transformation Example. As another example from Plyukhin et al. [2024], consider a resource pipeline with a producer p, two transformers r1 and r2, and a processor q. The producer creates a resource and sends it to a transformer. The transformer sends the transformed value to the processor. Finally, the processor sends the result back to the producer, which stores it together with the original value. In Figure 10 (reproduced from [Plyukhin et al. 2024]), the procedure X describes

1. $X(\mathrm{a}, \mathrm{b}, \mathrm{c}) =$
2. val a.w = *produce*();
3. a.$w \longrightarrow$ val b.x;
4. b.*transform*(x) $\longrightarrow$ val c.y;
5. c.*process*(y) $\longrightarrow$ val a.z;
6. a.*store*(w, z)
7. $X(\mathrm{p}, \mathrm{r}_1, \mathrm{q})$;
8. $X(\mathrm{p}, \mathrm{r}_2, \mathrm{q})$

Fig. 10. The resource transformation example shown in Figure 5 of Plyukhin et al. [2024]. Under fully out-of-order execution, the two calls to X can execute in parallel despite appearing in sequential order in the choreography.

```
1  twiceX :: Choreo IO ()
2  twiceX = do
3    x p r1 q `par` x p r2 q
4    where
5    x :: (KnownSymbol a, KnownSymbol b,
6          KnownSymbol c)
7          => Proxy a -> Proxy b -> Proxy c
8          -> Choreo IO ()
9      x a b c = do
10       aw <- a `locally` (\_ -> produce)
11       bx <- aw ~> b
12       cy <- (b, \un -> transform (un bx)) ~~> c
13       az <- (c, \un -> process (un cy)) ~~> a
14       a `locally` (\un -> store (un aw) (un az))
15       return ()
```

Fig. 11. A Parkour implementation of the resource transformation choreography. The two calls to `x` are explicitly composed in parallel with `par`.

this pipeline for one resource. Under Plyukhin et al. [2024]'s fully out-of-order semantics, the two sequential calls to X at the end can execute in parallel, modeling two resources being processed at the same time. However, the same limitation as with the microservices example arises when we need to express a necessary sequencing constraint. For instance, if we want to execute a log operation after both calls to X have completed, simply adding a line after line 8 does not impose that order: the log operation may be reordered with the two calls to X. Parkour expresses the same behavior directly, as shown in Figure 11. The helper `x` abstracts the pipeline over its participating locations, and `twiceX` composes two calls to `x` in parallel. Because the parallel composition itself is sequenced in the surrounding choreography, a log operation inserted after line 3 is guaranteed to run only after both parallel pipelines have completed.

3.5.3 Quorum Voting Example. As a more involved example of programming with Parkour, consider the quorum voting protocol in Figure 12. The `ballotBox` process collects yes and no votes from three voters and reports the majority result. Because a majority is reached after two votes for either outcome, `ballotBox` can report the result as soon as the second yes vote or the second no vote arrives. The vote-casting choreographies and the two counting choreographies all run in parallel, as made explicit by the uses of `par`. The shared variables `yesVotes` and `noVotes`, both located at the ballot box, record how many votes of each kind have arrived. Each `castVote` sub-choreography asks one voter for a decision, sends that decision to the ballot box, and then atomically increments the corresponding counter. The `countYesVotes` and `countNoVotes` sub-choreographies monitor those counters and report the first result whose counter reaches the quorum threshold.

The quorum voting protocol combines choreographic communication with local concurrency on a single endpoint. Parkour ensures that the voters and ballot box communicate according to the choreography. Meanwhile, on the ballot box process, the choreography makes use of Haskell's `stm` library, which provides the `TVar` abstraction. A `TVar` is a shared memory location that supports atomic memory transactions. Here, we use `TVar`s to synchronize the ballot box's local threads as they share and update the vote counters. The `TVar`s provide shared state with atomic access

```
quorumVoting :: Choreo IO ()
quorumVoting = do
  yesVotes <- ballotBox `locally` (\_ -> newTVarIO 0)
  noVotes <- ballotBox `locally` (\_ -> newTVarIO 0)
  castVote voter1 yesVotes noVotes `par`
    castVote voter2 yesVotes noVotes `par`
      castVote voter3 yesVotes noVotes `par`
        countYesVotes yesVotes `par`
          countNoVotes noVotes
  where
    castVote :: (KnownSymbol l) => Proxy l ->
      TVar Int @ "ballotBox" -> TVar Int @ "ballotBox" ->
      Choreo IO (() @ "ballotBox")
    castVote voter yesVotes noVotes = do
      vote <- (voter, \_ -> getVote) ~~> ballotBox
      ballotBox `locally` \un -> do
        if un vote then
          atomically $ modifyTVar (un yesVotes) (+1)
        else
          atomically $ modifyTVar (un noVotes) (+1)

    getVote :: IO Bool
    getVote = read <$> getLine

    countYesVotes :: TVar Int @ "ballotBox" -> Choreo IO ()
    countYesVotes yesVotes =
      void $ ballotBox `locally` \un -> do
        atomically (do x <- readTVar (un yesVotes); check (x >= 2))
        putStrLn "Yes wins"

    countNoVotes :: TVar Int @ "ballotBox" -> Choreo IO ()
    countNoVotes noVotes =
      void $ ballotBox `locally` \un -> do
        atomically (do x <- readTVar (un noVotes); check (x >= 2))
        putStrLn "No wins"
```

Fig. 12. Quorum voting. The `ballotBox` reports the result as soon as it receives enough yes or no votes.

through `atomically`. The `check` operations provide efficient blocking: a counting thread waits until the relevant **`TVar`** changes and the quorum condition can be satisfied, rather than polling the counter.

This example illustrates how Parkour choreographies can make use of existing concurrency abstractions in the host language, freely mixing them into choreographic code. Of course, a **`TVar`** cannot be used for synchronization *across* processes, and an attempt to use one as such in a **`Choreo`** computation would be ruled out by the type system. For instance, if `voter1` attempted to directly modify one of the **`TVar`**s located at `ballotBox`, the code would be rejected as ill-typed, because only `ballotBox` can unwrap and access those **`TVar`**s.

4 Recovering Implicit Parallelism in Parkour with ApplicativeDo

Explicit parallelism with par, as illustrated in Section 3.5, is the default way of achieving parallelism in Parkour. However, there are situations in which implicit parallelism in the style of Ozone—Plyukhin et al. [2024]'s implementation of fully out-of-order semantics—is more desirable to the programmer. In the implicit style, choreographies are written in an apparently-sequential style, but actions can be executed in parallel during runtime.

