HasChor: Functional Choreographic Programming for All (Functional Pearl)

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

Choreographic programming is an emerging paradigm for programming distributed systems. In choreographic programming, the programmer describes the behavior of the entire system as a single, unified program — a *choreography* — which is then compiled to individual programs that run on each node, via a compilation step called endpoint projection. We present a new model for functional choreographic programming where choreographies are expressed as computations in a monad. Our model supports cutting-edge choreographic programming features that enable modularity and code reuse: in particular, it supports *higher-order* choreographies, in which a choreography may be passed as an argument to another choreography, and *location-polymorphic* choreographies, in which a choreography can abstract over nodes. Our model is implemented in a Haskell library, *HasChor*, which lets programmers write choreographic programs while using the rich Haskell ecosystem at no cost, bringing choreographic programming within reach of everyday Haskellers. Moreover, thanks to Haskell's abstractions, the implementation of the HasChor library itself is concise and understandable, boiling down endpoint projection to its short and simple essence.

CCS Concepts: • **Software and its engineering → Concurrent programming structures**; **Functional languages**.

Additional Key Words and Phrases: Choreographic programming, freer monads

ACM Reference Format:
Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). *Proc. ACM Program. Lang.* 7, ICFP, Article 207 (August 2023), 25 pages. https://doi.org/10.1145/3607849

1 INTRODUCTION

A distributed system consists of a collection of *nodes* that operate independently and communicate by message passing. One of the challenges of programming distributed systems is the need to reason about the implicit *global* behavior of the system while writing the explicit *local* programs that actually run on each node. While running independently, nodes must exchange messages in a carefully executed dance: every message sent from one node must be expected by its recipient, or we risk deadlock.

The emerging paradigm of *choreographic programming* [Montesi 2013; Carbone and Montesi 2013; Cruz-Filipe and Montesi 2020; Cruz-Filipe et al. 2022; Hirsch and Garg 2022] helps to address this challenge by making the global behavior of the system *explicit*. In the choreographic paradigm, the programmer describes the behavior of a distributed system as a single, unified program: a *choreography*. One choreography is then compiled to multiple individual programs that run on

Authors' addresses: Gan Shen, University of California, Santa Cruz, USA; Shun Kashiwa, University of California, Santa Cruz, USA; Lindsey Kuper, University of California, Santa Cruz, USA.

2475-1421/2023/8-ART207
https://doi.org/10.1145/3607849

each node, via a compilation step called *endpoint projection* [Carbone et al. 2007, 2012]. If endpoint projection is sound, the resulting distributed system enjoys a guarantee of *deadlock freedom* [Qiu et al. 2007; Carbone and Montesi 2013]: by construction, every message sent will be paired with a corresponding receive. Furthermore, choreographies are amenable to whole-program analyses that can potentially rule out large classes of bugs. However, choreographic programming — and in particular *functional* choreographic programming — is still in its infancy, with the only existing functional choreographic language designs [Hirsch and Garg 2022; Cruz-Filipe et al. 2022] so far lacking any practically usable implementation.

In this paper, inspired by the potential of functional choreographic programming, we present a programming model in which choreographies are expressed as computations in a monad. We implement our model entirely as a Haskell library, which we call *HasChor*. Thanks to its embedding in Haskell, HasChor naturally supports cutting-edge choreographic programming features that enable a high level of abstraction. HasChor programmers have access to all of Haskell's rich ecosystem, making functional choreographic programming viable for practical software development. Furthermore, we find that Haskell's abstractions are a great fit for implementing the HasChor library itself, enabling a concise, understandable implementation of choreographic programming.

We make the following specific contributions:

- *A monad for choreographic programming.* Our main contribution is a new model for choreographic programming based on a monad and implemented as a Haskell library (Section 3). While choreographic programming (and functional choreographic programming) is not new, HasChor is to our knowledge the first practically usable implementation of functional choreographic programming (that is, implemented in a general-purpose programming language, rather than in a proof assistant or on paper), and the first to be based on a monad. HasChor supports *higher-order choreographies* and *location-polymorphic choreographies*, both features that enable modularity and code reuse.
- *A case study for practical functional choreographic programming.* As a case study, we use HasChor to implement a standard of the distributed systems literature: a replicated, in-memory key-value store (Section 4). We build up the implementation in stages, showing how higher-order choreographies and location polymorphism enable a high level of abstraction in our key-value store implementation. Our experience carrying out this case study suggests that HasChor is a practically usable implementation of functional choreographic programming. It can be installed just like any Haskell library, compiled just like any Haskell program, and can use any Haskell tools for development and debugging, bringing choreographic programming within reach of everyday Haskellers.
- *An understandable implementation of choreographic programming.* Finally, we contribute evidence that functional programming makes choreographic programming *itself* straightforward to implement (Section 5). The core implementation of the HasChor library is less than 150 lines of code.[1] We find that functional programming abstractions make it especially straightforward to implement endpoint projection, the central concept of choreographic programming. In fact, HasChor's concise implementation of endpoint projection helped *us* grasp the essence of choreographic programming; we hope it will do the same for readers.

We have published the HasChor implementation and a collection of example programs, including all of the examples from this paper, at github.com/gshen42/HasChor.

[1] The optional HTTP backend adds another roughly 100 lines.

```
buyer :: Network IO (Maybe Day)
buyer = do
  send title "seller"
  price <- recv "seller"
  if price <= budget
  then do
    send True "seller"
    date <- recv "seller"
    return (Just date)
  else do
    send False "seller"
    return Nothing
```

```
seller :: Network IO ()
seller = do
  title <- recv "buyer"
  send (priceOf title) "buyer"
  decision <- recv "buyer"
  if decision
  then do
    send (deliveryDate title) "buyer"
  else do
    return ()
```

Fig. 1. The bookseller protocol implemented as individual programs

```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = do
  title'   <- (buyer, title) ~> seller
  price    <- seller `locally` \un -> return (priceOf (un title'))
  price'   <- (seller, price) ~> buyer
  decision <- buyer `locally` \un -> return (un price' <= budget)

  cond (buyer, decision) \case
    True -> do
      date  <- seller `locally` \un -> return (deliveryDate (un title'))
      date' <- (seller, date) ~> buyer
      return (Just date')
    False -> do
      return Nothing
```

Fig. 2. The bookseller protocol implemented as a choreography

2 A TOUR OF CHOREOGRAPHIC PROGRAMMING IN HASCHOR

In this section, we give a tour of choreographic programming with a series of examples and introduce its key ideas through the lens of HasChor. We do not provide extensive explanations of language constructs in HasChor, but will point to sections in the rest of the paper where they are formally introduced.

2.1 The Bug-Prone Bookseller

As an example of the kind of bug that choreographic programming is designed to prevent, consider a well-known example from the literature: the "bookseller" protocol [Carbone et al. 2007, 2012; Honda et al. 2008; World Wide Web Consortium 2006]. The first version of the protocol that we will consider involves an interaction between two participants: a seller and a (would-be) buyer. The protocol begins with the buyer sending the title of a book they want to buy to the seller. The seller replies with the book's price, and the buyer checks if the price is within their budget. If the buyer can afford the book, they inform the seller and get back a delivery date; otherwise, they tell the seller they will not buy the book.

Figure 1 shows an implementation of the bookseller protocol as individual programs. We call these programs *network programs* and they are written in the `Network` monad (Section 5.4) provided by HasChor. The `buyer` has a particular `title :: String` and a `budget :: Int` in mind, and the `seller` has `priceOf :: String -> Int` and `deliveryDate :: String -> Day` functions for looking up the price of a book and the date on which a book can be delivered. Network programs interact with each other by sending messages with `send` and receiving them with `recv`. The `send` function takes as arguments a message of a serializable type (such as a book `title` on line 3 of the `buyer`, or a `Bool` on lines 7 and 11 of the `buyer`) and a destination location (such as `"seller"` on lines 3, 7, and 11 of the `buyer`), and transmits the message to the destination via some as-yet-unspecified transport mechanism. The `recv` function takes a location as its argument, blocks until a message has arrived from the specified location via the transport mechanism, and then returns the message.

Even in a simple protocol like this one, it is easy to introduce a bug that will lead to a deadlock. For example, suppose that the implementor of `buyer` forgets to write `send True seller` on line 7. Then both `buyer` and `seller` will be stuck: the `seller` will wait forever on line 5 for a decision from the `buyer`, while the `buyer` will wait forever on line 8 for a date from the `seller`.

