

Towards Lightweight and Efficient Choreographic Cloud Services

Alex Ionescu

Chalmers University of Technology and University of
Gothenburg
Gothenburg, Sweden
ionescua@chalmers.se

Alejandro Russo

Chalmers University of Technology and University of
Gothenburg
Gothenburg, Sweden
russo@chalmers.se

Abstract

Choreographic programming provides a high-level abstraction for writing distributed computations while ensuring deadlock-freedom by construction. HasChor, a Haskell-based choreographic programming framework, offers a practical implementation of these ideas but faces limitations in handling an arbitrary number of participants and optimizing the communication of branches. In this work, we introduce CloudChor, a set of extensions that enhance HasChor’s expressiveness and efficiency through the use of *multiply-located values*, allowing choreographies to distribute and collect data across multiple participants in a structured manner. Additionally, we refine HasChor’s *branching mechanism* through a lightweight static analysis, selectively propagating branching decisions only to relevant participants, reducing communication overhead. To strengthen theoretical guarantees, we establish a *formal semantics* for HasChor and our extensions, proving deadlock-freedom. Finally, we demonstrate the practicality of our enhancements by implementing a *data clean room protocol* using CloudChor. Our contributions improve the applicability of choreographic programming to cloud-based secure data collaborations, making it a stronger candidate for real-world deployments.

CCS Concepts: • Theory of computation → Semantics and reasoning; Distributed computing models; • Security and privacy → Software and application security.

Keywords: Choreographic programming, Generic programming, Haskell, Distributed systems, Data clean rooms

ACM Reference Format:

Alex Ionescu and Alejandro Russo. 2026. Towards Lightweight and Efficient Choreographic Cloud Services. In *Proceedings of the 2026 ACM SIGPLAN International Workshop on Partial Evaluation and Program Manipulation (PEPM ’26)*, January 11–17, 2026, Rennes, France. ACM, New York, NY, USA, 24 pages. <https://doi.org/10.1145/3779209.3779537>



This work is licensed under a Creative Commons Attribution 4.0 International License.

PEPM ’26, Rennes, France

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2357-5/2026/01

<https://doi.org/10.1145/3779209.3779537>

1 Introduction

In an era where data-driven decision-making is central to industries ranging from healthcare to finance, ensuring the security, privacy, and governance of shared data has become paramount. Increasingly strict regulations, such as the General Data Protection Regulation (GDPR) [12], the Data Act [14], and the Data Governance Act [13], impose requirements on how data is accessed, shared, and analyzed—such regulatory frameworks are not limited to Europe [6, 7, 29]. These regulations aim to safeguard sensitive information while still allowing for valuable insights to be extracted from data collaborations.

Major cloud providers have recently introduced secure cloud services based on data clean rooms, e.g. [2, 16, 35]. Intuitively, a data clean room is a controlled environment where *multiple parties can collaborate on data analysis without exposing raw data to each other*. These developments mark a significant step towards secure cloud-based data collaborations, but the complexity of implementing such secure services remains an open challenge. In such contexts, Trusted Execution Environments (TEEs) are often employed as a more practical alternative to multi-party computation and fully homomorphic encryption.

Writing distributed programs using explicit send and receive communication operations can be error-prone, often leading to deadlocks. One of the main difficulties in developing distributed systems is reasoning about their global behavior—ensuring that all participants interact correctly. *Choreographic programming* [8, 27] offers a solution by allowing developers to write a single unified program (a choreography) describing the behavior of the entire distributed system. A compiler then derives the correct message-passing code for each participant (through *endpoint projection*), ensuring *deadlock-freedom by construction*. Originally conceived for web services [31, 39], choreographic programming has the potential to serve as a foundation for implementing secure cloud data services, particularly those that demand high assurance and regulatory compliance.

Despite the advantages, current choreographic programming frameworks have limitations when applied to scenarios like data clean rooms. A major challenge is the difficulty of handling an unknown number of participants. Traditional

choreographic approaches often require explicitly enumerating participants (e.g. [8, 10, 17, 18, 34]), making it cumbersome to express protocols involving arbitrary numbers of parties. This limitation complicates reasoning and forces developers to manage each interaction manually, reducing clarity and maintainability. Additionally, choreographic frameworks typically support branching either by *propagating the scrutinee to all participants* [34], effectively treating it as a broadcast, or by communicating choices in a *one-by-one fashion between participants* (e.g. [10, 17, 18]). While broadcasting ensures that all participants remain synchronized, it leads to supfluous communication when only a subset of them is affected by a decision. On the other hand, propagating branching decisions one-by-one allows for more selective communication but leads to difficulties in expressing branches across a larger number of participants. These limitations make it harder to apply choreographic programming in the implementation of cloud services.

Recently, HasChor [34] was introduced as a Haskell library for implementing and executing choreographic programs in a practical setting. It provides a high-level abstraction for writing distributed computations while ensuring deadlock-freedom through Haskell's strong type system. However, it suffers from the above-mentioned issues, namely poor ergonomics when handling arbitrary numbers of participants, and inefficient implementation of branching.

To address these shortcomings, we propose CloudChor, a set of enhancements to HasChor that improve its *expressiveness* and *efficiency*.

Our first contribution is the introduction of *multiply-located values* in Section 3, which act as "bundles" of values where each component is placed at a different location. This mechanism enables choreographies to describe computations involving arbitrary numbers of participants, allowing reasoning about multiple locations simultaneously in a structured and intuitive manner. By leveraging *generic programming* techniques [11], this enhancement is implemented without modifying HasChor's implementation.

In Section 4, we refine HasChor's branching mechanism through a lightweight *static analysis* that determines which locations are involved in a conditional choice—where we apply techniques from partial evaluation literature [22]. Instead of propagating the choice to all participants, our approach selectively communicates it to the relevant subset, reducing overhead and improving scalability.

Beyond these practical improvements, we strengthen the theoretical foundations of HasChor by introducing a formal semantics in Section 5—an aspect that had not been previously established. We provide a semantic foundation for HasChor and our extensions and prove *deadlock-freedom* based on this formalization, reinforcing the correctness guarantees of our approach.

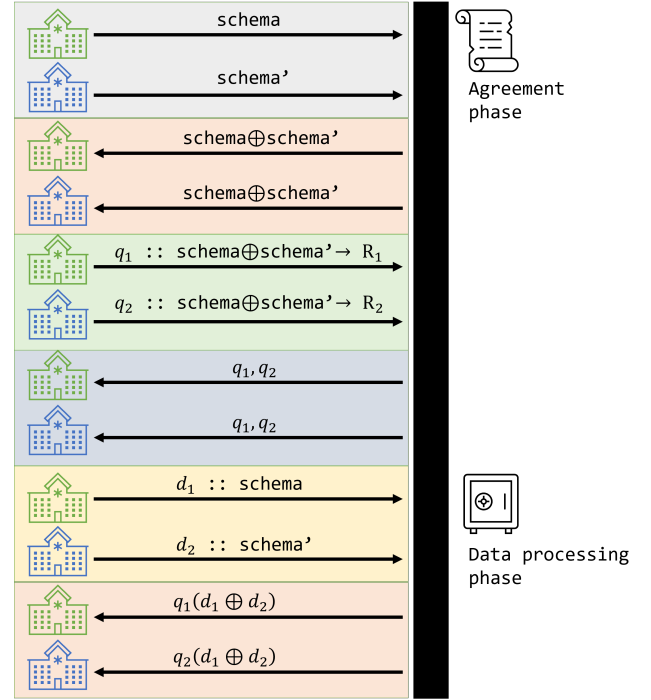


Figure 1. Overview of a data clean room protocol

1.1 Motivating Example

In the context of data clean rooms, a common application involves collaboration among healthcare organizations, e.g. hospitals. This setup allows for collaborative research and analysis while maintaining strict data privacy and compliance standards. Figure 1 shows a potential specification of a data clean room protocol. It consists of multiple phases. First, the hospitals send the schemas of the data they would like to share for aggregation purposes (schema and schema'). For simplicity, we assume that data schemas simply describe tables and their columns' types. The data clean room then combines the schemas and sends the result to the clients (schema ⊕ schema'). The hospitals proceed to specify the queries (q1 and q2) they wish to perform on the aggregated data. The server communicates these queries to other clients, asking for agreements. If all participants agree with the data that is going to be shared and the queries to perform, the protocol moves on and the clients send their datasets to the data clean room (d1 and d2). Finally, the data clean room runs all the queries on the aggregated dataset and distributes the results back to the clients.