In this section, we show that we can implement Ozone-like implicit parallelism in Parkour easily, by taking advantage of the Haskell ecosystem—in particular, the Glasgow Haskell Compiler (GHC)'s **ApplicativeDo** extension [Marlow et al. 2016].

When enabled, **ApplicativeDo** automatically desugars **do**-notation into **Applicative** operations when possible [Marlow et al. 2016]. Since Parkour uses the <*> operation of **Applicative** to represent parallel composition, we can simply write a sequential choreography using **do**-notation and then let **ApplicativeDo** desugar it using parallel composition <*>. This results in Ozone-like behavior. However, the **ApplicativeDo** extension is conservative as it does not change orders of effectful operations, so we cannot always get the fully out-of-order execution as Ozone does.

This implicit style based on **ApplicativeDo** suffers from the same limitation as Ozone: it can wrongfully parallelize computations with shared internal state. Fortunately, **ApplicativeDo** is opt-in. It can be used (or not) on a per-module basis, allowing a user to decide whether to use explicit or implicit parallelism according to the characteristics of each module in their system.

In this section, we first illustrate how **ApplicativeDo** works by rewriting the quorum voting examples of Figure 12 to an implicit style (Sections 4.1 and 4.2). We then show the conservativeness of **ApplicativeDo** via an example based on web services. We show how this conservativeness impacts parallelism in an implicit-style chorepgraphy, and how we can achieve a more Ozone-faithful execution via manually reordering certain computations (Section 4.3). Finally, we show how the implicit style based on **ApplicativeDo** suffers the same issues as Ozone, which motivates our explicit-by-default design (Section 4.4).

4.1 A Primer on ApplicativeDo

The key idea behind **ApplicativeDo** is that, when we have a computation whose arguments do not depend on the result of a prior computation, we can use the applicative <*> instead of the monadic (>>=). Consider the following example given by Marlow et al. [2016]:

```
1 mapM []     = pure []
2 mapM (x:xs) = do x'  <- f x
3                  xs' <- mapM f xs
4                  pure (x' : xs')
```

In the definition of mapM, the code on line 2 and line 3 share no data dependency in their function arguments. This means that we can rewrite mapM as follows:

```
mapM :: Applicative m => (a -> m b) -> [a] -> m [b]
mapM []     = pure []
mapM (x:xs) = (:) <$> f x <*> mapM f xs
```

This translation from the prior definition based on the **do**-notation to the second definition is exactly what **ApplicativeDo** does—it automatically analyzes data dependencies in function arguments and translates using <*> when applicable.

ApplicativeDo also works on more complicated cases. For example, it can desugar in a way that uses both <*> and (>>=) in different places inside the same function. In the cases that there are more than one possible translations, it automatically chooses the "best one" based on a cost

model. Interested readers can find more detailed description of the design and implementation of **ApplicativeDo** in Marlow et al. [2016].

4.2 Quorum Voting Revisited

Recall our quorum voting example in Figure 12. In this example, the code on line 3 and line 4 do not depend on each other. In addition, code on lines 5–9 depend on lines 3–4, but not on each other. This is exactly the kind of dependency that can be automatically analyzed by **ApplicativeDo**. Indeed, we can rewrite the quorum voting example as follows by enabling **ApplicativeDo**:

```
quorumVoting :: Choreo IO ()
quorumVoting = do
  yesVotes <- ballotBox `locally` (\_ -> newTVarIO 0)
  noVotes  <- ballotBox `locally` (\_ -> newTVarIO 0)
  castVote voter1 yesVotes noVotes
  castVote voter2 yesVotes noVotes
  castVote voter3 yesVotes noVotes
  countYesVotes yesVotes
  countNoVotes  noVotes
  return ()
  where
    ...
```

After desugaring, the above code is translated as follows (we simplified the code for presentation purposes):

```
quorumVoting :: Choreo IO ()
quorumVoting =
  join (
    (\ yesVotes noVotes ->
      (\ _ _ _ _ _ -> ())
      <$> (castVote voter1 yesVotes noVotes)       -- A
      <*> (castVote voter2 yesVotes noVotes)       -- B
      <*> (castVote voter3 yesVotes noVotes)       -- C
      <*> (countYesVotes yesVotes)                 -- D
      <*> (countNoVotes noVotes))                  -- E
    <$> (locally ballotBox (\ _ -> newTVarIO 0))   -- F
    <*> (locally ballotBox (\ _ -> newTVarIO 0)))  -- G
  where
    ...
```

In this definition, we first run lines 11 and 12 in parallel, thanks to `<*>`. The result of executing these lines are then fed into lines 4–10 as arguments `yesVotes` and `noVotes`. Note that code on lines 4–10 appear before lines 11–12, but is executed after lines 11–12 due to the function argument dependency and the `join` operation. After `join`ing, we run computations on lines 6–10 in parallel.

If we represent all computations on lines 6–12 using characters A to G in the comments, the execution order of the above program is `(F | G) ; (A | B | C | D | E)`, where `;` represents a sequential composition and `|` represents a parallel composition. If we implement quorum voting in Ozone, we will get the same order in the execution. Therefore, this implicit-style program in Parkour resembles fully out-of-order semantics.

4.3 ApplicativeDo is Conservative

The **ApplicativeDo** extension is implemented for general effects, so it must be conservative—in particular, it never tries to swap the order of operations. This means that the implicit style based on **ApplicativeDo** can result in less parallel execution than Ozone. For example, if we rewrite the web service example of Fig. 6 to use **do**-notation:

```
servicePar :: Choreo IO ()
servicePar = do
  req1 <- (client1, comp) ~~> server  -- A
  req2 <- (client2, comp) ~~> server  -- B
  (server, proc req1) ~~> client1     -- C
  (server, proc req2) ~~> client2     -- D
  return ()
```

When **ApplicativeDo** is enabled, the code will be desugared to exhibit the following runtime behavior: `(A | B) ; (C | D)`. Although this execution makes sufficient use of parallelism, it is suboptimal. This is because the arguments of computation `C` only rely on computation `A` (i.e., `req1`), and the arguments of computation `D` only rely on computation `B` (i.e., `req2`). Therefore, it is wasteful for `C` to wait for `B` or for `D` to wait for `A`.