2.2 The Choreographic Approach

As a first example of choreographic programming in HasChor, Figure 2 shows the implementation of the bookseller protocol as a single choreography. The choreography expresses the interaction between locations (Section 3.1) — in this case, buyer and seller — from an objective, *global* point of view, like the script of a play [Giallorenzo et al. 2021]; in HasChor, choreographies are written in the `Choreo` monad (Sections 3.3 and 5.3). The `bookseller` choreography returns a value of type `Maybe Day @ "buyer"`, which is a located value (Sections 3.2 and 5.2) that represents a value of type `Maybe Day` at the `"buyer"` location.[2]

One of the hallmarks of choreographic programming is a single language construct that takes the place of both `send` and `recv`. For example, `send title "seller"` (from line 3 of the `buyer` in Figure 1) and `recv "buyer"` (from line 3 of the `seller` in Figure 1) are replaced in Figure 2 by line 3 of the choreography:

```
title' <- (buyer, title) ~> seller
```

where `(~>)` is the HasChor construct that represents communication between two locations. Here, `title` represents the title sent by the buyer, while `title'` represents the title received by the seller. Locations can also perform local actions by using the `locally` function, which takes as arguments a location and a local computation (in this case, an `IO` computation) with access to a special `un` function that can unwrap located values into normal values for use with other Haskell functions. For example, on line 4 of Figure 2, the seller looks up the price of the book by doing a local computation.

The choreography in Figure 2 also illustrates the use of `cond`, another HasChor construct. The `cond` language construct implements *knowledge of choice*, a distinctive feature of choreographic languages [Carbone and Montesi 2013; Hirsch and Garg 2022; Giallorenzo et al. 2020]. To understand the intuition for `cond`, notice how in the programs in Figure 1, the buyer must explicitly inform the seller of its decision to buy the book or not, by calling either `send True seller` or `send False seller`, with the corresponding `recv` on line 5 of `seller`. This explicit synchronization is necessary in the non-choreographic implementation of the protocol to make the seller's `send` (or lack thereof) of the book's delivery date match up with a `recv` (or lack thereof) on the buyer's side. In HasChor, on the other hand, the `cond` language construct takes care of inserting the

[2]Do not confuse the `@` symbol with type application. Throughout this paper, the `@` symbol always refers to located values.

```
bookseller :: (Int @ "buyer" -> Choreo IO (Bool @ "buyer"))
           -> Choreo IO (Maybe Date @ "buyer")
bookseller mkDecision = do
  title'   <- (buyer, title) ~> seller
  price    <- seller `locally` \un -> return (priceOf (un t))
  price'   <- (seller, price) ~> buyer
  decision <- mkDecision p

  cond (buyer, decision) \case
    True -> do
      date  <- seller `locally` \un -> return (deliveryDate (un t))
      date' <- (seller, date) ~> buyer
      return (Just date')
    False -> do
      return Nothing

mkDecision1 :: Int @ "buyer" -> Choreo IO (Bool @ "buyer")
mkDecision1 p = do
  buyer `locally` \un -> return (un p <= budget)

mkDecision2 :: Int @ "buyer" -> Choreo IO (Bool @ "buyer")
mkDecision2 p = do
  price''  <- (buyer, price') ~> buyer2
  contrib  <- buyer2 `locall` \un -> return (un price' / 2)
  contrib' <- (buyer2, contrib) ~> buyer
  buyer `locally` \un -> return (un p <= un contrib' + budget)
```

Fig. 3. A higher-order version of the bookseller choreography with changes to Figure 2 highlighted

necessary synchronization *automatically*. We describe cond in more detail, along with the **Choreo** monad and the rest of the HasChor API, in Section 3.

2.3 Higher-Order Choreographies and Location Polymorphism

While the simple bookseller example suffices to illustrate the concept of a choreography, it is only the beginning of what is possible. Carbone and Montesi [2013] use choreographies to implement a *two-buyer* bookseller protocol, in which a second buyer, say, buyer2, contributes half of the book's price to the budget. We can implement this two-buyer protocol in HasChor by adding a little additional communication to the one-buyer choreography in Figure 2, for instance, by replacing line 6 of Figure 2 with

```
price''  <- (buyer, price') ~> buyer2
contrib  <- buyer2 `locally` \un -> return (un p' / 2)
contrib' <- (buyer2, contrib) ~> buyer
decision <- buyer `locally` \un -> return (un p <= un contrib' + budget)
```

Indeed, it is a strength of choreographies that protocols involving three or more parties are straightforward to express.

```
1  bookseller :: Proxy a -> Choreo IO (Maybe Day @ a)
2  bookseller buyer = do
3    title'   <- (buyer, title) ~> seller
4    price    <- seller `locally` \un -> return (priceOf (un title'))
5    price'   <- (seller, price) ~> buyer
6    decision <- buyer `locally` \un -> return (un price' <= budget)
7
8    cond (buyer, decision) \case
9      True -> do
10       date  <- seller `locally` \un -> return (deliveryDate (un t))
11       date' <- (seller, date) ~> buyer
12       return (Just date)
13     False -> do
14       return Nothing
```

Fig. 4. A location-polymorphic version of the bookseller choreography with changes to Figure 2 highlighted

However, we can do still better. Recent work on choreographic programming proposes *higher-order* choreographies [Giallorenzo et al. 2020; Hirsch and Garg 2022; Cruz-Filipe et al. 2022]. Hirsch and Garg motivate the need for higher-order choreographies by pointing out that the one-buyer protocol and the two-buyer protocol share a common pattern that can be abstracted out: in each case, the buyer makes a decision to buy or not buy via some process, potentially involving communication. With higher-order choreographies, this decision process can be implemented as a sub-choreography, which can then be passed as an argument to the main choreography, enabling code reuse. Figure 3 shows the implementation of such a higher-order bookseller. The `bookseller` function takes a sub-choreography of type `Int @ "buyer" -> Choreo IO (Bool @ "buyer")`. Lines 17–19 of Figure 3 implement the single-buyer bookseller, equivalent to what we saw earlier in Figure 2, as the function `mkDecision1`, while lines 21–26 of Figure 3 implement the two-buyer bookseller as the function `mkDecision2`.

Another feature that raises the abstraction level of choreographic languages is *location polymorphism* [Giallorenzo et al. 2020]. As a motivating example, rather than implementing the bookseller protocol in a way that is specific to a *particular* buyer, we might instead wish to implement a book-selling service to which an *arbitrary* buyer may connect. With location polymorphism, we can write a choreography that abstracts over the buyer as shown in Figure 4, again enabling code reuse and a higher level of abstraction. Here, the `bookseller` takes an argument of type `Proxy a`[3], which could be any location.

Neither higher-order choreographies nor location polymorphism are new contributions of HasChor (see Section 6 for a discussion of related work). However, HasChor is to our knowledge the first practical *functional* choreographic programming framework to support these features. Still, we cannot claim too much credit: a happy consequence of our implementation approach is that both higher-order choreographies and location polymorphism "just work" in HasChor, because we inherit support for higher-order and polymorphic programming from the host language.

2.4 Endpoint Projection

The central concept of the choreographic programming paradigm — and the technology that makes choreographies viable as a *programming* language — is *endpoint projection* (EPP) [Qiu et al. 2007;

[3]We use `Proxy a` to represent a type-level location, which we will explain in more detail in Section 3.1.

Mendling and Hafner 2005; Carbone et al. 2007, 2012]. A choreography like that in Figure 2 is already useful as a way of *specifying* the global behavior of a protocol. However, if we wish to actually *run* the protocol in a distributed fashion, then we must have a way of extracting from the global choreography the individual network programs that will run at the buyer's and the seller's respective endpoints and make explicit calls to `send` and `recv`, like the programs in Figure 1. This is precisely what EPP does. While EPP may sound complicated, Haskell's high-level abstractions — in particular, the *freer monad* [Kiselyov and Ishii 2015] — let us boil EPP down to its short and simple essence, with an implementation in just a few lines of code. We describe HasChor's implementation of EPP, along with other HasChor internals, in Section 5.

Finally, to run the collection of projected programs produced by EPP, HasChor needs a message transport backend to actually implement sending and receiving, whether by HTTP, TCP, or messenger pigeon. Here, again, the freer monad abstraction helps us: because freer monads separate the interface and implementation of an effectful computation, HasChor supports *swappable backends* that implement a given interface. Users of HasChor may implement their own backend or use the default HTTP backend that the HasChor framework provides.