While this high-level design of the protocol is easy to reason about, real implementations would need to deal with additional details such as synchronization, fault-tolerance, and performance. This greatly increases the complexity of implementation and makes reasoning about properties such as deadlock-freedom, type-safety, and security much more

difficult. In this work, we aim to use choreographic programming to bring the source code of a real implementation closer to the high-level protocol design, and thus provide stronger assurance about its correctness.

2 Background: HasChor

Choreographic programming [27] is a programming paradigm that enables the definition of deadlock-free distributed systems by describing the behavior of all involved machines in a single program, called a choreography, and then using Endpoint Projection to compile it to lower-level network programs specialized for each machine. HasChor [34] is a library for choreographic programming in Haskell. It provides a simple monadic interface for defining choreographies, and implements endpoint projection using Freer monads [23].

```

type a @ l
type Unwrap l = forall a. a @ l -> a
wrap :: a -> a @ l

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

```

Figure 2. The API of HasChor

The core datatype of HasChor is the choreography monad `Choreo m a`—see Figure 2 for its API. We advise the reader not to focus on the typeclass constraints, as they are mostly an implementation detail. Intuitively, the `Choreo m a` type encapsulates choreographies that produce outputs of type `a` and can perform local computations in the monad `m`. Choreographies involve multiple machines, called *locations*. The type `a @ l` represents a value of type `a` that is present at a single location `l`. By tracking the locations where values reside in their types, HasChor’s combinators can ensure that these values can be accessed exclusively by local computations performed at their respective locations. The `(~>)` combinator (referred to as `comm`) is HasChor’s communication primitive. It sends a value from a location `l` to `l'`, using the `Proxy` type [24] to make the (type-level) locations easier to specify. The `locally` primitive runs a local, effectful computation at a given location `l`. The local computation is supplied with a function of type `(forall a. a @ l -> a)`, abbreviated as `Unwrap l`, which allows it to read data that

is present only at location `l`. Lastly, the `cond` combinator implements *knowledge of choice*, allowing a choreography to branch on a value of type `a @ l` in a synchronized manner, ensuring that all locations follow the same code path. For convenience, HasChor also provides `(~>>)` and `cond'`, which combine locally with `(~>)` and respectively with `cond`, operating on the result of the local computation.

2.1 Implementation

Internally, `Choreo` is defined as a Freer monad whose signature describes the three combinators presented above:

```

type Choreo m = Freer (ChoreoSig m)
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

```

The formulation of choreographies as Freer monads allows for the easy implementation of multiple interpretations of the `ChoreoSig` signature, decoupling the syntax of choreographies from their semantics.

The simplest interpreter provided by HasChor is a trivial embedding of `Choreo m a` directly into `m`:

```

1 runChoreo :: Monad m => Choreo m a -> m a
2 runChoreo = interpFreer handler
3 where
4   handler :: Monad m => ChoreoSig m a -> m a
5   handler (Locally l m) = wrap <$> m unwrap
6   handler (Comm l a _) = return $ wrap (unwrap a)
7   handler (Cond l a c) = runChoreo $ c (unwrap a)

```

The function `runChoreo` maps each choreographic primitive directly to an effect in the `m` monad. Line 5 shows how the interpretation of `locally` provides the local computation with the library-internal function `unwrap :: a @ l -> a`. In line 6, communication is interpreted as a simple relabeling of the value of type `a @ l` into a value of type `a @ l'`. Finally, line 7 simply applies the scrutinee value to obtain the branch to be executed (`c (unwrap a)`).

2.2 Endpoint Projection

The second—and most important—interpreter for choreographies implements *endpoint projection* (EPP), which consists of translating the choreography into a series of lower-level *network programs* specialized for each location.

Similarly to `Choreo`, network programs in HasChor are represented by the `Network` monad, which is implemented as a Freer monad. However, the operations supported by the `Network` monad are of a lower level, lacking the precise location-aware typing of choreographies:

```

run    :: m a -> Network m a
send   :: Show a => a -> LocTm -> Network m ()
recv   :: Read a => LocTm -> Network m a
bcast  :: Show a => a -> Network m ()

```

The `send` and `recv` operators enable explicit one-to-one communication, while `bcast` sends a value to all other participants, and `run` performs a local computation at the current location.

Endpoint projection is then implemented as a Freer monad interpreter that maps the choreographic primitives to their network-level equivalents, specialized to a current location:

```

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 (Comm s a r)
6     | toLocTm s == l && toLocTm s == toLocTm r =
7       return $ wrap (unwrap a)
8     | toLocTm s == l = do
9       send (unwrap a) (toLocTm r) >> return Empty
10    | toLocTm r == l = wrap <$> recv (toLocTm s)
11    | otherwise = return Empty
12   handler (Cond l' a c)
13     | toLocTm l' == l =
14       bcast (unwrap a) >> epp (c (unwrap a)) l
15     | otherwise = recv (toLocTm l') >> \x -> epp (c x) l
16   handler (Locally l' m)
17     | toLocTm l' == l = wrap <$> run (m unwrap)
18     | otherwise = return Empty

```

The handler performs a pattern-match not only on the choreography `c`, but also on the projected location `l`. For instance, at the sending location `Comm` is projected to a `send` (lines 8–9), while at the destination it is projected to a `recv`, whose result is then wrapped into a located value (line 10). At all other locations, `Comm` produces an `Empty` located value (line 11)—the equivalent of a “no-op”. In the case where the sender and receiver are the same, the value is simply re-wrapped and no communication is performed (lines 6–7). The branching combinator `Cond` is projected to a broadcast at the sender, i.e. the location holding the scrutinee (lines 13–14), and to a `recv` everywhere else (line 15). In both cases, the branching choreography `c` is then supplied the broadcasted value and recursively endpoint-projected. Similarly, `Locally` only performs the local computation when projected at the specified location (line 17), and becomes a no-op everywhere else (line 18).

Network programs can themselves be interpreted in multiple ways. HasChor provides a `Backend` typeclass encapsulating this, and includes two backends: one to execute network programs locally, using channels for communication, and one for distributed execution, using HTTP for communication. Users can also implement their own backends by defining new instances of the typeclass.

2.3 Limitations

Fixed number of participants. While HasChor provides useful tools for choreographic programming, its API makes it cumbersome to describe choreographies where the locations are unknown at compile-time or large in number, such as the data clean room protocol. This is because its notion of located values focuses on single locations, and its combinators only facilitate one-to-one interactions. This limitation is not exclusive to HasChor, but is also present in other choreographic programming approaches, e.g. *Pirouette* [18]. For designing complex protocols, we need the ability to program the interactions between multiple locations (e.g. a server and many clients) in a compact and convenient manner, even when the exact number of participants is unknown at compile-time. We present our solution to this problem in Section 3.

Broadcasting branching decisions. When branching using the `cond` combinator, HasChor propagates the scrutinee to *all* locations, even in situations where only a subset of them is involved in the branching computation. This leads to unnecessary communication overhead, especially for choreographies where many branches occur independently between different subsets of locations (e.g. between a server and each of its clients). HasChor’s authors mention “... *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* ...”. [34, p. 22] In Section 4, we show that such annotations are not needed, and a simple static analysis suffices to optimize `cond`.

Unsoundness. The formulation of HasChor’s `Choreo m` monad also presents a soundness issue—which is not surprising given that HasChor’s authors focus on the implementation rather than a formalization of their approach. When the monad `m` (intended for *local* effects) is instantiated with monads that permit unrestricted IO or short-circuiting of computations (such as `Either`), the deadlock-freedom guarantees provided by choreographic programming can be circumvented, for example by locations establishing communication channels untracked by the `Choreo` monad. Two concrete counter-examples are provided in Appendix A. The usage of `SafeHaskell` [37] together with restricted-IO monads (e.g. [1, 32, 36]) is sufficient to avoid these issues, but we will not explore them further in this work.