A more parallel runtime behavior is `(A ; C) | (B ; D)`. However, **ApplicativeDo** intentionally avoids this translation because such a translation needs to move `C` before `B`. This is undesirable for general effectful computations because swapping orders can alter the semantics. In our `servicePar`, swapping `B` and `C` is fine, but there is no way for **ApplicativeDo** to know that. On the other hand, Ozone is designed only for choreographic programming, so it easily assumes that `B` and `C` can be reordered, leading to a more parallel execution. In Parkour, we need to instead manually swap the order inside the **do**-notation of `servicePar`:

```
servicePar :: Choreo IO ()
servicePar = do
  req1 <- (client1, comp) ~~> server
  (server, proc req1) ~~> client1
  req2 <- (client2, comp) ~~> server
  (server, proc req2) ~~> client2
  return ()
```

In Marlow et al. [2016], the authors propose that allowing reordering *commutative* monads in **ApplicativeDo** can be possible future work. If such a feature is implemented, we could potentially achieve implicit parallelism in Parkour that matches the behavior of Ozone.

4.4 Implicit Parallelism is Not Always Good

As we discussed in Section 2.2, the assumption that any computations without a data dependency can be reordered can lead to unexpected behavior in Ozone. This is just as true when using **ApplicativeDo** in Parkour. For example, consider operations for reading a file. The operations **Open**, **Read**, and **Close** must occur in sequential order for a program to be correct, so we write it sequentially in **do**-notation:

```
choreo :: Choreo FileReading (String @ "alice")
choreo = do
  alice `locally` (\un -> Open)
  s <- alice `locally` (\un -> Read)
  alice `locally` (\un -> Close)
  return s
```

However, there are no data dependencies among code in lines 3–5. If ApplicativeDo is enabled, the above code will be desugared into:

```
choreo = (\ _ s _ -> s)
    <$> (locally alice (\ _ -> Open))
    <*> (locally alice (\ _ -> Read))
    <*> (locally alice (\ _ -> Close))
```

The desugared code has all three operations being composed in parallel! Therefore, in this case, using implicit-style parallelism is inappropriate. This is the same issue that Ozone suffers from.

Fortunately, Parkour is designed to use explicit parallism by default—ApplicativeDo is opt-in. This means that we can resolve this issue by simply *not* enabling ApplicativeDo. Currently, the ApplicativeDo extension is enabled per module, so we need to either not enable ApplicativeDo in choreo's module, or move choreo to a different module.

5 Implementation

In this section, we discuss the implementation details of Parkour. First, we introduce CTree, a data structure that represents computations with both sequential and parallel compositions (Section 5.1). CTree forms the backbone of both the Choreo monad, which represents choreographies (and which we have already seen throughout the paper), and the Network monad, which represents network programs that run at a particular location. After that, we talk about how located values are represented in Parkour (Section 5.2). Next, we look at the implementation of the Choreo and Network monads (Section 5.3 and Section 5.4). Then, discuss endpoint projection, the hallmark of choreographic programming, which turns a choreography into a set of network programs by linking up Choreo and Network (Section 5.5). Finally, we briefly discuss our HTTP backend for network programs (Section 5.6).

As an extension to HasChor, Parkour's implementation is heavily influenced by HasChor. Our focus here will be on the additions that Parkour makes to HasChor. For an introduction to the idea of implementing library-level choreographic programming via an embedded monadic DSL, we refer the reader to Shen et al. [2023].

5.1 CTree

The CTree data structure represents computation trees with both sequential and parallel compositions. We name the data structure CTree because it resembles ctree_total defined by Swamy et al. [2020] (we defer a more detailed comparison to Section 7). Both the Choreo and Network monads are defined in terms of the CTree data structure.

```
data CTree f a where
  Perf :: f a -> CTree f a
  Pure :: a -> CTree f a
  App  :: CTree f (a -> b) -> CTree f a -> CTree f b
  Bind :: CTree f a -> (a -> CTree f b) -> CTree f b
```

A CTree f a describes a computation that performs effects in f and returns a value of type a. f :: Type -> Type is called the effect signature. An f a describes an effect that returns a value of type a. Perf performs an effect in f. Pure lifts a pure value into a CTree. App runs two CTrees in parallel and applies the result of the first to the second. Bind runs a computation first, then passes its result to a continuation that describes the subsequent computation.

CTree can be viewed as an extension of the freer monad used in HasChor, with the additional constructor App representing parallel computations. Alternatively, CTree can be viewed as a reification of Haskell's Applicative and Monad type classes, with each of its constructors corresponding

to a method of one of these classes. To make this connection explicit, we define the following straightforward instances so that we can use Haskell's **do**-notation to construct `CTree` programs.

```
instance Functor (CTree f) where
  fmap f t = Bind t (Pure . f)

instance Applicative (CTree f) where
  pure = Pure
  (<*>) = App

instance Monad (CTree f) where
  (>>=) = Bind
```

`CTree` only defines the syntax of computations, as its constructors merely record what is to happen, rather than performing the computation directly. The semantics of `CTree` is given by an interpretation function `interp`.

```
interp :: (Applicative m, Monad m) => (forall a. f a -> m a) -> CTree f a -> m a
interp hdl (Perf e)   = hdl e
interp hdl (Pure a)   = pure a
interp hdl (App f a)  = interp hdl f <*> interp hdl a
interp hdl (Bind t k) = interp hdl t >>= (interp hdl . k)
```

Given a domain `m` that implements `Applicative` and `Monad`, `interp` takes an effect handler that interperts the effects in `f` within `m`, and maps a `CTree` into `m`. The definition of `interp` mostly just uses the corresponding type class methods of `m` to interpret each constructor of `CTree`. Although `Applicative` is a superclass of `Monad` in Haskell, we list it explicitly as a type class constraint in the signature of `interp` to emphasize that `m` is expected to provide an ad hoc `Applicative` implementation that exploits parallelism, rather than relying on the default implementation derived from `Monad`.

5.2 Located Values

Parkour represents located values internally using the following type, which is essentially an option type (or a `Maybe` type, as it is known to Haskellers):

```
data a @ l = Wrap a | Empty
```

At location `l`, a value of type `a @ l` contains a value of type `a`. At all other locations, it is empty. Internally, Parkour uses the `wrap` and `unwrap` functions to create and extract located values.

```
wrap :: a -> a @ l
wrap = Wrap

unwrap :: a @ l -> a
unwrap (Wrap a) = a
unwrap Empty    = error " Internal Error: unwrapping an empty located value."
```

Note that the `unwrap` function is partial: it throws an error if applied to an empty located values. In the Parkour implementation, we maintain the invariant that `unwrap` is only ever applied to non-empty located values, so this partiality is not an issue. In Section 6, we present a type-safe implementation that statically guarantees this invariant.