3 THE HASCHOR API

In this section, we present the API of HasChor for writing and executing choreographies. This section is oriented toward the user of HasChor and describes how to use the constructs provided by the library, while Section 5 provides an exposition of how these constructs are implemented. HasChor can be viewed as an embedded domain-specific language for choreographic programming in Haskell. The programming model it provides is typed, functional, higher-order, and polymorphic. To fully support all these features, we require a number of language extensions from the Glasgow Haskell Compiler. In particular, we use the `GHC2021` [GHC Team 2023] set of language extensions, and additionally, we use `DataKinds` and `GADTs` extensively.

HasChor's API design is influenced by a variety of choreographic and *multitier* [Weisenburger et al. 2020] languages, such as Pirouette [Hirsch and Garg 2022], Choral [Giallorenzo et al. 2020], ML5 [Murphy et al. 2007], and ScalaLoci [Weisenburger et al. 2018]; we discuss these and other related works in Section 6.

3.1 Locations

Choreographic programming abstracts nodes in a distributed system as *locations*. These locations are treated atomically, and we assume location equality is decidable, so we can distinguish different locations. In HasChor, we define a type alias `LocTm` for locations, which we represent as `String`s. Because we also need locations to show up in types for *located values* (which we describe next in Section 3.2), we also define type-level locations as `Symbol`s (which are type-level `String`s).

```
type LocTm = String -- term-level location
type LocTy = Symbol -- type-level location
```

To provide a type-level location at the term level, we use the standard `Proxy` datatype. For example, the `buyer` in the `bookseller` choreography we saw in Figure 2 is defined as a `Proxy` of type `Proxy "buyer"`:

```
buyer :: Proxy "buyer"
buyer = Proxy -- a term-level proxy for a type-level location
```

3.2 Located Values

Since choreographic programming provides a global view of distributed programming, values at different locations will show up in the same choreography, which would make it possible for a

location to access a value that doesn't reside on it, if we are not careful. To avoid this problematic behavior, HasChor annotates each value with the location where it resides and ensures that only that location can access it. We call these annotated values *located values*, and write their types as `a @ l`, which represents a value of type `a` at location `l`.

Located values are not immediately usable. To use a located value in a choreography, the value must first be "unwrapped" to a normal value by applying an *unwrap function* of type `forall a. a @ l -> a` to it. Crucially, HasChor needs to ensure that only location `l` is allowed to unwrap an `a @ l` to `a` and use it. To accomplish this, we leave the definition of located values opaque to the user, and only provide access to the unwrap function in a safe manner through the `Choreo` monad.

3.3 The Choreo Monad

The programming model of HasChor is structured around a monad, `Choreo`. In HasChor, choreographies are computations of type `Choreo m a`:

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

`Choreo` computations are parameterized by a user-supplied *local monad* `m`. HasChor assumes that each location participating in a choreography of type `Choreo m a` can run *local computations* of type `m a`. The user can choose the local monad as they wish, with the only requirement being that the local monad needs to subsume `IO`, i.e. being an instance of `MonadIO`, as each node should be able to send and receive messages. For example, the `bookseller` choreography in Figure 2 has type `Choreo IO (Maybe Day)`.

As mentioned above, a `Choreo` computation can be thought of as a program in an embedded DSL for choreographic programming in Haskell. This embedded DSL supports three primitive language constructs: `locally`, for carrying out local computations at a location; `(~>)`, for communication between locations, and `cond`, for choreographic conditionals. We describe each of the language constructs in more detail below.

3.3.1 Local Computation. The `locally` function is for doing local computation at a given location. It takes a location `l` and a local computation of type `m a` and returns a value of type `a` located at `l`:

```
locally :: Proxy l -> (Unwrap l -> m a) -> Choreo m (a @ l)
```

The type of `locally`'s second argument calls for some additional explanation. The local computation is given an unwrap function of type `Unwrap l`, which is an alias for `forall a. a @ l -> a`. The unwrap function allows the local computation to unwrap values located at `l` in the context, but not values located at any other locations. For example, the seller in the bookseller choreography in Figure 2 uses the following code to look up the price of the book:

```
price <- seller `locally` \un -> return (priceOf (un t))
```

The seller first unwraps the book title `t` that it receives, and then calls `priceOf` on it and returns the result.

3.3.2 Communication. A choreographic language must have a language construct for communication between a sender and a receiver. The `(~>)` function takes a pair of the sender's location and a value of type `a` located at the sender, and a receiver's location, and returns a value of the same type `a` but located at the receiver:

```
(~>) :: (Proxy l, a @ l) -> Proxy l' -> Choreo m (a @ l')
```

For example, the seller in the bookseller choreography in Figure 2 uses the following code to communicate the price of the book that it locally computed to the buyer:

```
price' <- (seller, price) ~> buyer
```

Here, `price'` has type `Int @ "buyer"`, which represents the price the buyer receives.

For convenience, HasChor also provides a derived operation `(~~>)` that combines `locally` and `(~>)`. The `(~~>)` function carries out a local computation at a sender and then communicates the result to a receiver. It takes as arguments a pair of the sender's location and a local computation of type `m a`, and a receiver's location. Like `(~>)`, it returns a value of type `a` located at the receiver. `(~~>)` has a straightforward implementation in terms of `locally` and `(~>)`:

```
(~~>) :: (Proxy l, Unwrap l -> m a) -> Proxy l' -> Choreo m (a @ l')
(~~>) (l, c) l' = do
  x <- l `locally` c
  (l, x) ~> l'
```

For example, the seller in the bookseller protocol can use `(~~>)` to combine looking up the price of the book and communicating that price to the buyer:

```
p <- (seller, \un -> return (priceOf (un t))) ~~> buyer
```

3.3.3 Conditionals. In choreographic programming, when one node in a system makes a choice such as taking one or another branch of a conditional, other nodes need to be informed of the choice in case it affects their communication pattern in the code that follows. For example, in the bookseller protocol, the buyer's decision to buy or not buy the book must be communicated to the seller (as in the `send True seller` and `send False seller` calls in `buyer` of Figure 1). In a choreography, this would amount to writing a `(~>)` expression in every branch of the conditional, which would be tedious for the programmer. HasChor therefore provides a `cond` language construct that inserts the necessary communication automatically. `cond` takes as its arguments a pair of a location and a *scrutinee* at that location, and a function describing the follow-up choreographies depending on the scrutinee, and returns one of the follow-up choreographies.

```
cond :: (Proxy l, a @ l) -> (a -> Choreo m b) -> Choreo m b
```

We have already seen an example use of `cond` in the bookseller choreography, on lines 8–14 of Figure 2. In that example, the scrutinee expression is `decision`, which must be one of `True` or `False`. While in the bookseller choreography the scrutinee happens to be of type `Bool`, in general the scrutinee in a `cond` expression may be of any type `a`.

HasChor's `cond` makes a different design decision from most choreographic languages: typically [Hirsch and Garg 2022; Giallorenzo et al. 2020], choreographic languages require the programmer to manually send synchronization messages in the branches of a conditional expression, to notify other locations about the decision that was made. For example, in Pirouette [Hirsch and Garg 2022], endpoint projection is undefined if a choreography neglects to include these synchronization messages. In HasChor, on the other hand, the location where the conditional expression is evaluated will automatically broadcast the result to all locations. This implementation strategy can make using conditionals a bit easier for users, at the cost of potentially adding unnecessary communication (for instance, by sending the result to locations whose behavior is not affected by it). We discuss this trade-off in more detail in Section 6.

3.4 Running Choreographies

To run a choreographic program in a distributed fashion, we must project it to a collection of *network programs* that run individually at each node. The HasChor API provides a `runChoreography`

```
main :: IO ()
main = do
  [loc] <- getArgs
  case loc of
    "buyer"   -> runChoreography cfg bookseller "buyer"
    "seller"  -> runChoreography cfg bookseller "seller"
    _         -> error ("Unknown location: " ++ loc)
  return ()
  where
    cfg = mkHttpConfig [ ("buyer",  ("alice.some-school.edu", 3000))
                       , ("seller", ("bookstore.haskell.org", 4000)) ]
```

Fig. 5. Deploying the bookseller choreography of Figure 2 with HasChor's HTTP backend

function that, given a choreography and a location name, projects the choreography to a network program at the specified location using endpoint projection, and then runs the network program:

```
runChoreography :: Backend config => config -> Choreo m a -> LocTm -> m a
```

We defer further discussion of network programs and HasChor's implementation of endpoint projection to Section 5. As the type of `runChoreography` shows, it also requires the user to provide a *backend configuration* `config`, which must be an instance of the **`Backend`** type class. To run a network program that is the result of endpoint projection, HasChor needs a *message transport backend* to actually handle sending and receiving of messages. A backend configuration specifies how locations are mapped to network hosts and ports, and provides a `runNetwork` function that runs a network program at a specific location using some message transport mechanism.