CloudChor extensions. To alleviate HasChor’s limitations and to make it feasible to express protocols such as data clean rooms as choreographies, we propose two extensions to HasChor: *multiply-located* values with associated combinators for one-to-many, many-to-one, and pointwise communication (Section 3); an optimized `cond_` that uses static analysis to propagate branching choices only to the relevant parties (Section 4). We note that the only extension requiring modifications to HasChor’s core implementation is the `cond_` combinator. Multiply-located values and their

associated combinators are implemented purely in terms of HasChor's existing primitives, using generic programming techniques [11].

3 Multiply-Located Values

Our first proposed addition to the programming model of HasChor is the concept of *multiply-located values*, which enables users to succinctly describe choreographies with an arbitrary number of participants. The single-location type constructor $(@)$ represents values found at *one* specific location. However, when dealing with choreographies involving numerous participants, it is essential to reason simultaneously about multiple locations in a simple manner. To this end, we introduce the type of multiply-located values, $a @@ ls$, which consists of a "bundle" of values, each located at one of the locations in the list ls .

Leveraging generic programming as provided by the library `sop-core` [11], we define $(@@)$ as an N-ary product (or a heterogeneous list) of singly-located values:

```
type a @@ ls = NP ((@) a) ls
```

Intuitively, the above definition maps the partially-applied type constructor for a located value $a @ l$ over each location l in ls , resulting in a tuple of shape $(a @ l_1, a @ l_2, \dots, a @ l_n)$. To use multiply-located values effectively, we also propose a number of combinators for communication between multiple locations. Thanks to the formulation of $(@@)$ in terms of $(@)$, all of the following combinators are implemented in terms of HasChor's existing communication primitives, *without requiring modifications to the library*.

3.1 One-to-Many and Many-to-One Communication

The first two combinators that involve multiply-located values are "scatter" $(\sim>*)$, which sends a value from a single location to many others, *producing* a multiply-located value; and "gather" $(*\sim>)$, which *collects* values from different locations and sends them all to a single destination.

```
(~>*) :: (KnownSymbol l, All KnownSymbol ls, Show a, Read a)
      => (Proxy l, a @ l) -> NP Proxy ls
      -> Choreo m (a @@ ls)
```

```
(*\sim>) :: (KnownSymbol l, All KnownSymbol ls, Applicative m,
            Show a, Read a)
          => a @@ ls -> Proxy l -> Choreo m (NP (K a) ls @ l)
```

The "scatter" $(\sim>*)$ operator sends a value located at l to all locations in the list ls , producing a multiply-located value $a @@ ls$ as output. The sending location is specified by the first `Proxy l` parameter, while the recipients are specified as an N-ary product of `Proxy` values ($NP \text{ Proxy } ls$). Its implementation is conceptually simple: iterate over the list of destinations (i.e. the N-ary product of `Proxy` values), apply HasChor's $(\sim>)$ combinator to each one, and collect the results back into an N-ary product. To achieve this in

a generic programming style, we rely on the `htraverse'` combinator provided by the `sop-core` library:

```
(l, a) ~>* ls =
  hctraverse' (Proxy @KnownSymbol) ((l, a) ~>) ls
```

The primitive `hctraverse'` maps the partially applied function $((l, a) \sim>)$ over the N-ary product ls , while requiring that each of its input satisfies the `KnownSymbol` constraint. The type constraint `All KnownSymbol ls` demands exactly that, over all types in ls . To help with type inference, we mention this constraint explicitly in the application of `hctraverse'`, through `Proxy @KnownSymbol`.

The "gather" $(*\sim>)$ operator provides the dual operation to $(\sim>*)$: it collects a list of values from different locations to a single destination. Its return type, $NP (K a) ls @ l$ (ignoring the `Choreo` monad), requires further explanation. Conceptually, $(*\sim>)$ produces a list of all values received by the destination location l . However, expressing this as a simple list $[a] @ l$ loses the type-level information about the length of the list, and where the data came from—thus making it difficult to perform further communications with the locations the data has been collected from (as it occurs in the data clean room protocol, for example). To avoid losing this information, we express the output as an N-ary product parameterized in the same type-level list ls . This N-ary product should not contain located values (i.e. of type $a @ l$), but simple values of type a (since the whole product is located at l). We instantiate `NP` with the constant functor `K a b`, the type-level equivalent of the constant function, which simply ignores its second type argument. The output type $NP (K a) ls$ can be thought of as a N-ary product of values of type a , where ls 's content is phantom—i.e. when applied to any location l , $K a l$ is equivalent to a value of type a .

Functionally speaking, $(*\sim>)$ traverses over the multiply-located value $as :: a @@ ls$, forwarding each element, in turn, to the target location l :

```
as *~> l = do
  as' <- hctraverse' (Proxy @KnownSymbol)
    (\a -> K <$> ((Proxy, a) ~> l)) as
  locally l (\unwrap -> pure $ hmap (mapKK unwrap) as')
```

To accomplish this, it employs `hctraverse'` to perform the $(\sim>)$ operation at every location in ls via $(Proxy, a) \sim> l$. This process yields an N-ary product as' of values that are all located at l , each having type $K (a @ l) l'$ (for some l' in ls). All that is needed to turn this into the desired output type is to "re-wrap" the list, i.e. turn it from a list of located values at l into a list of values wholly located at l . Doing this requires access to an `unwrap` function, and so must be done locally at l . The `unwrap` function is mapped over each element in the list using `mapKK` and `hmap` (the equivalents of `map` for the constant functor and N-ary products, respectively).

3.2 Sending Different Values to Different Locations

As presented before, the "scatter" combinator ($\sim>*$) sends a single value to n locations, duplicating it n times. In other scenarios, the source location may want to send *different* values to each destination, for example to send back the results of a "gather" ($*\sim>$) after processing the data locally. This pattern is enabled by the "pointwise" ($\sim>.$) combinator:

```
(~>.) :: (KnownSymbol l, All KnownSymbol ls, Applicative m
        , Show a, Read a)
      => (Proxy l, NP (K a) ls @ l)
      -> NP Proxy ls -> Choreo m (a @@ ls)
```

This combinator takes as input an N -ary product located at l (of type $(NP (K a) ls) @ l$)—as discussed when presenting the gather operator—which tracks the provenance of the data. Each element of the product, of type $K a l'$ (where l' is in ls), is sent to its corresponding location l' , i.e. the combinator sends the values in a "pointwise" manner to their respective destinations. A detailed explanation of the implementation of ($\sim>.$) can be found in Appendix B.

3.3 Running Local Computations at Many Locations

The final combinator we provide, `multilocally`, is a multiply-located analogue of `locally`. It runs *the same local computation* at each location in a list ls , collecting their results into a multiply-located value.

```
type Unwraps ls = forall a. a @@ ls -> a

multilocally :: All KnownSymbol ls => NP Proxy ls
              -> (forall l. KnownSymbol l
                  => Proxy l -> Unwraps ls -> m a)
              -> Choreo m (a @@ ls)
```

The local computation is provided with a `Proxy l` value indicating the current location and an unwrapping function (of type `Unwraps ls`) that allows it to project out data from multiply-located values of type $a @@ ls$. The implementation of `multilocally` is more involved than the other combinators, as it needs to construct appropriate `Unwraps` functions for each location involved—see Appendix B for details. We also provide variants of ($\sim>*$) and ($*\sim>$) that combine the operators with `locally` (respectively `multilocally`), named ($\sim\sim>*$) and ($*\sim\sim>$). Appendix C provides the full API of our proposed extensions, and the full source code is included in the supplementary material [21].