5.3 The Choreo Monad

The `Choreo` monad is defined as a `CTree` instantiated with the effect signature `ChoreoSig`. The three constructors of `ChoreoSig`—`Locally`, `Comm`, and `Cond`—correspond respectively to the three

user-facing operations locally, (~>), and cond that we discussed earlier in Section 3.3. Absent from this list is par, whose implementation is of a different nature from the other user-facing operations: it is available for use in Choreo computations since Choreo is a CTree (and hence an instance of Applicative).

```
data ChoreoSig m a where
  Locally :: (KnownSymbol l) =>
    Proxy l -> (Unwrap l -> m a) -> ChoreoSig m (a @ l)
  Comm :: (Show a, Read a, KnownSymbol l, KnownSymbol l') =>
    Proxy l -> a @ l -> Proxy l' -> ChoreoSig m (a @ l')
  Cond :: (Show a, Read a, KnownSymbol l) =>
    Proxy l -> a @ l -> (a -> Choreo m b) -> ChoreoSig m b

type Choreo m a = CTree (ChoreoSig m) a
```

5.4 The Network Monad

Network programs are the programs run at individual locations and the result of endpoint projection. In Parkour, they are expressed as computations in the Network monad, and defined as CTree instantiated with the effect signature NetworkSig.

```
data NetworkSig m a where
  Exec :: m a -> NetworkSig m a
  Send :: Show a => SessionId -> a -> LocTm -> NetworkSig m ()
  Recv :: Read a => SessionId -> LocTm -> NetworkSig m a
  BCast :: Show a => SessionId -> a -> NetworkSig m ()

type Network m = CTree (NetworkSig m)
```

The Network monad provides four operations for use in endpoint projection: Exec, which executes a local computation; Send, which sends a message to a location; Recv, which receives a message from a location; and BCast, which broadcasts a message to all locations and is used to implement conditionals. For simplicity, Parkour uses Haskell's Show and Read for message serialization and deserialization.

The SessionIds used here require some explanation: they are used to distinguish messages from the same sender in parallel choreographies and pair up sends and receives. During choreography execution, each receiver can uniquely identify a message by the pair of sender name and session ID. For example, if we run the following two network programs together:

```
aliceN :: Network IO ()
aliceN = (send sid1 bob 42) `par` (send sid2 bob "hello, world")

bobN :: Network IO ()
bobN = (do x <- recv sid1 alice; print x) `par` (do y <- recv sid2 alice; print y)
```

then x will receive 42 and y will receive "hello, world". We discuss how session IDs are defined and generated in the next section.

The above example also illustrates how par appears in Network computations. Just as with Choreo, par is not part of the effect signature, but is available in Network computations because Network is a CTree.

5.5 Endpoint Projection

5.5.1 Session IDs. With parallelism introduced by the `App` constructor of `CTree`, it becomes challenging to distinguish messages from the same sender. Session IDs are conceived to resolve this issue. Their definition reflects the structure of `App` and can be viewed as an index into the tree made out of nested `App`.

```
data SessionId where
  Root      :: SessionId
  NestLeft  :: SessionId -> SessionId
  NestRight :: SessionId -> SessionId
```

We call a choreography running in parallel with others a *session*. `SessionId`s are assigned in the following way:

- The top-most choreography gets ID `Root`.
- For the left child `c1` of `App c1 c2`, we assign `NestLeft sid`, where `sid` is the session ID of the current choreography.
- Similarly, for the right child `c2`, we assign `NestRight sid`.

This strategy for assigning session IDs ensures that each sub-choreography gets a session ID distinct from its parent's, thereby guaranteeing the local uniqueness of session IDs in arbitrarily nested parallel choreographies. For parallel choreographies across `Bind`s, their session IDs may overlap. However, since `Bind` enforces sequential ordering, at any given point during execution, there is still at most one choreography with a given session ID. Plyukhin et al. [2024] uses a similar strategy, called *integrity keys*, to distinguish messages sent out-of-order. Unlike their approach, which is based on static information such as line numbers that may not be available in certain cases, ours is more dynamic and can readily support recursive choreographies without requiring any extension.

5.5.2 Endpoint Projection Monad. We want to use the `interp` function discussed earlier in Section 5.1 to interpret a `Choreo` monad, which requires an appropriate domain `m`. During endpoint projection, we also need to keep track of the current session ID and update it when entering the sub-choreographies in `<*>`. To this end, we wrap `Network` in a `ReaderT` monad transformer with context `SessionId`, and use it as the domain for `interp`, calling it `Epp`.

```
newtype Epp m a = Epp { unEpp :: ReaderT SessionId (Network m) a }
  deriving (Functor, Monad, MonadReader SessionId, MonadTrans)
```

For most type class instances, we use Haskell's standard `deriving` mechanism, which provides sensible defaults. The `Applicative` instance for `Epp`, however, needs a dedicated definition, as we need to update the session IDs of parallel choreographies in `<*>`.

```
instance Applicative (Epp m) where
  pure = Epp . pure
  f <*> a = Epp $
    local (Nest (Left ())) (unEpp f) <*> local (Nest (Right ())) (unEpp a)
```

The `<*>` operation nests the context session ID with `Left` and `Right` for each parallel choreography and calls the underlying `Network` monad's `<*>` operation (which is again an uninterpreted operation defined in terms of `CTree`).

5.5.3 Endpoint Projection, Defined. With `Epp` in place, we can now define endpoint projection, which is shown in Figure 13. The function `epp` takes a choreography, a target location, and returns a computaion in the `Epp` monad. The body of `epp` is simply a call to `interp`, with a effect handler function that handles each effect in `ChoreoSig`. The effect handler function is almost the same as

```
epp :: Choreo m a -> LocTm -> Epp m a
epp c l' = interp handler c
  where
    handler :: ChoreoSig m a -> Epp m a
    handler (Locally l m)
      | toLocTm l == l' = Epp $ wrap <$> lift (exec (m unwrap))
      | otherwise       = return Empty
    handler (Comm s a r)
      | toLocTm s == toLocTm r = return $ wrap (unwrap a)
      | toLocTm s == l' = Epp $ do
        sid <- ask
        lift (send sid (unwrap a) (toLocTm r) >> return Empty)
      | toLocTm r == l' = Epp $ do
        sid <- ask
        lift (wrap <$> recv sid (toLocTm s))
      | otherwise = return Empty
    handler (Cond l a k)
      | toLocTm l == l' = do
        Epp (do sid <- ask; lift $ broadcast sid (unwrap a))
        epp (k (unwrap a)) l'
      | otherwise = do
        x <- Epp (do sid <- ask; lift $ recv sid (toLocTm l))
        epp (k x) l'
```

Fig. 13. Parkour's implementation of endpoint projection. The highlighted lines are the parts of the code that differ from HasChor [Shen et al. 2023], and pertain to handling of session IDs.

the one used by HasChor [Shen et al. 2023], except that it retrieves the current session ID from the context and uses it for sending and receiving messages, as highlighted in the code.