While users are free to implement their own message transport backends, HasChor comes with an HTTP-based backend out of the box. The HasChor API function `mkHttpConfig` constructs an instance of **`Backend`** that can be passed to `runChoreography`.

Putting these pieces together, Figure 5 gives an example of using `runChoreography` to project the bookseller choreography of Figure 2 to a network program at each node, and then run the resulting network program at each node using HasChor's HTTP backend.

An important caveat about HasChor's support for higher-order choreographies is that it lacks *separate compilation* for higher-order choreographies. That is, in HasChor a higher-order choreography cannot be projected by itself, but must be applied to an argument choreography at projection time. For example, if we wished to project the higher-order choreography of Figure 3 to network programs, we would need to apply `bookseller` to one of `mkDecision1` or `mkDecision2` before calling `runChoreography`. Other higher-order choreographic languages, such as Pirouette [Hirsch and Garg 2022] and Choral [Giallorenzo et al. 2020], do not have this limitation.

For testing and debugging purposes, HasChor users may also wish to run a **`Choreo`** computation *directly*, without the use of EPP. To that end, HasChor provides a `runChoreo` function, which has type **`Choreo`** `m a -> m a`, and interprets a choreography as a non-distributed, single-threaded program. We can view `runChoreo` as giving a semantics to choreographies directly, rather than in terms of endpoint projection. From a verification perspective, `runChoreo` can be thought of as a *specification* of how choreographies should behave, and correctness of endpoint projection becomes a question of whether the projected collection of network programs faithfully implements the specification. We will discuss the implementation of `runChoreo` further in our discussion of HasChor internals in Section 5.

4 BEYOND BOOKSELLERS: A REPLICATED IN-MEMORY KEY-VALUE STORE

In this section, we showcase the features of HasChor by using it to implement a standard of distributed systems: a replicated in-memory key-value store. We build up the implementation in stages, starting first with a simple client-server architecture (Section 4.1), and then a *primary-backup* replication approach (Section 4.2). Next, we show how we can use HasChor's higher-order choreographies to *abstract* over the previous two implementations (Section 4.3). Finally, we show how we can use HasChor's location polymorphism to abstract out repeated code in the primary-backup implementation (Section 4.4).

The full implementations of all our examples are available at github.com/gshen42/HasChor. Along with the key-value store examples that we describe in this section, these include an implementation of Diffie-Hellman key exchange [Diffie and Hellman 1976], a two-phase commit protocol [Gray 1978; Lampson and Sturgis 1979], and a distributed merge sort.

4.1 A Simple Client-Server Key-Value Store

As a first step, we begin with a simple client-server architecture for our key-value store. The client sends requests to the server, and the server handles requests and sends responses back to the client. Figure 6 shows the HasChor implementation of the client-server key-value store. The server stores pairs of **`String`**s as a **`Map`** inside a mutable **`IORef`** as its **`State`**. It supports two kinds of **`Request`**s: **`Put`**, to set a given key-value pair, and **`Get`**, to look up the value associated with a specified key. The server uses **`Maybe String`** as its response: for **`Put`**, it sends back the value it put; for **`Get`**, it sends back the corresponding value or **`Nothing`** if it is not present. The core of the implementation is the `kvs` choreography, which describes one instance of the client-server interaction. When `kvs` is invoked, the client first sends the request to the server (line 9). Then, the server calls `handleRequest` to process the request, updates the server state as needed, and generates a response (line 10). Finally, the server sends the response to the client (line 11).

The entry point for the key-value store is `mainChoreo` (lines 22–32), which initializes the server state to the empty **`Map`** and then enters an infinite loop. Inside the loop, the program reads a command from the client's terminal by calling `readRequest` (omitted for brevity), then calls `kvs` to process the request. After the client prints the response, it goes back to the beginning of the loop and waits for the user to make another request.

To run the key-value store choreography, we use the `main` function shown in Figure 7. The call to `mkHttpConfig` sets up HasChor's HTTP backend, specifying a hostname and port number for each of the two locations (lines 9 and 10). The program takes a location name as a command-line argument and starts the choreography. Assuming the built executable is called `kvs`, with `kvs server` running, a client can interact with the key-value store from the command line:

```
$ kvs client
GET hello
> Nothing
PUT hello world
> Just "world"
GET hello
> Just "world"
```

4.2 A Replicated Key-Value Store

Now that we've seen a simple key-value store, let us consider how we can implement a ubiquitous feature of distributed systems: *data replication*. The implementation in Section 4.1 had only

```
type State = Map String String
data Request = Put String String | Get String
type Response = Maybe String

kvs :: Request @ "client"
    -> IORef State @ "server"
    -> Choreo IO (Response @ "client")
kvs request state = do
  request' <- (client, request) ~> server
  response <- server `locally` \un -> handleRequest (un request') (un state)
  (server, response) ~> client

handleRequest :: Request -> IORef State -> IO Response
handleRequest request state = case request of
  Put key value -> do
    modifyIORef state (Map.insert key value)
    return (Just value)
  Get key -> do
    state <- readIORef state
    return (Map.lookup key state)

mainChoreo :: Choreo IO ()
mainChoreo = do
  state <- server `locally` \_ -> newIORef Map.empty
  loop state
  where
    loop :: IORef State @ "server" -> Choreo IO ()
    loop state = do
      request <- client `locally` \_ -> readRequest
      response <- kvs request state
      client `locally` \un -> do putStrLn ("> " ++ show (un response))
      loop state
```

Fig. 6. The choreography for the client-server key-value store

one server, so if the server fails, we lose all the stored data. Replication mitigates this risk by creating multiple copies of data and distributing them across multiple locations.

A classic replication approach is *primary-backup* replication [Alsberg and Day 1976]. In primary-backup replication, we designate one node to be the *primary*, with which clients interact, and all other nodes to be *backup* nodes, with which the primary interacts. To begin with, we will consider a simple primary-backup configuration with only one backup. Both the primary and the backup store a full copy of the data. The client sends requests to the primary, and in the case of a **Get** request, the backup need not be involved at all; the primary can respond to the request using its own copy of the data. In the case of a **Put** request, the client forwards the request on to the backup, which applies the change to its own state and then sends back an acknowledgment to the primary. After receiving the acknowledgment from the backup, the primary applies the update to

```
main :: IO ()
main = do
  [loc] <- getArgs
  case loc of
    "client" -> runChoreography cfg mainChoreo "client"
    "server" -> runChoreography cfg mainChoreo "server"
    _        -> error ("Unknown location: " ++ loc)
  where
    cfg = mkHttpConfig [ ("client", ("alice.some-school.edu", 3000))
                       , ("server", ("big-cloud-service.com", 4000)) ]
```

Fig. 7. Deploying the key-value store choreography with HasChor's HTTP backend

```
kvs :: Request @ "client"
    -> (IORef State @ "primary", IORef State @ "backup")
    -> Choreo IO (Response @ "client")
kvs request (primarySt, bkupSt) = do
  request' <- (client, request) ~> primary
  cond (primary, request') \case
    Put _ _ -> do
      req <- (primary, request') ~> backup
      ack <- (backup, \un -> handleRequest (un req) (un bkupSt)) ~~> primary
      return ()
    Get _ -> return ()
  response <- primary `locally` \un ->
    handleRequest  (un request') (un primarySt)
  (primary, response) ~> client
```

Fig. 8. The choreography for the primary-backup key-value store with changes to Figure 6 highlighted

its own state and finally sends a response back to the client. Therefore the client does not receive a response until *both* replicas have applied the update.[4]

Figure 8 shows an updated version of the `kvs` choreography that uses primary-backup replication.[5] As with the simple client-server setup that we saw in Figure 6, `kvs` takes a client request as its first argument. However, because the choreography now needs to keep track of states on `primary` and `backup`, it takes a pair of replica states as its second argument.