3.4 Data Clean Room Protocol as a Choreography

Using `CloudChor`, we present in Figure 3 a Haskell implementation of the data clean room protocol described in Section 1.1. The `cleanRoom` choreography makes extensive use of multiply-located values and the combinators presented above. In the first phase (lines 5–9), clients load their data schemas and send them to the server using ($*\sim\sim>$), and the server communicates a merged schema back using ($\sim>*$)—we omit the details of schema- and data-processing functions,

```
1 cleanRoom :: (KnownSymbol server, All KnownSymbol clients)
2           => Proxy server -> NP Proxy clients
3           -> Choreo IO ()
4 cleanRoom server clients = do
5   -- First phase: Server aggregates data schemas
6   schemasC <- (clients, \c _ -> loadSchema c) *\sim\sim> server
7   schemasS <- locally server $ \unwrap ->
8     pure $ mergeSchemas $ unwrap schemasC
9   finalSchema <- (server, schemasS) ~>* clients
10
11  -- Second phase: Clients specify queries to run
12  clientQueries <- multilocally clients $ \c unwrap ->
13    loadQuery c $ unwrap finalSchema
14  clientQueriesS <- clientQueries *\sim> server
15
16  queriesS <- locally server \unwrap ->
17    pure $ hcollapse $ unwrap clientQueriesS
18  queries <- (server, queriesS) ~>* clients
19
20  -- Third phase: Server checks for agreement on queries
21  agreements <- multilocally clients \c unwrap ->
22    agreeWithQueries c $ unwrap queries
23  agreementsS <- agreements *\sim> server
24  proceed <- locally server $ \unwrap ->
25    pure $ allAgree $ unwrap agreementsS
26
27  -- Final phase: Server runs queries if everyone agrees
28  cond_ (server, proceed) $ \case
29    False ->
30      multilocally clients (\_ _ -> print "Disagreement")
31    True -> do
32      datasets <- getDatasets 5 schemasS server clients
33      resultsS <- locally server $ \unwrap ->
34        runQueries (unwrap queries) (unwrap datasets)
35
36      results <- (server, resultsS) ~>. clients
37      multilocally clients $ \_ unwrap ->
38        print $ unwrap results
```

Figure 3. Data clean room protocol as a choreography

as our focus here is on the choreographic structure. In the second phase (lines 11–18), clients communicate the queries they wish to run, and the server broadcasts this information so all clients of the others' queries. The use of `hcollapse` in line 15 is required to convert the length-indexed `NP` list obtained from ($*\sim\sim>$) into a regular list—the reasoning for which is described earlier in this section. In the agreement phase (20–25), clients `multilocally` decide whether they wish to continue the protocol, and the server aggregates these decisions to make a final choice. Finally, the server notifies clients of this choice using `cond_`. If anyone disagreed, the protocol ends (lines 29–30). Otherwise, it proceeds with the clients sending their datasets to the server, who then

runs all the queries and distributes back the results (lines 31–38). We describe `cond_` and the error-handling performed by `getDatasets` in Section 4.

3.5 Specifying Participants at Runtime

There is one more aspect to consider for running choreographies with arbitrary participants. The representation of locations as Proxies of type-level strings normally requires their names to be known at compile-time, but more complex scenarios might require executing a choreography with different sets of participants each time, and recompiling it for each run is unfeasible. In this section, we present the "wrapper" code used to execute the `cleanRoom` choreography with dynamically-known participants.

Revisiting `cleanRoom`'s type signature from Figure 3, what first needs to be done is to instantiate the type parameters `server` and `clients`, and construct appropriate proxies for them. Because locations' names only become known at runtime, a conversion is required from term-level strings to type-level Symbols (which is how locations are represented). To perform this conversion, we define the following function:

```
withProxy :: String
  -> (forall l. KnownSymbol l => Proxy l -> r) -> r
withProxy str k =
  withSomeSSymbol str $ \ll :: SSymbol l ->
    withKnownSymbol ll (k @ Proxy)
```

`withProxy` is written in continuation-passing style, which makes this kind of type-level programming in Haskell easier. The continuation `k` allows computations to depend on the type-level representation of the input string. Specifically, the function `withSomeSSymbol` is used to convert the runtime string into an *existentially quantified* `SSymbol`—a *singleton* type that uniquely associates a value with a type. The lambda expression introduces a scoped type variable `ll`, and uses `withKnownSymbol` to satisfy the continuation's `KnownSymbol` constraint with `l`. Building on `withProxy`, we can also define a function `withProxies` to iterate over a list of strings and convert it into an N-ary product of proxies. For brevity, we omit its implementation here.

To run the choreography using the HTTP backend, we construct an `HttpConfig` value that maps each location to an IP and port.

```
httpConfig :: [String] -> HttpConfig
httpConfig ls = mkHttpConfig $ zipWith nodes ls [3000..]
  where nodes l port = (l, ("localhost", port))
```

For simplicity, we execute all locations on the same machine at consecutive ports, but a more complex mapping could instead be created by e.g. reading a JSON configuration file. We tie everything together with a short `main` function, which parses the list of locations, as well as the current location `self`, from command-line arguments:

```
main :: IO ()
main = do
```

```
self : ls@(server : clients) <- getArgs
runChoreography (httpConfig ls) choreo self
  where choreo = withProxy server $ \s ->
    withProxies clients $ \cs -> cleanRoom s cs
```

4 Improving the Knowledge of Choice

In this section, we address an issue in HasChor's implementation of *knowledge of choice* [10]. To ensure that all participants remain in sync, HasChor's `cond` operator (shown in Figure 2) broadcasts branching choices to *all* locations. However, in many cases, only a subset of locations is involved in the branching computation, so broadcasting the choice to everyone is wasteful. To fix this limitation, we propose a new combinator, `cond_`, which relies on static analysis to determine the subset of locations involved in the choice, and only communicate with them.

4.1 Adapting the Type Signature of `cond`

Recalling `cond`'s type signature (Figure 2), it accepts a scrutinee value, located at `l` (the *sender*), which is broadcasted to everyone else; and a function that, based on the broadcasted value, produces a choreography whose final result is `b`. Because the provided function is executed at *every* location, the value it produces is present everywhere, so its output is a "global" value `b`, rather than a located value like `b @ l`.

This is exactly the invariant that our proposed `cond_` aims to eliminate. Because `cond_` is *not* meant to be executed by every location, its output type can no longer be just `b`—it has to be a located value, but located *where*? The only location that is guaranteed to have a copy of the scrutinee is the *sender*, while all other participants in essence just mimic the sender's branching decision to remain in sync. This is reflected in `cond_`'s type signature, which produces a final value located at the sender `l`:

```
cond_ :: (Bounded a, Enum a, Show a, Read a, KnownSymbol l)
  => (Proxy l, a @ l)
  -> (a -> Choreo m (b @ l)) -> Choreo m (b @ l)
```

4.2 Static Analysis

To find out which locations are actually involved in a `cond_`'s body, we rely on a simple static analysis of the choreography. This analysis is implemented as a function `participants` which, given a choreography as input, produces a set of its involved locations.

```
1 participants :: [LocTm] -> Choreo m a -> Set LocTm
2 participants ls = fromMaybe (Set.fromList ls)
3   . flip execStateT Set.empty . interpFreer handler
4   where
5     handler :: ChoreoSig m a -> StateT (Set LocTm) Maybe a
6     handler = \case
7       Locally l _ -> modify' (insertLoc l) $> Empty
8       Comm l _ l' ->
9         modify' (insertLoc l . insertLoc l') $> Empty
10      Cond _ _ -> modify' (insertLocs ls) *> empty
```

```

11 Cond_ l _ c -> do
12   modify' (insertLoc l . Set.union (enumerate ls c))
13   pure Empty

```

It achieves this by interpreting the ChoreoSig effect and embedding it into a StateT monad that collects a Set of encountered locations. In the Locally and Comm cases (lines 7–9), it just collects the locations mentioned by the constructors, using the modify' function to update the state. The helper insertLoc is used to insert new locations into the set. The effect interpreter is not actually running the choreography, but it nevertheless needs to produce a result for each interpreted operation, as required by the Freer monad abstraction—see Section 2.1. For that, it uses the Empty constructor (an implementation detail that is not exposed in HasChor’s API) to create empty (dummy) located values.

In the case of Cond, which by definition involves everyone, the static analysis collects *all* locations (represtend by ls) and short-circuits (using the Maybe monad, and empty from Control.Applicative—not to be confused with Empty).

The handling of the newly-introduced Cond_ (lines 11–13) is more involved. Here, we employ *the trick*—a partial evaluation technique [22] that has been used for domain specific language design [30, 38]. The trick works by generating a list of all possible values of type a, and applying the provided function (the body of the Cond_) to each of them, recursively collecting the participants of the resulting choreographies. This is encapsulated in the enumerate function:

```

enumerate :: (Bounded a, Enum a)
=> [LocTm] -> (a -> Choreo m b) -> Set LocTm
enumerate ls c = foldMap (participants ls . c) [minBound ..]