5.6 The HTTP Backend

Computations in the **Network** monad rely on a network transport backend to run. Following HasChor, we define a **Backend** type class for implementing backends.

```
class Backend c where
  runNetwork :: c -> LocTm -> Network IO a -> IO a
```

Because we use Haskell's `forkIO` function to run choreographies in parallel using native threads, the result monad of `runNetwork` must be **IO**. For simplicity, we also require the local monad of **Network** to be **IO**, so that local computations can be run in the result monad. This limitation could be alleviated by defining a general interface between an arbitrary local monad and **IO**, and only requiring the **Network** monad to use local monads supported by this interface. In addition, we could make the result monad of `runNetwork` an associated type, allowing parallelism mechanisms other than `forkIO`. For simplicity, and to avoid distracting from our main contributions, we do not explore these directions and leave them for future work.

Parkour users may supply their own backend implementations, and Parkour also provides an HTTP backend. The HTTP backend spawns a HTTP server for handling incoming messages and interprets effects in the **Network** monad as appropriate actions on the HTTP server. Most notably,

the HTTP server uses `forkIO` to interpret `App` in the `Network` monad so that each branch of `App` is executed in parallel.

Finally, to tie everything together, Parkour provides the `runChoreography` function mentioned earlier in Section 3.4, which projects a choreography to a given location with the initial session ID `Root` and runs the resulting `Network` program with a specific backend. It is implemented as follows:

```
runChoreography :: Backend config => config -> Choreo IO a -> LocTm -> IO a
runChoreography cfg c l = runNetwork cfg l (runReaderT (unEpp (epp c l)) Root)
```

6 A Type-Safe Implementation

In this section, we present a variant of the Parkour implementation that uses types to ensure that located values are always present, removing the need for the unsafe, partial `unwrap` function. The key idea is to make located values projectable as well, so that they have different meanings at different locations. The cost is that every definition related to the `Choreo` monad must now carry an additional projection-location argument. Moreover, we rely on the `singletons` library [Eisenberg and Weirich 2012] for advanced type-level programming, which complicates the implementation and slightly clutters the interface. Because these changes break compatibility with HasChor, we do not make this implementation the default. However, from a user's perspective, if the additional type argument is ignored, the interface remains largely compatible with HasChor, and there is a trivial translation between the two.

6.1 Type-Safe Located Values and Choreo Monad

The main reason that located values `a @ l` are not type-safe in the original `Choreo` monad is that they have a *uniform* representation for every projection location. That is, an `a @ l` is represented as a sum type, and at run time it is possible for it to be either `Wrap` or `Empty`. The core idea of the type-safe implementation is to make located values projectable, giving each projection location its own version of a located value. To do so, we add an extra projection-location argument to the type and define located values as `At a l l'`.

```
type At a (l :: LocTy) (l' :: LocTy) = (l :~: l') -> a
```

Intuitively, an `At a l l'` represents an original located value `a @ l`. The additional argument `l'` is the location to which we later want to project the value. We represent `At a l l'` as a function that, given a proof that `l` is equal to `l'`, returns a value of type `a`. Otherwise, the value is empty. Here, "empty" means that the function will never be called, so we do not need to provide any value. This observation is essential for defining type-safe endpoint projection. We use `(:~:)` from the `singletons` library to represent propositional equality. We can now define a type-safe unwrap function, which also takes a projection location.

```
type Unwrap l l' = forall a. At a l l' -> a

unwrap :: (l :~: l') -> Unwrap l l'
unwrap pf x = x pf
```

Similarly, we also add projection locations to `ChoreoSig` and `Choreo`. Instead of using the `Proxy l` type for locations, we use `SSymbol l` from the `singletons` library, which provides a single representation of locations at both the type and term levels. The `SSymbol l` type lets us pattern-match on a term-level location and use the result to refine types, a common programming pattern in dependently typed programming. The user-facing APIs are defined similarly to those in the previous section, so we omit them here.

```
1  epp :: forall m l' a. Choreo m l' a -> SSymbol l' -> Epp m a
2  epp c l' = interp handler c
3    where
4      handler :: forall a. ChoreoSig m l' a -> Epp m a
5      handler (Locally l m) = case l %~ l' of
6        (Proved pf) -> Epp $ do
7          a <- lift (exec (m (unwrap pf)))
8          return (\_ -> a)
9        (Disproved dpf) -> return (\pf -> absurd (dpf pf))
10     handler (Comm s a r) = case s %~ r of
11       (Proved Refl) -> return a
12       (Disproved dpf) -> case (s %~ l', r %~ l') of
13         (Proved pf1, Proved pf2) -> absurd (dpf (trans pf1 (sym pf2)))
14         (Proved pf1, Disproved dpf2) -> Epp $ do
15           sid <- ask
16           lift (send sid (a pf1) (toLocTm r))
17           return (\pf -> absurd (dpf2 pf))
18         (Disproved dpf1, Proved pf2) -> Epp $ do
19           sid <- ask
20           a <- lift (recv sid (toLocTm s))
21           return (\_ -> a)
22         (Disproved dpf1, Disproved dpf2) -> return (\pf -> absurd (dpf2 pf))
23     handler (Cond l a k) = case l %~ l' of
24       (Proved pf) -> let a' = a pf in do
25         Epp (do sid <- ask; lift $ broadcast sid a')
26         epp (k a') l'
27       (Disproved dpf) -> do
28         x <- Epp (do sid <- ask; lift $ recv sid (toLocTm l))
29         epp (k x) l'
```