Our use of `cond` in Figure 8 illustrates HasChor's support for scrutinees of arbitrary type instead of just **`Bool`**, as discussed in Section 3.3.3. In the function passed to the `cond` expression, we pattern-match on the variants of **`Request`**. If the request is a **`Put`**, it must be forwarded to the backup, but **`Get`** requests do not need to be passed to the backup, and are handled solely by the primary.

After the `cond` has run its course, the primary handles the client request, be it **`Put`** or **`Get`**, by calling `handleRequest`, and then communicates the response to the client.

[4]We can contrast this approach with another common design in which the primary eagerly acknowledges the client's **`Put`**s while asynchronously sending an update to the backups, which trades off strong consistency among replicas in exchange for lower request latency.

[5]Other parts of the code are the same except that the configuration also needs to provide a mapping for `backup`.

```
type ReplicationStrategy a =
  Request @ "primary" -> a -> Choreo IO (Response @ "backup")

null :: ReplicationStrategy (IORef State @ "primary")
null request primarySt =
  primary `locally` \un -> handleRequest (un request) (un primarySt)

primaryBackup ::
  ReplicationStrategy (IORef State @ "primary", IORef State @ "backup")
primaryBackup request (primarySt, bkupSt) =
  cond (primary, request) \case
    Put _ _ -> do
      req <- (primary, request') ~> backup
      ack <- (backup, \un -> handleRequest (un req) (un bkupSt)) ~~> primary
      return ()
    Get _ -> return ()
    primary `locally` \un -> handleRequest (un request) (un primarySt)
```

Fig. 9. A type that characterizes replication strategy, with examples of no (`null`) and primary-backup (`primaryBackup`) replication strategies

```
kvs :: Request @ "client"
    -> a -> ReplicationStrategy a
    -> Choreo IO (Response @ "client")
kvs request states replicationStrategy = do
  request' <- (client, request) ~> primary
  response <- replicationStrategy request' states
  (primary, response) ~> client
```

Fig. 10. Key-value store choreography that takes a replication strategy, with changes to Figure 6 highlighted

4.3 Abstracting over Replication Strategies: Higher-Order Choreographies

So far, we've seen two versions of our key-value store: one with no replication, and the other with primary-backup replication. Comparing the `kvs` choreographies in Figures 6 and 8, we see a common pattern: both choreographies take a request on the client and state(s) on the server, handle the request, and return the response on the client. Indeed, from the *client's* perspective, the behavior of the key-value store should be indistinguishable, regardless of what backups are being done or not done on the server's side.

HasChor's support for higher-order choreographies lets us exploit this commonality and factor out the details of replication from `kvs`. To begin with, we define a **`ReplicationStrategy`** type as shown in Figure 9, whose values describe the details of how the primary should replicate data; since different replication strategies might keep track of different types of server state, we use a type variable `a` to represent the type of the server state.

The simple client-server key-value store from Section 4.1 does *no* replication, with the primary solely handling the request. We define this strategy as `null`, as shown in Figure 9.

On the other hand, the primary-backup key-value store from Section 4.2 needs the more sophisticated replication strategy `primaryBackup` as shown in Figure 9, which forwards **`Put`** requests

```
doBackup :: Proxy a -> Proxy b
         -> Request @ a -> IORef State @ b
         -> Choreo IO (Response @ a)
doBackup locA locB request state = do
  cond (locA, request) \case
    Put _ _ -> do
      request' <- (locA, request) ~> locB
      (locB, \un -> handleRequest (un request') (un state)) ~~> locA
      return ()
    Get _ -> return ()
```

Fig. 11. doBackup location-polymorphic choreography

```
doubleBackup ::
  ReplicationStrategy
    (IORef State @ "primary", IORef State @ "bkup1", IORef State @ "bkup2")
doubleBackup request (primarySt, backup1St, backup2St) = do
  doBackup primary bkup1 request backup1St
  doBackup primary bkup2 request backup2St
  primary `locally` \un -> handleRequest (un request) (un primarySt)
```

Fig. 12. A double-backup replication strategy

to the backup. Of course, further implementations of `ReplicationStrategy` are possible, but for now, these two versions suffice for our example.

Finally, we modify `kvs` to a function that takes a `replicationStrategy` argument, and call `replicationStrategy` to process each request, as shown in Figure 10. This refactored version of the `kvs` choreography describes the simple communication pattern between the server and the client, involving just two uses of `(~>)`. With the details of the replication strategy abstracted away, the new version of `kvs` makes it easy to see what a key-value store *does*, as far as clients are concerned: it accepts requests and produces responses.

When invoking the higher-order version `kvs` from the entry point, we pass a concrete replication strategy. That is, in the `mainChoreo` function in Figure 6, the call to `kvs` on line 30 would become `kvs request state null` if we don't want a backup, or `kvs request state primaryBackup` if we do. (In the latter case, we would also need to update `mainChoreo` to initialize the server state to the empty `Map` on both the primary and the backup.)

4.4 Abstracting over Backup Nodes: Location Polymorphism

What's better than one backup? How about two? In production environments, one primary and one backup may not be enough to satisfy data durability requirements. Indeed, distributed data storage systems such as Hadoop default to a replication factor of three [Shvachko et al. 2010]. Therefore, we might wish to implement a *double-backup* replication strategy for our key-value store. This strategy is similar to the primary-backup approach of Section 4.3, but it replicates data to two backup locations to further improve durability.

A naive implementation of double-backup replication would repeat the part of the choreography that forwards the request from the primary to the backup, using the same code for forwarding to

the second backup. However, with HasChor's location polymorphism, we can abstract away the shared logic into a location-agnostic choreography.

Figure 11 defines a `doBackup` choreography that involves two abstract locations: `locA` and `locB`. `doBackup` takes a request at `locA` and state at `locB` as arguments, examines the type of the request at `locA`, and handles the request at `locB` if it is a `Put` request.

The `doubleBackup` function in Figure 12 implements our double-backup replication strategy using `doBackup`. It calls `doBackup` twice with different backup locations, `bkup1` and `bkup2`. We could also refactor `primaryBackup` to call `doBackup` once, and we can easily extend it to any number of backup locations. We could further generalize our approach to support replication strategies with different topologies, such as chain replication [van Renesse and Schneider 2004].

5 IMPLEMENTATION

In this section, we turn our attention to the implementation of the HasChor library itself. The implementation is centered around two monads: `Choreo` (Section 5.3), for choreographies, and `Network` (Section 5.4), for network programs. Both monads are implemented as a *freer monad* instantiated with an effect signature that describes the effectful operations the monad provides, so we begin with a short primer on freer monads (Section 5.1). We also discuss our implementation of located values (Section 5.2). Finally, we present our implementation of endpoint projection (Section 5.5), the central concept of choreographic programming that links up `Choreo` and `Network`.

5.1 Freer Monads

We start with the freer monad [Kiselyov and Ishii 2015], which is defined as follows:

```
data Freer f a where
  Return :: a -> Freer f a
  Do     :: f b -> (b -> Freer f a) -> Freer f a
```

A freer monad `Freer f a` represents an effectful computation that returns a result of type `a`. The parameter `f :: * -> *` is an effect signature that defines the effectful operations allowed in the computation. `Return r` denotes a pure computation that returns a value `r` of type `a`. `Do eff k` denotes an effectful computation: the first argument `eff :: f b` is the effect to perform, and it returns a result of type `b`; the second argument `k :: b -> Freer f a` is a continuation that specifies the rest of the computation given the result of the performed effect.

The first advantage of using freer monads is that they free us from defining boilerplate monad instances. `Freer f` is a monad given any effect signature `f`:

```
instance Monad (Freer f) where
  return = Return

  (Return r) >>= f = f r
  (Do eff k) >>= f = Do eff (k >=> f)
```

The monadic return simply corresponds to the `Return` constructor. The monadic bind operator directly applies the follow-up monadic computation to a pure computation or chains together the follow-up monadic computation with the continuation of the effectful computation.[6]

The second advantage of using freer monads is that they separate the interface and implementation of effectful computations. A freer monad by itself doesn't assign any meaning to effects; it merely accumulates them as a term. To interpret effects in a freer monad, we define an `interpFreer` function that interprets the effects in a freer monad in terms of another monad:

[6]The `(>=>)` operator is the Kleisli composition and has type `(a -> m b) -> (b -> m c) -> a -> m c`.

```
interpFreer :: Monad g => (forall a. f a -> g a) -> Freer f a -> g a
interpFreer handler (Return r) = return r
interpFreer handler (Do eff k) = handler eff >>= interpFreer handler . k
```

An effect handler, of type `forall a. f a -> g a` for some monad `g`, maps effects described by `f` to monadic operations in `g`. `interpFreer` takes such an effect handler and folds it over a freer monad: for `Return r`, we simply return `r` in the monad without using the effect handler; for `Do eff k`, we use the effect handler `handler` to interpret the effect `eff`, then bind the result to the continuation `k` and recursively interpret the result of running the continuation.