```

The usage of enumerate requires additional constraints on the type a: it must be Bounded and Enumerable. The implementation is simple: the list [minBound ..] generates all possible input values of type a, and by using foldMap, it obtains the corresponding branch for each value and the participants involved in it (participants ls . c), computing the union of all such sets.

Using *the trick*, however, comes with some limitations. Firstly, the static analysis has to be repeated for every possible value of type a. This is inefficient if the domain of a is large, and could lead to an exponential blow-up if the choreography includes deeply-nested applications of cond_. And secondly, if cond_ is used within a recursive function with no recursion limit, the static analysis will enter an infinite loop, repeatedly attempting to analyze the recursive calls. In practice, we found that these limitations do not pose major problems in most situations, since nested and unbounded-recursive conds are rarely needed, and most conditionals branch on small-domain values such as booleans.

4.3 Endpoint Projection

We update HasChor’s endpoint projection procedure to handle the translation of cond_, performing the static analysis

presented above, such that the sender in a cond_ only communicates with its involved locations:

```

1 epp :: [LocTm] -> Choreo m a -> LocTm -> Network m a
2 epp ls c l = interpFreer handler c
3   where
4     handler :: ChoreoSig m a -> Network m a
5     ...
6     handler (Cond_ l' a c)
7       | toLocTm l' == l = do
8         bcast (unwrap a) (Set.toList $ Set.delete l ls')
9         epp ls (c (unwrap a)) l
10      | Set.member l ls' = recv l' >=> \x -> epp ls (c x) l
11      | otherwise = pure Empty
12      where ls' = enumerate ls c

```

The above snippet extends the definition of epp with a handler for Cond_, which enumerates the provided function to determine its participants (line 12). The sender broadcasts the value a to the computed participants ls' (except itself), then proceeds to execute c. (lines 7–9). The other involved participants perform a recv to get the broadcasted value, similarly continue executing c (line 10); while uninvolved locations do *not* execute c, simply producing an Empty located value (line 11). Handling Cond_ also required slight modifications to HasChor’s Network and Backend layers, changing the bcast operator to accept a list of target locations, and adding a method to the Backend typeclass to query the full list of locations present in the current execution—information that any backend can easily provide.

4.4 Efficient Failure Recovery via cond_

When designing complex protocols and choreographies, there is often a need to perform error-handling at the business-logic level. In this section, we show how cond_ can be used to implement such error-handling in a way that is both performant and maintainable.

The error condition we are going to consider is that of clients in the data clean room protocol sending datasets that do not conform to the agreed-upon schema. This would likely lead to errors when evaluating other clients’ queries, so the server must ensure all received datasets are schema-conformant. The check for this condition can be encapsulated in a sub-choreography like the following:

```

1 getDataset :: (KnownSymbol server, KnownSymbol client)
2   => Int -> Schema @ server -> Proxy server
3   -> Proxy client -> Choreo IO (Dataset @ server)
4 getDataset tries schema server client =
5   if tries <= 0 then
6     pure $ wrap emptyDataset
7   else do
8     ds <- (client, \_ -> loadDataset client) ~> server
9     conforms <- locally server $ \unwrap ->
10      conformsToSchema (unwrap schema) (unwrap ds)
11     cond_ (server, conforms) $ \case
12       True -> pure ds
13       False -> getDataset (tries - 1) schema server client

```


First, the client loads their dataset and sends it to the server (line 9). The server checks if the dataset conforms to the schema and, using `cond_`, branches on the boolean result together with *only* the client in question (lines 10–12). If the dataset conforms, it is returned and the sub-choreography ends (line 13). Otherwise, to repeat the process, a recursive call is made to `getDataset` (line 14). The amount of attempts is limited to a fixed number (`tries`) to prevent clients from stalling the protocol. This also helps avoid an infinite loop during the static analysis, as described in Section 4.2. The data clean room implementation presented in Figure 3 includes a call to `getDatasets`, which simply applies `getDataset` to each client.

4.5 Performance Evaluation of `cond_`

To assess the differences in performance between HasChor's broadcasting `cond` and our improved version, we performed a series of benchmarks comparing the execution speed and number of network-level Send operations performed when running choreographies with each implementation.

We chose 3 choreographies for this benchmark: 2 small examples included in HasChor's distribution (`bookseller` and `key-value-store`), to test the overhead of our static analysis when the benefit it provides is limited; and the data clean room choreography from Section 3.4, instantiated with different numbers of participants and failures-per-participant, to see how performance scales as communication increases.

The results are summarized in Table 1. The parameters of the data clean room choreography are reflected in its identifier, e.g. `clean-room-50-2` is instantiated with 50 participants, and 2 failures per participant (each participant fails to send its dataset twice before succeeding).

The execution time metrics were collected by averaging 10 runs of each choreography using HasChor's HTTP backend on a single machine (so each location corresponds to a different port on localhost), and the number of Send operations was collected by running each choreography using an instrumented version of HasChor's local backend (which can be found in the supplementary material [21]).

The benchmarks show that our optimized `cond_` greatly reduces the amount of network-level communications as the number of participants scales, especially in cases where many branches involve only a subset of locations, such as the data clean room's error recovery. Using `cond_` also leads to faster execution times, even though this difference isn't as pronounced. This shows that even on a single machine, where communication overhead is very low, the benefits of our static analysis outweigh the overhead of computing involved locations. We conjecture that on production deployments, where locations are mapped to different machines and thus network overhead is larger, the execution speed benefits of `cond_` are even greater.

5 Formal Semantics

In this section, we formalize the semantics of core HasChor, and of the newly-introduced `cond_` operator. In the following section, we will provide a proof of deadlock-freedom based on this formalization.

5.1 Local Computations

The first component of the formalization is the calculus, illustrated below. We use an expression-based functional language based on the simply-typed λ -calculus, whose syntax is separated into three categories, for local computations (e), choreographies (c), and network programs (p).

$$\begin{array}{ll} \text{Expression} ::= & e \quad \text{Local computation} \\ & | \quad c \quad \text{Choreography} \\ & | \quad p \quad \text{Network program} \end{array}$$

We are first going to focus on local expressions. This category includes identifiers, let-bindings, function abstraction and application, numeric addition, and allows expressing *local* effects in the local monad m . For simplicity, we include only the `print` effect, but the formalization holds for any effect that can be modeled using a local state, as described further below.

$$\begin{array}{ll} e ::= & \dots \text{ (standard } \lambda\text{-calculus constructions)} \\ & | \quad \text{print}_m e \mid e \gg_m e \mid \text{pure}_m e \end{array}$$

This category also includes values v , which are the unit value `()`, integers n , and located values constructed using either `Wrap` or `Empty`, as well as the special function `unwrap` (which is not part of the surface syntax).

$$\begin{array}{ll} \text{Value } v ::= & () \quad | \quad n \\ & | \quad \text{Wrap}_l v \quad | \quad \text{Empty} \\ & | \quad \text{unwrap}_l \end{array}$$

Reduction. We denote small-step *local* reduction with the (\longrightarrow) symbol. Aside from the usual reduction rules for a λ -calculus, we include a rule for reducing `unwrap`:

$$\begin{array}{l} \text{L-UNWRAP-WRAP} \\ \text{unwrap}_l (\text{Wrap}_l e) \longrightarrow e \end{array}$$

5.2 Choreographies

Choreographic expressions include communication primitives and monadic operations for composing them:

$$\begin{array}{ll} \text{Choreography } c ::= & \text{Locally } l \ u.e \\ & | \quad (l_1, e) \rightsquigarrow l_2 \quad | \quad \text{Cond } l \ e \ x.c \\ & | \quad c \gg_c c \quad | \quad \text{pure}_c e \end{array}$$

The primitive $(l_1, e) \rightsquigarrow l_2$ sends a value from location l_1 to l_2 . The primitive `Cond` $l \ e \ x.c$ broadcasts a value from location l and runs a subsequent choreography, substituting x for the broadcasted value. Lastly, `Locally` $l \ u.e$ runs a local computation at location l , providing it with a suitable `unwrap` function u . Note that the formalized language does not include derived operators such as HasChor's (\rightsquigarrow) and `cond'`, nor our multiply-located extensions. Because they

Table 1. Performance differences between cond and cond_

Choreography	Runtime (cond)	Runtime (cond_)	# of Sends (cond)	# of Sends (cond_)
bookseller	0.506 s	0.506 s	4	4
key-value-store	1.013 s	1.012 s	26	21
clean-room-10-0	2.788 s	2.780 s	180	90
clean-room-10-2	2.806 s	2.782 s	400	130
clean-room-10-4	2.821 s	2.784 s	620	160
clean-room-50-0	13.117 s	12.914 s	2900	450
clean-room-50-2	13.568 s	12.936 s	8000	650
clean-room-50-4	14.001 s	12.942 s	13100	800
clean-room-100-0	26.471 s	25.614 s	10800	900
clean-room-100-2	28.282 s	25.637 s	31000	1300
clean-room-100-4	30.067 s	25.648 s	51200	1600

are implemented in terms of the formalized primitives, the correctness of these operators follows from the correctness of the primitives.