Fig. 14. Type-Safe Endpoint Projection

```
data ChoreoSig m l' a where
  Locally :: (KnownSymbol l) =>
    SSymbol l -> (Unwrap l l' -> m a) -> ChoreoSig m l' (At a l l')
  Comm :: (Show a, Read a, KnownSymbol l, KnownSymbol l'') =>
    SSymbol l -> At a l l' -> SSymbol l'' -> ChoreoSig m l' (At a l'' l')
  Cond :: (Show a, Read a, KnownSymbol l) =>
    SSymbol l -> At a l l' -> (a -> Choreo m l' b) -> ChoreoSig m l' b

type Choreo m l' a = Tree (ChoreoSig m l') a
```

6.2 Type-Safe Endpoint Projection

Figure 14 shows the type-safe endpoint projection function `epp`. The type of `epp` now needs to account for the additional projection-location argument `l'`. Compared to the non-type-safe implementation we saw earlier in Figure 13, the main differences lie in how we compare the initiator and target locations in `handler`. We use the proof-carrying comparison (`%~`) from `singletons`, which produces either a proof or a disproof when comparing two singleton values. We can then

use the proof to create a safe unwrap function, use the located value provided by the constructor, or use the disproof to show that a case is unreachable or that a located value can never be used. For example, on line 7, we use the proof to create a safe unwrap function for the local computation. On line 9, we are required to create a located value. In the previous non-type-safe version, we would return `Empty`. Here, however, we can return a dummy function that uses `absurd`, indicating that the function will never be called because its argument would contradict the disproof. Other cases follow a similar pattern.

7 Related Work

A brief history of choreographic programming. Choreographies originally emerged as specification languages for distributed systems [World Wide Web Consortium 2004, 2005] and were not seen as a way to write executable programs. Early research on choreographies formalized these specification languages and developed the theory of endpoint projection. Several such early works [Qiu et al. 2007; Carbone et al. 2012; Lanese et al. 2008] had explicit parallel operators; in that sense, our work on explicit parallelism in choreographies can be seen as reviving an old idea, rather than proposing an entirely new one. However, Parkour aims to be a fully-fledged choreographic programming language rather than a specification language.

The paradigm of choreographies as executable programs began to take shape with the work of Carbone and Montesi [2013] and with Montesi's dissertation in 2013. Subsequent work [Cruz-Filipe and Montesi 2017, 2020; Cruz-Filipe et al. 2022; Hirsch and Garg 2022; Graversen et al. 2024] has advanced the theory of choreographic programming in various ways. The choreographic languages presented in these works use standard out-of-order semantics and do not have explicit parallelism. Plyukhin et al. [2024] introduced fully out-of-order choreographies, which we discussed at length in Section 2, and implemented fully out-of-order choreographies in Ozone, which is an API for the Choral choreographic programming language [Giallorenzo et al. 2024].

Implementation approaches for CP languages. Two flavors of practical implementations of choreographic programming have emerged: standalone languages with their own compiler, exemplified by Choral [Giallorenzo et al. 2024]; and library implementations, exemplified by HasChor [Shen et al. 2023] and others [Wiersdorf and Greenman 2025; Bohosian and Hirsch 2025; Bates et al. 2025], which are embedded in a host language. Our Haskell implementation of Parkour takes the latter, library-based approach.

To achieve a purely library-level implementation with no compiler support needed, endpoint projection can be carried out at run time [Shen et al. 2023], and that is how the Parkour Haskell library does it. One disadvantage of the run-time EPP approach (which Parkour shares with HasChor) is that propagation of knowledge of choice is inefficient: we broadcast the value of the scrutinee of a conditional expression to all locations participating in the entire choreography, whether they need the information or not. To mitigate this problem, Bates et al. [2025] introduced *conclaves*, a more sophisticated knowledge-of-choice strategy that tracks the locations participating in each choreographic expression and interprets broadcasts within that context. Although we have not tried to implement conclaves in Parkour, we believe they would synergize with our approach.

Finally, if the host language has a sufficiently powerful macro system, an alternative to run-time EPP for library-level choreographic programming is to carry out EPP at macro expansion time [Wiersdorf and Greenman 2025; Lugović and Jongmans 2024; Bohosian and Hirsch 2025]. We have not yet looked into the possibility of bringing explicit parallelism to choreographies that are implemented as macros.

Tree structure for computations. Free and freer monads have been used in functional programming to model algebraic effects [Kiselyov and Ishii 2015]. Examples of such libraries in Haskell include freer-simple,[4] fused-effects,[5] and polysemy,[6], among others.

The view of free monads and related structures as *trees* originated with interaction trees [Xia et al. 2020], which are coinductive freer monads used in theorem provers. Since then, various tree structures have been proposed, including action trees [Swamy et al. 2020], choice trees [Chappe et al. 2025], and guarded interaction trees [Stepanenko et al. 2025]. Li and Weirich [2022] generalized the study to free structures of other typeclasses, including a version of applicative functors that support parallelism. All these tree structures model computation using "mixed" embeddings for verification purposes.

Our `CTree` resembles Swamy et al. [2020]'s `ctree_total`, whose definition is as follows:

```
type ctree_total : nat -> Type -> Type =
| Ret : #a:_ -> x:a -> ctree_total 0 a
| Act : #a:_ -> act:action_tot a -> ctree_total 1 a
| Par : (#aL #aR #nL #nR:_) ->
        ctree_total nL aL -> ctree_total nR aR ->
        ctree_total (nL+nR+1) (aL & aR)
| Bind : (#a #b #n1 #n2:_) ->
         f:ctree_total n1 a -> g:(x:a -> ctree_total n2 b) ->
         ctree_total (n1+n2+1) b
```

Our definition differs from `ctree_total` in two ways: (1) Our `CTree` is not annotated with a natural number that counts the number of `Act`, `Par`, and `Bind` nodes, and (2) We define parallel composition using the more general reified applicative functor `App`, then we define `par` as a wrapper over `App`.

Our `CTree` is also similar to a combination of three *program adverbs* proposed by Li and Weirich [2022]:

```
Variant ReifiedPure (K : Set -> Set) (R : Set) : Set :=
| Pure (r : R).
Variant ReifiedApp (K : Set -> Set) (R : Set) : Set :=
| LiftA2 {X Y : Set} (f : X -> Y -> R) (g : K X) (a : K Y).
Variant ReifiedMonad (K : Set -> Set) (R : Set) : Set :=
| Bind {X : Set} (m : K X) (g : X -> K R).
```

Li and Weirich use Church encodings of the least fixpoint combinator to compose the above three program adverbs for compositionality. We use a fixed definition here because that is sufficient for choreographic programming. In addition, Li and Weirich's definition is based on the `liftA2` operation of reified applicative functors, whereas our `CTree` is based on `<*>`.