To use an effectful operation in the freer monad, we lift it into the monad by wrapping the effect in the `Do` constructor with `Return` as the continuation:

```
toFreer :: f a -> Freer f a
toFreer eff = Do eff Return
```

We use `Freer` to define the `Network` and `Choreo` monads, as we'll see in the following sections.

5.2 Located Values

Before discussing the implementation of the `Choreo` monad and endpoint projection, let's first take a look at how located values are implemented, as we will use them to introduce those two concepts. A located value `a @ l` represents a value of type `a` located at location `l`:

```
data a @ (l :: LocTy) = Wrap a | Empty
```

`a @ l` has two constructors, `Wrap` and `Empty`. `Wrap` represents a located value from location `l`'s point of view — it's just a value of type `a`. `Empty` represents a located value from locations other than `l`'s point of view — it's empty to them, and they should never try to access it.

Internally, we provide two functions `wrap` and `unwrap` to create and use a located value:

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

unwrap :: a @ l -> a
unwrap (Wrap a) = a
unwrap Empty    = error "Should never happen for a well-typed choreography!"
```

The `wrap` function simply calls the `Wrap` constructor. The `unwrap` function unwraps a located value. It is erroneous to unwrap an empty located value, as this represents a location accessing a value that doesn't reside on it. As a result, the `unwrap` function cannot be directly exposed to users. Instead, users access it through the argument of type `Unwrap l` in the function passed to `locally` (Section 3.3.1). When using `unwrap` internally, we carefully avoid unwrapping an empty value.

5.3 The Choreo Monad

We are now ready to discuss the implementation of the `Choreo` monad, which represents choreographies. It is defined as a `Freer` monad instantiated with the below `ChoreoSig` effect signature:

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

data ChoreoSig m a where
  Local :: Proxy l -> (Unwrap l -> m a) -> ChoreoSig m (a @ l)
  Comm  :: Proxy l -> a @ l -> Proxy l' -> ChoreoSig m (a @ l')
  Cond  :: Proxy l -> a @ l -> (a -> Choreo m b) -> ChoreoSig m b

type Choreo m = Freer (ChoreoSig m)
```

`ChoreoSig` is parameterized by an underlying monad `m` that represents local computations, i.e., computations that are run with `locally` (Section 3.3.1). Each constructor of `ChoreoSig` corresponds to one of the three main language constructs we introduced in Section 3.3. In fact, the language constructs are merely these effects lifted into the `Choreo` monad:

```
locally :: Proxy l -> (Unwrap l -> m a) -> Choreo m (a @ l)
locally l m = toFreer (Local l m)

(~>) :: (Proxy l, a @ l) -> Proxy l' -> Choreo m (a @ l')
(~>) (l, a) l' = toFreer (Comm l a l')

cond :: (Proxy l, a @ l) -> (a -> Choreo m b) -> Choreo m b
cond (l, a) c = toFreer (Cond l a c)
```

A `Choreo` computation can be directly executed as a single-threaded local program with the `runChoreo` function:

```
runChoreo :: Monad m => Choreo m a -> m a
runChoreo = interpFreer handler
  where
    handler :: Monad m => ChoreoSig m a -> m a
    handler (Local _ m)  = wrap <$> m unwrap
    handler (Comm _ a _) = return $ (wrap (unwrap a))
    handler (Cond _ a c) = runChoreo $ c (unwrap a)
```

The `runChoreo` function interprets a `Choreo` monad as a single monadic program, with the only intricacy being that we need to appropriately wrap and unwrap values. `Local l m` is interpreted as just running the local computation, `m`, `Comm s a r` is interpreted as directly returning the value being communicated, `a`; and `Cond l a c` is interpreted as directly applying the rest of the choreography `c` to the scrutinee `a`.

5.4 The Network Monad

The `Network` monad represents programs with explicit message sends and receives, which we call *network programs*, and is the target of endpoint projection. The `Network` monad relies on a message transport backend to carry out message sends and receives, such as the HTTP backend discussed earlier in Section 3.4. Happily, because free monads separate the interface and implementation of an effectful computation, defining and using a new backend is as simple as defining and calling a new interpretation function.

The `Network` monad is defined as a `Freer` monad instantiated with the following `NetworkSig` effect signature:[7]

```
data NetworkSig m a where
  Run   :: m a -> NetworkSig m a
  Send  :: a -> LocTm -> NetworkSig m ()
  Recv  :: LocTm -> NetworkSig m a
  BCast :: a -> NetworkSig m ()

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

[7]We assume each piece of data is an instance of `Show` and `Read` for (de)serialization and omit the instance declaration for clarity.

Like **ChoreoSig**, **NetworkSig** is also parameterized by an underlying monad `m` that represents local computations. It has four constructors: **Run** corresponds to a local computation; **Send** and **Recv** correspond to sending and receiving messages to and from a location, respectively; and **BCast** corresponds to broadcasting a message (that is, sending to all locations). As we will see shortly in Section 5.5, we use broadcasting to implement endpoint projection for the `cond` operation. We lift the above effects into the **Network** monad using `toFreer`:

```
run :: m a -> Network m a
run m = toFreer (Run m)

send :: a -> LocTm -> Network m ()
send a l = toFreer (Send a l)

recv :: LocTm -> Network m a
recv l = toFreer (Recv l)

broadcast :: a -> Network m ()
broadcast a = toFreer (BCast a)
```

Network can be implemented in a variety of ways with different message transport backends. HasChor supports this by providing a type class **Backend**:

```
class Backend config where
  runNetwork :: config -> LocTm -> Network m a -> m a
```

A message transport backend defines a configuration type that specifies how locations are mapped to network hosts and ports and a `runNetwork` function that runs a network program at a specific location. For example, in the provided HTTP backend, `runNetwork` is implemented using the Servant web API library [Mestanogullari et al. 2015]. The configuration for the HTTP backend is a map from locations to hostnames and ports, as in Figures 5 and 7. Each location runs a web server that provides an endpoint for clients to send it messages, and puts the messages into a buffer when it receives a message. A **Send** `a l` is interpreted as calling the endpoint at location `l` with message `a`, a **Recv** `l` is interpreted as taking a message from location `l` from the buffer, and a **Bcast** `a` is interpreted as a sequence of **Send**s to all locations with message `a`.

5.5 Endpoint Projection

We have finally arrived at endpoint projection (EPP), the interpretation of a choreography as the corresponding network program for a specific location. In HasChor, EPP corresponds to interpreting a **Choreo** program as a **Network** program with respect to a term-level location. The function `epp` implements EPP in HasChor, and it is shown in Figure 13.[8] The `epp` function calls `interpFreer` with a `handler` that maps each effect in **ChoreoSig** into a monadic action in **Network**:

- For effect **Local** `l m`, if the location being projected to is the same as `l`, then we `run` the local computation `m` given an `unwrap` function and `wrap` the result to a located value; otherwise we return a **Empty** located value.
- For effect **Comm** `s a r`, if the location being projected to is the same as the sender's location, `s`, then we interpret it as a `send` of `a` to the receiver location `r` and return an **Empty** located value; if it's the same as the receiver's location `r`, then we interpret it as a `recv` from `s` and `wrap` the result to a located value; otherwise, we return a **Empty** located value.

[8] The function `toLocTm :: forall (l :: LocTy). Proxy l -> LocTm` in Figure 13 turns a type-level location to a term-level location and is defined as `symbolVal` from the `GHC.TypeLits` module.