Typing. We employ a standard type system for our calculus, which includes ground types (for simplicity, only unit () and Int), functions, located values, and the monads required to represent computations at each "layer": $m\tau$ for local computations, Choreo $m\tau$ for choreographies, and Network $m\tau$ for network programs. The full grammar for types can be found in Appendix D.

Based on this type system, the typing of choreographic primitives resembles the type signatures of their HasChor equivalents. For simplicity, we only consider the communication of integer values, but the semantics and our proofs hold for any serializable types.

As an example, the typing of Comm is described by the following rule:

$$\frac{\text{T-COMM} \quad \Gamma \vdash e : \text{Int} @ l_1}{\Gamma \vdash (l_1, e) \rightsquigarrow l_2 : \text{Choreo } m (\text{Int} @ l_2)}$$

The full listing of typing rules can be found in Appendix D.

Choreographic states. Before describing the operational semantics of choreographies, we need to introduce the concept of *choreographic states*:

$$\begin{aligned} \text{Local state } \sigma &::= \dots \\ \text{Choreographic state } \Sigma &::= (l \triangleright \sigma_l)^* \\ \text{Choreography with state } C &::= \Sigma; c \end{aligned}$$

Our formalization assumes that the monad m only performs *local* effects (recall the motivation in Section 2.3). Each location l is assigned a local monadic state σ_l to encapsulate these effects. The state of an entire choreography, Σ , is simply a collection of the states of all participating locations. We abbreviate the pairing of a choreography with its state $(\Sigma; c)$ as C .

Operational semantics. The operational semantics of choreographies express the high-level meaning of reduction for a choreography, giving a global view of the choreography's behavior, without differentiating which locations are involved in each reduction. They are defined by the small-step reduction relation (\rightsquigarrow_C) , which operates on pairs of choreographic states and expressions $(\Sigma; e)$. This relation is modeled after the runChoreo function in HasChor. The big-step reduction relation (\rightsquigarrow_C^*) is defined as the reflexive, transitive closure of (\rightsquigarrow_C) . Rule C-COMM reduces the (\rightsquigarrow_C) operator by simply re-wrapping the value at the target location. Because this rule models network communication, it requires that the value to be sent is fully evaluated. Similarly, rule C-COND substitutes the the broadcasted value into the target choreography. Rule C-LOCALLY reduces Locally by executing the effectful computation (the notation (\Downarrow_m) represents big-step monadic reduction), updating the location's state σ_l . We do not further specify the effectful semantics of m , but assume that the effects are contained only in the local state σ_l . Lastly, the rules C-BIND and C-PURE-BIND enable monadic sequencing of choreographic actions. Figure 4 includes the full listing of choreographic reduction rules.

5.3 Network Programs

While the operational semantics for choreographies model high-level behavior, in practice, choreographies are *endpoint-projected* before being executed. This section presents the syntax and semantics of *network programs*, which represent the output of endpoint projection.

$$\begin{aligned} \text{Network program } p &::= \text{Send } e \ l \quad | \text{Recv } l \\ &\quad | \text{BCast } e \ ls \quad | \text{Run } e \\ &\quad | p \gg_N p \quad | \text{pure}_N e \end{aligned}$$

Network programs are modeled after HasChor's Network monad, with a slight difference in the representation of the BCast operation. Instead of broadcasting a value to *all* locations, it only sends it to a specified list of destinations ls . This allows for its reduction (explained later) to be defined inductively. The typing of network primitives follows the type

$$\begin{array}{c}
\text{C-LOCAL} \quad \frac{c \longrightarrow^* c'}{\Sigma; c \rightsquigarrow_C \Sigma; c'} \quad \text{C-BIND} \quad \frac{\Sigma; c \rightsquigarrow_C \Sigma'; c'}{\Sigma; c \gg_C f \rightsquigarrow_C \Sigma'; c' \gg_C f} \\
\\
\text{C-PURE-BIND} \quad \Sigma; \text{pure}_C e \gg_C f \rightsquigarrow_C \Sigma; f e \\
\\
\text{C-COMM} \quad \Sigma; ((l_1, \text{Wrap}_{l_1} v) \rightsquigarrow l_2) \rightsquigarrow_C \Sigma; \text{pure}_C (\text{Wrap}_{l_2} v) \\
\\
\text{C-COND} \quad \Sigma; \text{Cond } l (\text{Wrap}_l v) x.c \rightsquigarrow_C \Sigma; c[v/x] \\
\\
\text{C-COND}_- \quad \Sigma; \text{Cond}_- l (\text{Wrap}_l v) x.c \rightsquigarrow_C \Sigma; c[v/x] \\
\\
\text{C-LOCALLY} \quad \frac{\sigma_l; e[\text{unwrap}_l / u] \Downarrow_m \sigma'_l; e'}{\Sigma[l \triangleright \sigma_l]; \text{Locally } l u.e \rightsquigarrow_C \Sigma[l \triangleright \sigma'_l]; \text{pure}_C (\text{Wrap}_l e')}
\end{array}$$

Figure 4. Reduction rules for choreographies

signatures of their HasChor equivalents—see Appendix D for the full listing.

Network systems. A network program describes the code executed at one location, but to reason about endpoint projection we need to reason about the entire *system* of all locations' network programs. A network system S consists of a parallel composition of network programs p_n , along with a local monadic state for σ_n each location l_n :

$$\begin{array}{ll}
\text{Program state } P ::= & \sigma_n; p_n \\
\text{Network system } S ::= & (l_n \triangleright P_n) \quad (\text{A single location}) \\
& | S \parallel S \quad (\text{Parallel composition})
\end{array}$$

Operational semantics. Network-level semantics are separated between two relations: *network reduction* and *system reduction*.

Network reduction specifies how an individual network program reduces, tracking in-progress communications via *transition labels*. A transition label can be empty (\emptyset) or a single message indicating either an in-progress send or recv operation. Thus, the rules N-SEND and N-BCAST-CONS produce the label $(v \rightsquigarrow l_2)$, and the rule N-RECV produces $(l_1 \rightsquigarrow v)$. Other reduction rules either produce an empty label (such as N-PURE-BIND) or propagate the label of a sub-step. Figure 5 provides the full listing of network reduction rules.