Functional abstractions for parallelism. Various works have studied abstractions for parallelism in functional programming. Our approach is directly inspired by Haxl [Marlow et al. 2014], which identifies applicative functors as an interface for implementing parallelism. Haxl intentionally sets apart the behavior of applicative functors and monads (we can call this kind of applicative functor a *non-rigid* applicative functor, following the definitions of Mokhov et al. [2019]), so that `<*>` encodes parallel operations and `(>>=)` encodes sequential operations. This also enables Haxl to use Haskell's `ApplicativeDo` extension to write parallel programs using the syntax of sequential programs [Marlow et al. 2016]. This is exactly the same as our approach. However, Haxl uses a special `Fetch` datatype as the applicative functor instance, whereas we use a general `CTree` that

[4]https://hackage.haskell.org/package/freer-simple
[5]https://hackage.haskell.org/package/fused-effects
[6]https://hackage.haskell.org/package/polysemy

reifies applicative functors and monads. In addition, Haxl restricts its use case to read-only programs to avoid wrongfully parallelizing operations that mutate shared state. Parkour does not have this restriction, but instead offers the `ApplicativeDo` extension as an opt-in feature for use cases that call for Ozone-like behavior.

Others have studied parallelism using selective applicative functors in the form speculative execution [Mokhov et al. 2019], or using arrow-based abstraction [Hughes 2000] such as parallel arrows [Braun et al. 2018] and causal commutative arrows [Liu et al. 2011; Yallop and Liu 2016]. Arrows are a more expressive abstraction than applicative functors [Lindley et al. 2011], and VanDomelen et al. [2025] have demonstrated freer arrows as a suitable abstraction for choreographic programming, so freer arrows could potentially be useful for implementing parallelism for library-level choreographic programming. Indeed, we initially experimented implementing Parkour with arrows for this reason. However, we eventually decided against using arrows because our prior experiments show that the applicative interface and `ApplicativeDo` syntax provide a cleaner way to write choreographies than the arrow interface based on >>>, `first`, and arrows' **do**-notation [Paterson 2001]. We leave the exploration of a better interface for the arrow-based approach as future work.

8 Conclusion

We presented Parkour, a new library-level choreographic programming framework that brings explicit parallelism to choreographies, avoiding the need for out-of-order semantics. Parkour's design is based on a shift in perspective: instead of writing a choreography that *appears* sequential, and then relying on the language implementation to relax ordering constraints where possible (and risk relaxing them too much!), in Parkour the programmer uses `par` for explicit parallel composition of choreographies. Projected programs can have intra-node parallelism, but only when a node participates in multiple choreographies that were composed with `par`.

Parkour's implementation as a Haskell library means that programmers can leverage the extensive ecosystem of the host language, for instance, by using abstractions such as software transactional memory to coordinate parallel tasks, or by opting in to Ozone-style implicit parallelism via **`ApplicativeDo`** if desired. Parkour therefore reaps the benefits of the library-level approach to choreographic programming that originated with HasChor [Shen et al. 2023], leveraging the host-language ecosystem in new ways that HasChor did not anticipate.

Parkour's explicit parallelism is not an entirely new idea, but a resurrection of the explicit parallel operator used in early choreography specification languages [Qiu et al. 2007; Carbone et al. 2012; Lanese et al. 2008] that predated the use of choreographies as executable programs [Carbone and Montesi 2013; Montesi 2013]. With Parkour, we hope to demonstrate that explicit parallel composition is feasible and useful in executable choreographies, too, and that it fits elegantly into a practical framework for library-level choreographic programming.

References

Randy Allen and Ken Kennedy. 2001. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach* (1 ed.). Morgan Kaufmann. 816 pages.

Mako Bates, Shun Kashiwa, Syed Jafri, Gan Shen, Lindsey Kuper, and Joseph P. Near. 2025. Efficient, Portable, Census-Polymorphic Choreographic Programming. *Proc. ACM Program. Lang.* 9, PLDI, Article 193 (June 2025), 24 pages. doi:10.1145/3729296

Alexander Bohosian and Andrew K. Hirsch. 2025. Choreographies as Macros. *Electronic Proceedings in Theoretical Computer Science* 420 (May 2025), 12–21. doi:10.4204/eptcs.420.2

Martin Braun, Oleg Lobachev, and Phil Trinder. 2018. Arrows for Parallel Computation. *CoRR* abs/1801.02216 (2018). arXiv:1801.02216 http://arxiv.org/abs/1801.02216

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

Marco Carbone and Fabrizio Montesi. 2013. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 263–274. doi:10.1145/2429069.2429101

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

Nicolas Chappe, Paul He, Ludovic Henrio, Eleftherios Ioannidis, Yannick Zakowski, and Steve Zdancewic. 2025. Choice trees: Representing and reasoning about nondeterministic, recursive, and impure programs in Rocq. *J. Funct. Program.* 35 (2025). doi:10.1017/S0956796825100105

Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. 2022. Functional Choreographic Programming. In *Theoretical Aspects of Computing – ICTAC 2022*, Helmut Seidl, Zhiming Liu, and Corina S. Pasareanu (Eds.). Springer International Publishing, Cham, 212–237.

Luís Cruz-Filipe and Fabrizio Montesi. 2017. Procedural Choreographic Programming. In *Formal Techniques for Distributed Objects, Components, and Systems*, Ahmed Bouajjani and Alexandra Silva (Eds.). Springer International Publishing, Cham, 92–107.