```
1  epp :: Choreo m a -> LocTm -> Network m a
2  epp c l' = interpFreer handler c
3    where
4      handler :: ChoreoSig m a -> Network m a
5      handler (Local l m)
6        | toLocTm l == l' = wrap <$> run (m unwrap)
7        | otherwise       = return Empty
8      handler (Comm s a r)
9        | toLocTm s == l' = send (unwrap a) (toLocTm r) >> return Empty
10       | toLocTm r == l' = wrap <$> recv (toLocTm s)
11       | otherwise       = return Empty
12     handler (Cond l a c)
13       | toLocTm l == l' = broadcast (unwrap a) >> epp (c (unwrap a)) l'
14       | otherwise       = recv (toLocTm l) >>= \x -> epp (c x) l'
```

Fig. 13. Endpoint projection function epp

- For effect `Cond l a c`, if the location being projected to is the same as `l`, then this is the location who's making the decision, so it broadcasts the decision to all locations and then continues projecting the branches of the `cond` expression; otherwise, the location *receives* a decision and then continues projecting the branches with the received decision.

Since all effects return a located value, if the location being projected to owns the value, for example, on lines 6 and 10 of epp, we use `wrap <$>` to create a located value, indicating that the value resides on that location. Otherwise, on lines 7 and 11 of epp, we return an `Empty` value indicating that the value doesn't reside on that location.

The concise implementation of epp closely resembles previous pen-and-paper presentations of endpoint projection [Cruz-Filipe and Montesi 2020, Fig. 9], while being executable code. Consider the meaning of `(~>)`, for instance: if you're the sender, it means `send`; if you're the receiver, it means `recv`; and if you're neither of those, it's a no-op. This semantics is at the heart of choreographic programming, and our use of `Freer` lets us express it cleanly in epp, in a way that is completely decoupled from the actual message transport backend that implements `send` and `recv`.

Not only does this decoupling make the implementation of epp elegant, it also makes HasChor developer-friendly by not mandating any particular choice of message transport mechanism, and instead enabling a well-defined way to plug in one's own transport layer, using the `Backend` interface. While our provided HTTP backend is intended for web programming, nothing else about HasChor is specific to the web setting. Alternative backends could make HasChor a viable option for programming in *any* setting in which participants communicate by message passing.

6 RELATED WORK

Choreographies emerged in the mid-2000s in the context of web services. The Web Services Choreography Model [World Wide Web Consortium 2004] and Web Services Choreography Description Language [World Wide Web Consortium 2005] specifications aimed to establish standards for global descriptions of the behavior of collections of communicating processes. These standardization efforts took place with the participation of academic experts (particularly from the session types community), and they informed and inspired research on choreographies and endpoint projection [Qiu et al. 2007; Mendling and Hafner 2005; Carbone et al. 2007, 2012; Lanese et al. 2008;

McCarthy and Krishnamurthi 2008; Corin et al. 2007], laying the foundation for practical choreographic programming; Montesi [2023] gives an overview of these developments.

While choreographies as a *specification* mechanism came earlier, Carbone and Montesi [2013] pioneered the concept of a choreographic *programming* language with their Chor language. Carbone and Montesi reason about the correctness of choreographies in terms of *multiparty session types* [Honda et al. 2008]. Endpoint projection is reminiscent of Honda et al. [2008]'s concept of projection of a global type to a local type, although at the term level rather than the type level. The related literature on session types is too vast to summarize here, but Hüttel et al. [2016] and Ancona et al. [2016] provide good surveys.

Hirsch and Garg [2022]'s Pirouette language and Cruz-Filipe et al. [2022]'s Chorλ language are the first *functional* choreographic programming languages. The focus of both these works is on the semantic foundations of functional choreographic programming, rather than on practically usable and deployable implementations. Pirouette is implemented in the Coq proof assistant and stands out for having a fully mechanized proof of deadlock freedom. Pirouette supports higher-order choreographic functions, but lacks support for location polymorphism. Chorλ supports higher-order choreographic functions and a limited form of location polymorphism, in which only top-level function definitions may be location-polymorphic. In recent work, Graversen et al. [2023] present PolyChorλ, a successor to Chorλ that supports full process polymorphism, i.e., process-polymorphic lambda expressions that are usable in arbitrary expression contexts.

Currently, the most fully-realized incarnation of choreographic programming in a practical language may be Choral [Giallorenzo et al. 2020], an object-oriented language that extends Java with choreographic programming features. In Choral, choreographies are objects, and so Choral supports higher-order choreographies in the sense that choreographic objects may contain fields that are themselves choreographic objects. Choral also supports location polymorphism (which it calls *role parameterization*); in fact, it was by porting examples of location-polymorphic Choral code to HasChor that we discovered that HasChor also enjoys support for location polymorphism. Finally, Choral, like HasChor, is designed to be independent of any particular message transport mechanism. HasChor's implementation on top of Haskell is in some ways analogous to Choral's implementation on top of Java, although, unlike Choral, HasChor is "just a library" and does not require a separate compiler.

As discussed in Section 3.3.3, the design of HasChor's `cond` trades off efficiency for programmer convenience: rather than requiring programmers to manually write synchronization code in the branches of a conditional expression, HasChor inserts the needed communication automatically. As Dalla Preda et al. [2016] observe, ensuring that all participants in a choreography remain sufficiently "aware of the evolution of the global computation" seems to involve an efficiency/ease-of-use trade-off in choreographic languages in general. One approach is to deem a choreography unprojectable if the programmer forgets to write the communication code necessary for knowledge of choice [Carbone et al. 2007, 2012; Hirsch and Garg 2022]. On the other hand, HasChor's approach of inserting communication automatically is reminiscent of the choreographic language AIOCJ [Dalla Preda et al. 2014, 2016].

However, the HasChor approach is admittedly heavy-handed in that a `cond` expression results in a broadcast to *all* participants in the choreography, whether or not they actually need to know what choice was made. Previous work on choreography *amendment* and *repair* [Cruz-Filipe and Montesi 2020; Lanese et al. 2013; Basu and Bultan 2016] involves statically analyzing choreographies and inserting only the minimum amount of communication needed. In particular, Cruz-Filipe and Montesi [2020]'s amendment analysis is based on *merging* [Carbone et al. 2012]. Because of HasChor's rather unconventional implementation approach that involves dynamic interpretation

of freer monads, choreography amendment in the traditional sense would seem difficult to accomplish in HasChor, if not impossible. It might still be possible, though, to improve the efficiency of HasChor's `cond` by providing a way to annotate choreographies with the set of locations that participate in them. We intend to investigate this idea further in future work.

Finally, choreographic programming is a close cousin of *multitier* programming [Cooper et al. 2007; Serrano et al. 2006; Murphy et al. 2007; Chlipala 2015; Serrano and Prunet 2016; Weisenburger et al. 2018] (see Weisenburger et al. [2020] for a comprehensive survey of the multitier programming literature). Multitier programming emerged in response to the complexity of web programming in the early 2000s, which required programming the *tiers* of an application in different languages (for instance, JavaScript for client-side code, Java for server-side code, and SQL for an underlying database tier). In multitier programming, one uses a single, unified language to program all of the tiers in the same compilation unit, with a compiler taking care of splitting the program into deployable units in distinct languages — a technique not unlike endpoint projection. Giallorenzo et al. [2021] offer a thoughtful exploration of the evident relationship between multitier and choreographic programming. We posit that functional choreographic languages like HasChor could be a good platform for further investigation of this relationship; after all, many multitier languages, such as the pioneering multitier languages Links [Cooper et al. 2007] and Hop [Serrano et al. 2006; Serrano and Prunet 2016], Murphy et al. [2007]'s ML5, Chlipala [2015]'s Ur/Web, and Weisenburger et al. [2018]'s ScalaLoci, are distinctly functional in nature. ML5 and ScalaLoci in particular influenced HasChor's API design, especially our notion of located values.

7 CONCLUSION AND FUTURE WORK