System reduction allows network systems to reduce by syncing up in-progress send and recv operations between its locations. It consists of only 3 reduction rules, shown in Figure 6. Rule S-SINGLE lifts a single network reduction step into a system step, but only if its label is \emptyset . Rule S-SYNC synchronizes a send at one location with a recv at another one, reducing both while consuming the labels. Lastly, rule

$$\begin{array}{c}
\text{N-LOCAL} \quad \frac{p \longrightarrow^* p'}{\sigma; p \rightsquigarrow_N \sigma; p'} \quad \text{N-BIND} \quad \frac{\sigma; p \rightsquigarrow_N \sigma'; p'}{\sigma; p \gg_N f \rightsquigarrow_N \sigma'; p' \gg_N f} \\
\\
\text{N-PURE-BIND} \quad \sigma; \text{pure}_N e \gg_N f \rightsquigarrow_N \sigma; f e \\
\\
\text{N-SEND} \quad \sigma; \text{Send } v l \xrightarrow{(v \rightsquigarrow l)}_N \sigma; \text{pure}_N () \\
\\
\text{N-RECV} \quad \sigma; \text{Recv } l \xrightarrow{(l \rightsquigarrow v)}_N \sigma; \text{pure}_N v \\
\\
\text{N-BCAST-NIL} \quad \sigma; \text{BCast } v [] \xrightarrow{\emptyset}_N \sigma; \text{pure}_N () \\
\\
\text{N-BCAST-CONS} \quad \sigma; \text{BCast } v (l : ls) \xrightarrow{(v \rightsquigarrow l)}_N \sigma; \text{BCast } v ls \\
\\
\text{N-RUN} \quad \frac{\sigma; e \Downarrow_m \sigma'; e'}{\sigma; \text{Run } e \xrightarrow{\emptyset}_N \sigma'; \text{pure}_N e'}
\end{array}$$

Figure 5. Reduction rules for network programs

S-DISJOINT permits the subset of a system to step independently. An important aspect of this definition is that single-location network steps "lift" to system steps only if they are \emptyset -labeled, or if their labels "cancel out" by S-SYNC. Thus, a system step leaves no unhandled messages. We also note that, unlike other constructions such as the π -calculus, our formalized language does not include features such as process spawning and higher-order communication channels.

$$\begin{array}{c}
\text{S-SINGLE} \quad \frac{P \xrightarrow{\emptyset}_N P'}{(l \triangleright P) \rightsquigarrow_S (l \triangleright P')} \quad \text{S-DISJOINT} \quad \frac{S \rightsquigarrow_S S'}{S \parallel S_1 \rightsquigarrow_S S' \parallel S_1} \\
\\
\text{S-SYNC} \quad \frac{P_1 \xrightarrow{(v \rightsquigarrow l_2)}_N P'_1 \quad P_2 \xrightarrow{(l_1 \rightsquigarrow v)}_N P'_2}{(l_1 \triangleright P_1) \parallel (l_2 \triangleright P_2) \rightsquigarrow_S (l_1 \triangleright P'_1) \parallel (l_2 \triangleright P'_2)}
\end{array}$$

Figure 6. Reduction rules for network systems

When applying a parameterized system step, such as to apply a local reduction step (N-LOCAL) at location l_1 , we use the following notation: S-DISJOINT (S-SINGLE (N-LOCAL))

at l_1 . Similarly to choreographies, we refer to the reflexive, transitive closure of (\rightsquigarrow_N) and (\rightsquigarrow_S) as (\rightsquigarrow_N^*) and (\rightsquigarrow_S^*) , respectively.

5.4 Endpoint Projection

Endpoint projection is the process of translating a choreography into a system of network programs. This is achieved by mapping the function $\llbracket c \rrbracket_l$ (which endpoint-projects choreography c at location l) over all the locations participating in the choreography. $\llbracket c \rrbracket_l$ is defined inductively over the structure of the choreography, as illustrated below. A full listing is provided in Appendix D.

$$\llbracket (l_1, e) \rightsquigarrow l_2 \rrbracket_l = \begin{cases} \text{pure}_N e & (\text{if } l = l_1 = l_2) \\ \text{Send } (\text{unwrap}_{l_1} e) l_2 \gg_N \lambda_. \text{Empty} & (\text{if } l = l_1 \neq l_2) \\ \text{Recv } l \gg_N \lambda x. \text{pure}_N (\text{Wrap}_{l_2} x) & (\text{if } l = l_2 \neq l_1) \\ \text{pure}_N \text{Empty} & (\text{otherwise}) \end{cases}$$

$$\llbracket \text{pure}_C e \rrbracket_l = \begin{cases} \text{pure}_N \text{Empty} & \text{if } (e \longrightarrow^* \text{Wrap}_{l_1} e') \wedge l \neq l_1 \\ \text{pure}_N e & \text{otherwise} \end{cases}$$

Each choreographic combinator is endpoint-projected to their appropriate network-level equivalent, depending on the role of the current location l . In the definition above, (\rightsquigarrow) is translated to a `Send` if l is equal to the sender, to a `Recv` if l is equal to the receiver, or to an `Empty` located value if l is not mentioned by the combinator. The second equation specializes the projection of `pure` in order to remove some artifacts. If the argument of `pure` locally reduces (at projection time) to a `Wrap` constructor at a location *different* to the current location l , then it is endpoint-projected to `Empty` instead.

Other choreographic combinators are projected similarly to (\rightsquigarrow) , while for all other categories of expressions, the endpoint projection function is applied homomorphically.

Finally, the endpoint projection of a choreographic state $(\Sigma; c)$ results in a network system where the program at each location l is the projection of c at l :

$$\llbracket \Sigma; c \rrbracket = (l_1 \triangleright \Sigma[l_1]; \llbracket c \rrbracket_{l_1}) \parallel \dots \parallel (l_n \triangleright \Sigma[l_n]; \llbracket c \rrbracket_{l_n})$$

5.5 Formalizing `cond_`

In this section, we extend our formalization to cover the `cond_` operator introduced in Section 4. The typing of `Cond_` (included in Appendix D) is similar to its Haskell type signature, and its high-level semantics are identical to those of `Cond` (see Figure 4). However, their endpoint projections differ.

We model the endpoint projection of `Cond_` based on the *free locations* of the target choreography (i.e. free variables

of kind `Loc`), defined as the function *freeLocs*:

$$\begin{aligned} \text{freeLocs}((l_1, e) \rightsquigarrow l_2) &= \{l_1, l_2\} \\ \text{freeLocs}(\text{Cond } l_1 e a.c) &= \text{locs}(c) \\ \text{freeLocs}(\text{Cond}_- l_1 e a.c) &= \{l_1\} \cup \text{freeLocs}(c) \\ \text{freeLocs}(\text{Locally } l_1 u.e) &= \{l_1\} \\ \text{freeLocs}(c \gg_C f) &= \text{freeLocs}(c) \cup \text{freeLocs}(f) \\ \text{freeLocs}(\text{pure}_C e) &= \emptyset \end{aligned}$$

The free locations of the primitives (\rightsquigarrow) and `Locally` are just their arguments, while the free locations of an `Cond_` include the broadcaster and the free locations of the inner choreography. For other categories of expressions, *freeLocs* is applied homomorphically. The static analysis described in Section 4.2 essentially implements *freeLocs* by way of *the trick*. Our formal result relies on this definition and not on the implementation details.

The projection of `Cond_` differs based on the role played by the current location l in the branch c :

$$\llbracket \text{Cond}_- l_1 e a.c \rrbracket_l = \begin{cases} \text{BCast } (\text{unwrap}_{l_1} e) \text{ freeLocs}(c) \gg_N \lambda_. \llbracket c[\text{unwrap}_{l_1} e / a] \rrbracket_l & (\text{if } l = l_1) \\ \text{Recv } l \gg_N \lambda x. \llbracket c[x / a] \rrbracket_l & (\text{if } l \in \text{freeLocs}(c)) \\ \text{pure}_N \text{Empty} & (\text{otherwise}) \end{cases}$$

At the sender (l_1), it computes *freeLocs*(c) and broadcasts the value to its members before proceeding to execute c . At other locations included in *freeLocs*(c), it performs a `Recv` and instantiates c with the result. At locations uninvolved in c , it becomes a no-op (`Empty` located value).

6 Deadlock-Freedom

The primary guarantee offered by choreographic programming is *deadlock-freedom*: the execution of well-typed choreographies should not result in deadlocks at any participating location. It can be stated formally as follows:

Theorem 1 (DEADLOCK-FREEDOM).

$$\begin{aligned} \forall C, S, m, \tau. (\emptyset \vdash C : \text{Choreo } m \tau) \wedge (\llbracket C \rrbracket \rightsquigarrow_S^* S) \\ \implies (S \text{ final}) \vee (\exists S'. S \rightsquigarrow_S^* S') \end{aligned}$$

Theorem 1 states that, given a well-typed choreography C , if its endpoint projection steps to some system S , then S cannot get "stuck": it is either final (i.e. the network program at each location is a value), or it can step to some other system S' .