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

Richard A. Eisenberg and Stephanie Weirich. 2012. Dependently typed programming with singletons. In *Proceedings of the 2012 Haskell Symposium* (Copenhagen, Denmark) *(Haskell '12)*. Association for Computing Machinery, New York, NY, USA, 117–130. doi:10.1145/2364506.2364522

Saverio Giallorenzo, Fabrizio Montesi, and Marco Peressotti. 2024. Choral: Object-oriented Choreographic Programming. *ACM Trans. Program. Lang. Syst.* 46, 1, Article 1 (Jan. 2024), 59 pages. doi:10.1145/3632398

Eva Graversen, Andrew K. Hirsch, and Fabrizio Montesi. 2024. Alice or Bob?: Process polymorphism in choreographies. *Journal of Functional Programming* 34 (2024), e1. doi:10.1017/S0956796823000114

Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable memory transactions. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Chicago, IL, USA) *(PPoPP '05)*. Association for Computing Machinery, New York, NY, USA, 48–60. doi:10.1145/1065944.1065952

Andrew K. Hirsch and Deepak Garg. 2022. Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.* 6, POPL, Article 23 (Jan. 2022), 27 pages. doi:10.1145/3498684

Paul Hudak. 1996. Building domain-specific embedded languages. *ACM Comput. Surv.* 28, 4es (Dec. 1996), 196–es. doi:10.1145/242224.242477

John Hughes. 2000. Generalising monads to arrows. *Science of Computer Programming* 37, 1 (2000), 67–111. doi:10.1016/S0167-6423(99)00023-4

Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, Ben Lippmeier (Ed.). ACM, 94–105. doi:10.1145/2804302.2804319

Ivan Lanese, Claudio Guidi, Fabrizio Montesi, and Gianluigi Zavattaro. 2008. Bridging the Gap between Interaction- and Process-Oriented Choreographies. In *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods (SEFM '08)*. IEEE Computer Society, USA, 323–332. doi:10.1109/SEFM.2008.11

Yao Li and Stephanie Weirich. 2022. Program adverbs and Tlön embeddings. *Proc. ACM Program. Lang.* 6, ICFP (2022), 312–342. doi:10.1145/3547632

Sam Lindley, Philip Wadler, and Jeremy Yallop. 2011. Idioms are Oblivious, Arrows are Meticulous, Monads are Promiscuous. *Electron. Notes Theor. Comput. Sci.* 229, 5 (2011), 97–117. doi:10.1016/j.entcs.2011.02.018

Hai Liu, Eric Cheng, and Paul Hudak. 2011. Causal commutative arrows. *J. Funct. Program.* 21, 4-5 (2011), 467–496. doi:10.1017/S0956796811000153

Lovro Lugović and Sung-Shik Jongmans. 2024. Klor: Choreographies in Clojure. https://github.com/lovrosdu/klor

Simon Marlow. 2011. Parallel and concurrent programming in Haskell. In *Proceedings of the 4th Summer School Conference on Central European Functional Programming School* (Budapest, Hungary) *(CEFP'11)*. Springer-Verlag, Berlin, Heidelberg, 339–401. doi:10.1007/978-3-642-32096-5_7

Simon Marlow, Louis Brandy, Jonathan Coens, and Jon Purdy. 2014. There is no fork: an abstraction for efficient, concurrent, and concise data access. In *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, Johan Jeuring and Manuel M. T. Chakravarty (Eds.). ACM, 325–337. doi:10.1145/2628136.2628144

Simon Marlow, Simon Peyton Jones, Edward Kmett, and Andrey Mokhov. 2016. Desugaring Haskell's do-notation into applicative operations. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September*

22-23, 2016, Geoffrey Mainland (Ed.). ACM, 92–104. doi:10.1145/2976002.2976007

Andrey Mokhov, Georgy Lukyanov, Simon Marlow, and Jerémie Dimino. 2019. Selective applicative functors. *Proc. ACM Program. Lang.* 3, ICFP (2019), 90:1–90:29. doi:10.1145/3341694

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

Fabrizio Montesi. 2023. *Introduction to Choreographies*. Cambridge University Press. doi:10.1017/9781108981491

Ross Paterson. 2001. A New Notation for Arrows. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP '01), Firenze (Florence), Italy, September 3-5, 2001*, Benjamin C. Pierce (Ed.). ACM, 229–240. doi:10.1145/507635.507664

Dan Plyukhin, Marco Peressotti, and Fabrizio Montesi. 2024. Ozone: Fully Out-of-Order Choreographies. In *38th European Conference on Object-Oriented Programming (ECOOP 2024) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 313)*, Jonathan Aldrich and Guido Salvaneschi (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 31:1–31:28. doi:10.4230/LIPIcs.ECOOP.2024.31

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

Christopher Rawlinson and Mirko Guaralda. 2011. Play in the city: Parkour and architecture. In *Proceedings of the First International Conference on Engineering, Designing and Developing the Built Environment for Sustainable Wellbeing*. Queensland University of Technology, 19–24.

Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). 7, ICFP, Article 207 (Aug. 2023), 25 pages. doi:10.1145/3607849

Sergei Stepanenko, Emma Nardino, Dan Frumin, Amin Timany, and Lars Birkedal. 2025. Context-Dependent Effects in Guarded Interaction Trees. In *Programming Languages and Systems - 34th European Symposium on Programming, ESOP 2025, Held as Part of the International Joint Conferences on Theory and Practice of Software, ETAPS 2025, Hamilton, ON, Canada, May 3-8, 2025, Proceedings, Part II (Lecture Notes in Computer Science)*, Viktor Vafeiadis (Ed.). Springer, 286–313. doi:10.1007/978-3-031-91121-7_12

Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 2020. SteelCore: an extensible concurrent separation logic for effectful dependently typed programs. *Proc. ACM Program. Lang.* 4, ICFP (2020), 121:1–121:30. doi:10.1145/3409003

Grant VanDomelen, Gan Shen, Lindsey Kuper, and Yao Li. 2025. Freer Arrows and Why You Need Them in Haskell. In *Proceedings of the 18th ACM SIGPLAN International Haskell Symposium, Haskell 2025, Singapore, October 12-18, 2025*, J. Garrett Morris and Ningning Xie (Eds.). ACM, 94–108. doi:10.1145/3759164.3759352

Ashton Wiersdorf and Ben Greenman. 2025. Chorex: Restartable, Language-Integrated Choreographies. *Programming* 10, 3 (2025), 20:1–20:30. doi:10.22152/programming-journal.org/2025/10/20

The World Wide Web Consortium. 2004. WS Choreography Model Overview. https://www.w3.org/TR/ws-chor-model/

The World Wide Web Consortium. 2005. Web Services Choreography Description Language Version 1.0. https://www.w3.org/TR/ws-cdl-10/

Li-yao Xia, Yannick Zakowski, Paul He, Chung-Kil Hur, Gregory Malecha, Benjamin C. Pierce, and Steve Zdancewic. 2020. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. doi:10.1145/3371119

Jeremy Yallop and Hai Liu. 2016. Causal commutative arrows revisited. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, Geoffrey Mainland (Ed.). ACM, 21–32. doi:10.1145/2976002.2976019