We presented HasChor, a framework for functional choreographic programming in Haskell. Our model for choreographic programming is based on a monad, **`Choreo`**, for expressing choreographies. Our Haskell-based implementation means that we enjoy support for higher-order choreographies and location-polymorphic choreographies — both recently-proposed features of choreographic languages — essentially for free. Furthermore, HasChor users have access to all of Haskell's libraries and tooling, bringing choreographic programming within reach of everyday Haskellers and bringing the power of Haskell within reach of choreographic programming. Finally, HasChor is an *understandable* and *usable* implementation of choreographic programming: our implementation, based on the freer monad, makes endpoint projection straightforward and at the same time enables a programmer-friendly design of swappable network backends. Although HasChor relies heavily on Haskell's monad abstraction and flexible type system, we believe the same approach could be applied to other languages as well.

HasChor is certainly no remedy for all the problems of distributed programming. In particular, two of the biggest challenges of distributed programming are *asynchrony*, in which messages take arbitrarily long to arrive at their destinations, and *faults*, in which nodes may crash and messages may be lost entirely. HasChor — like other functional choreographic languages [Hirsch and Garg 2022; Cruz-Filipe et al. 2022] — does nothing in and of itself to address these difficulties. In future work, we are interested in exploring the theory and practice of fault-tolerant and asynchronous choreographic programming, building on HasChor as a basis for experimentation.

Another avenue of future work is formalizing the semantics of HasChor, so that we can ultimately prove the correctness of endpoint projection and guarantee deadlock freedom. Since **`Choreo`** choreographies are directly runnable using `runChoreo` as well as being projectable to network programs, it may be fruitful to view the direct semantics of **`Choreo`** as a specification, and define the correctness of endpoint projection with respect to that specification.

ACKNOWLEDGEMENTS

We thank Jonathan Castello, Andrew Hirsch, Fabrizio Montesi, Patrick Redmond, and the anonymous ICFP '23 reviewers, all of whom gave valuable feedback that improved this paper. Patrick Redmond also contributed significantly to the implementation of HasChor's HTTP backend.

This material is based upon work supported by the National Science Foundation under Grant No. CCF-2145367. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

Peter A. Alsberg and John D. Day. 1976. A Principle for Resilient Sharing of Distributed Resources. In *Proceedings of the 2nd International Conference on Software Engineering* (San Francisco, California, USA) *(ICSE '76)*. IEEE Computer Society Press, Washington, DC, USA, 562–570.

Davide Ancona, Viviana Bono, Mario Bravetti, Joana Campos, Giuseppe Castagna, Pierre-Malo Deniélou, Simon J. Gay, Nils Gesbert, Elena Giachino, Raymond Hu, Einar Broch Johnsen, Francisco Martins, Viviana Mascardi, Fabrizio Montesi, Rumyana Neykova, Nicholas Ng, Luca Padovani, Vasco T. Vasconcelos, and Nobuko Yoshida. 2016. Behavioral Types in Programming Languages. *Foundations and Trends® in Programming Languages* 3, 2-3 (2016), 95–230. https://doi.org/10.1561/2500000031

Samik Basu and Tevfik Bultan. 2016. Automated Choreography Repair. In *Fundamental Approaches to Software Engineering*, Perdita Stevens and Andrzej Wąsowski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 13–30.

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

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

Marco Carbone and Fabrizio Montesi. 2013. Deadlock-Freedom-by-Design: Multiparty Asynchronous Global Programming. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Rome, Italy) *(POPL '13)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/2429069.2429101

Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 153–165. https://doi.org/10.1145/2676726.2677004

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Formal Methods for Components and Objects*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 266–296.

Ricardo Corin, Pierre-Malo Denielou, Cedric Fournet, Karthikeyan Bhargavan, and James Leifer. 2007. Secure Implementations for Typed Session Abstractions. In *20th IEEE Computer Security Foundations Symposium (CSF'07)*. 170–186. https://doi.org/10.1109/CSF.2007.29

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

Mila Dalla Preda, Maurizio Gabbrielli, Saverio Giallorenzo, Ivan Lanese, and Jacopo Mauro. 2016. Dynamic Choreographies: Theory And Implementation. Logical Methods in Computer Science, Volume 13, Issue 2 (April 10, 2017) lmcs:3263. (2016). https://doi.org/10.23638/LMCS-13(2:1)2017 arXiv:arXiv:1611.09067

Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbrielli. 2014. AIOCJ: A Choreographic Framework for Safe Adaptive Distributed Applications. *CoRR* abs/1407.0975 (2014). arXiv:1407.0975 http://arxiv.org/abs/1407.0975

W. Diffie and M. Hellman. 1976. New directions in cryptography. *IEEE Transactions on Information Theory* 22, 6 (1976), 644–654. https://doi.org/10.1109/TIT.1976.1055638

The GHC Team. 2023. 6.1.1. Controlling extensions — Glasgow Haskell Compiler 9.7.20230225 User's Guide. https://ghc.gitlab.haskell.org/ghc/doc/users_guide/exts/control.html#extension-GHC2021

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

Saverio Giallorenzo, Fabrizio Montesi, Marco Peressotti, David Richter, Guido Salvaneschi, and Pascal Weisenburger. 2021. Multiparty Languages: The Choreographic and Multitier Cases. In *35th European Conference on Object-Oriented Programming (ECOOP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 194)*, Anders Møller and Manu Sridharan (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 22:1–22:27. https://doi.org/10.4230/LIPIcs.ECOOP.2021.22

Eva Graversen, Andrew K. Hirsch, and Fabrizio Montesi. 2023. Alice or Bob?: Process Polymorphism in Choreographies. https://doi.org/10.48550/ARXIV.2303.04678

J. N. Gray. 1978. *Notes on data base operating systems*. Springer Berlin Heidelberg, Berlin, Heidelberg, 393–481. https://doi.org/10.1007/3-540-08755-9_9

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. https://doi.org/10.1145/3498684

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

Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. 2016. Foundations of Session Types and Behavioural Contracts. *ACM Comput. Surv.* 49, 1, Article 3 (apr 2016), 36 pages. https://doi.org/10.1145/2873052

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

Butler Lampson and Howard E. Sturgis. 1979. Crash Recovery in a Distributed Data Storage System. (June 1979). https://www.microsoft.com/en-us/research/publication/crash-recovery-in-a-distributed-data-storage-system/ This unpublished paper was widely circulated in samizdat.

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. https://doi.org/10.1109/SEFM.2008.11

Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. 2013. Amending Choreographies. In *Proceedings 9th International Workshop on Automated Specification and Verification of Web Systems, WWV 2013, Florence, Italy, 6th June 2013 (EPTCS, Vol. 123)*, António Ravara and Josep Silva (Eds.). 34–48. https://doi.org/10.4204/EPTCS.123.5

Jay McCarthy and Shriram Krishnamurthi. 2008. Cryptographic Protocol Explication and End-Point Projection. In *Computer Security - ESORICS 2008*, Sushil Jajodia and Javier Lopez (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 533–547.

Jan Mendling and Michael Hafner. 2005. From Inter-organizational Workflows to Process Execution: Generating BPEL from WS-CDL. In *On the Move to Meaningful Internet Systems 2005: OTM 2005 Workshops*, Robert Meersman, Zahir Tari, and Pilar Herrero (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 506–515.

Alp Mestanogullari, Sönke Hahn, Julian K. Arni, and Andres Löh. 2015. Type-Level Web APIs with Servant: An Exercise in Domain-Specific Generic Programming. In *Proceedings of the 11th ACM SIGPLAN Workshop on Generic Programming* (Vancouver, BC, Canada) *(WGP 2015)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/2808098.2808099

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

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

Tom Murphy, VII, Karl Crary, and Robert Harper. 2007. Type-Safe Distributed Programming with ML5. In *Trustworthy Global Computing, Third Symposium, TGC 2007, Sophia-Antipolis, France, November 5-6, 2007, Revised Selected Papers*. 108–123. https://doi.org/10.1007/978-3-540-78663-4_9

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

Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop: a language for programming the web 2.0. In *Companion to the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*. 975–985. https://doi.org/10.1145/1176617.1176756

Manuel Serrano and Vincent Prunet. 2016. A Glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming* (Nara, Japan) *(ICFP 2016)*. Association for Computing Machinery, New York, NY, USA, 180–192. https://doi.org/10.1145/2951913.2951916

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–10. https://doi.org/10.1109/MSST.2010.5496972

Robbert van Renesse and Fred B. Schneider. 2004. Chain Replication for Supporting High Throughput and Availability. In *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6* (San Francisco, CA) *(OSDI'04)*. USENIX Association, USA, 7.

Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 129 (oct 2018), 30 pages. https://doi.org/10.1145/3276499

Pascal Weisenburger, Johannes Wirth, and Guido Salvaneschi. 2020. A Survey of Multitier Programming. *ACM Comput. Surv.* 53, 4, Article 81 (sep 2020), 35 pages. https://doi.org/10.1145/3397495

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/

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

Received 2023-03-01; accepted 2023-06-27