Deadlock-freedom follows as a corollary from the fact that endpoint projection is both sound (Theorem 3) and complete (Theorem 2). We include the proof in Appendix E.

6.1 Completeness of Endpoint Projection

Completeness means that the endpoint projection process preserves the high-level behavior of choreographies. Stated

formally, if a well-typed choreography C reduces to some choreography C' , then its projection reduces (in one or more steps) to the projection of C' :

Theorem 2 (COMPLETENESS).

$$\begin{aligned} \forall C, C', m, \tau. (\emptyset \vdash C : \text{Choreo } m \tau) \wedge (C \rightsquigarrow_C^* C') \\ \implies (\llbracket C \rrbracket \rightsquigarrow_S^* \llbracket C' \rrbracket) \end{aligned}$$

The proof of Theorem 2 (see Appendix E) proceeds by induction on the reduction semantics (\rightsquigarrow_C), showing in each case how a reduction trace can be built starting from $\llbracket C \rrbracket$ to reach $\llbracket C' \rrbracket$. In summary, this proof relies on two key aspects: firstly, on the fact that system reduction steps leave no messages unhandled (in other words, all "in-flight" messages must be consumed for a system step to occur); and secondly, on the special-cased endpoint projection of pure_C , which automatically projects $\text{Wrap}_l v$ expressions to Empty at locations other than l .

6.2 Soundness of Endpoint Projection

Soundness states that endpoint-projected choreographies cannot "go astray": any network system that resulted from the projection of a choreography C will eventually sync up with the projection of the next choreographic step C' . Formally, if the projection of a choreography C steps to some system S , then there exists some choreography C' such that C steps to it, and S steps to its projection:

Theorem 3 (SOUNDNESS).

$$\begin{aligned} \forall C, S. (\emptyset \vdash C : \text{Choreo } m \tau) \wedge (\llbracket C \rrbracket \rightsquigarrow_S^* S) \\ \implies \exists C'. (C \rightsquigarrow_C^* C') \wedge (S \rightsquigarrow_S^* \llbracket C' \rrbracket) \end{aligned}$$

The proof, shown below, relies on the choreographic type system's **PROGRESS** property, and the **CONFLUENCE** of system reduction (which is described further below).

Proof. By **PROGRESS**, the choreography C is either final or steps to some other choreography C_1 .

Case 3.1 (C is final). The goal is reached by choosing $C' = C$.

Case 3.2 ($C \rightsquigarrow_C^* C_1$). There are two subcases, based on whether S steps to $\llbracket C_1 \rrbracket$.

Case 3.2.1 ($S \rightsquigarrow_S^* \llbracket C_1 \rrbracket$). The goal is reached by choosing $C' = C_1$.

Case 3.2.2 (S does not step to $\llbracket C_1 \rrbracket$). By **CONFLUENCE**, there exists some system S_1 such that $S \rightsquigarrow_S^* S_1$ and $\llbracket C_1 \rrbracket \rightsquigarrow_S^* S_1$.

Apply **SOUNDNESS** recursively on C_1 and S_1 , producing a choreography C_2 such that $C_1 \rightsquigarrow_C^* C_2$ and $S_1 \rightsquigarrow_S^* \llbracket C_2 \rrbracket$.

The goal is then reached by choosing $C' = C_2$, because $C \rightsquigarrow_C^* C_1 \rightsquigarrow_C^* C_2$ and $S \rightsquigarrow_S^* S_1 \rightsquigarrow_S^* \llbracket C_2 \rrbracket$.

The recursion is well-founded because every choreography will eventually reach a final state (matching Case 1), due to **PROGRESS** and the lack of unbounded recursion in the choreographic language.

□

6.3 Confluence of System Reduction

Confluence in the context of network system reduction means that the order in which independent locations (or subsets of locations) reduce doesn't affect the result of the reduction. Stated formally, if a system S can step to different systems S_1 and S_2 , then both S_1 and S_2 eventually step to some common system S' :

Theorem 4 (CONFLUENCE).

$$\begin{aligned} \forall S, S_1, S_2. (S \rightsquigarrow_S^* S_1) \wedge (S \rightsquigarrow_S^* S_2) \\ \implies \exists S'. (S_1 \rightsquigarrow_S^* S') \wedge (S_2 \rightsquigarrow_S^* S') \end{aligned}$$

Theorem 4 follows from the fact that system reduction enjoys the stronger *diamond property* [40, Chapter Confluence].

6.4 The Diamond Property

The diamond property is similar in formulation to **CONFLUENCE**, but it provides a stronger claim by relating individual reduction steps ($S \rightsquigarrow_S S_1$) rather than reduction traces ($S \rightsquigarrow_S^* S_1$):

Theorem 5 (DIAMOND-PROPERTY).

$$\begin{aligned} \forall S, S_1, S_2. (S \rightsquigarrow_S S_1) \wedge (S \rightsquigarrow_S S_2) \\ \implies \exists S'. (S_1 \rightsquigarrow_S S') \wedge (S_2 \rightsquigarrow_S S') \end{aligned}$$

Proof sketch. The proof proceeds by induction on the steps $S \rightsquigarrow_S S_1$ and $S \rightsquigarrow_S S_2$. Since the **S-DISJOINT** rule is only used to select which location(s) to step at, we will consider combinations of **S-SINGLE** and **S-SYNC** under any number of **S-DISJOINT** compositions.

Case 5.1 (**S-SINGLE** and **S-SINGLE**, at distinct locations). Because **S-SINGLE** requires the network step to be \emptyset -annotated, steps at different locations commute. Thus, S_1 can step to S' by applying the step that leads to S_2 and vice versa.

Case 5.2 (**S-SINGLE** and **S-SINGLE**, at the same location). The steps commute by the **DIAMOND-PROPERTY** of the network language, which holds on the assumption that the local language (a simply-typed lambda calculus) is confluent.

Case 5.3 (**S-SYNC** and **S-SYNC**, at distinct locations l_1, l_2, l_3, l_4). Because all locations are distinct, the communication between l_1 and l_2 and that between l_3 and l_4 are independent, so the steps commute.

Case 5.4 (**S-SYNC** and **S-SYNC**, with the same sender, or receiver, or both). By the definition of (\rightsquigarrow_S), this case is impossible.

Case 5.5 (**S-SINGLE** and **S-SYNC**, at different locations l_1, l_2, l_3). The local reduction at l_1 and the communication between l_2 and l_3 are independent, so the steps commute.

Case 5.6 (**S-SINGLE** and **S-SYNC**, with either the same sender or receiver). By the definition of (\rightsquigarrow_S), this case is impossible.

□

7 Related Work

Choreographic programming. The notion of choreographies originated as a way to write web services (e.g. [31, 39]). The concept of choreographic programming was introduced by Carbone and Montesi [8, 27]. The first functional choreographic languages are Pirouette [18] and Chor λ [10]. In recent work [17], the authors introduce an extension of Chor λ , adding support for polymorphism, called PolyChor λ . Different from these works, which focus on semantic foundations, HasChor [34], the closest related work, focuses on providing a practically usable and deployable implementation. We adhere to the inspiration line of HasChor and provide the concept of multiply-located values, a more efficient branching operator (*cond*), and the first formal semantics for HasChor. One of the main difference with our formal approach is the formulation of branching. Pirouette projects each branch of an *if-then-else* separately and then combines them using a *partial merge operator*. This differs from our formalization, where the *cond* operator both performs a local choice and broadcasts it to the involved locations—which we believe leads to a simpler semantics. Unlike Pirouette and HasChor, Chor λ does not make use of a separate local language. Instead, both local and global computations are expressed in the same language. Endpoint projection translates Chor λ into a process language (inspired by the π -calculus) with explicit *send* and *recv* operations—an inspiration we also took in our formalization. Our approach enhances PolyChor λ by demonstrating the practical application of location polymorphism in intricate protocols, such as data clean rooms, through the lens of generic programming—thus enabling location polymorphic protocols to be type-checked once and instantiated with different participants.

Multiply-located values. In concurrent work, Bates et al. [5] introduce a new framework, MultiChor, for choreographic programming in Haskell, Rust, and TypeScript, based on multiply-located values and *conclaves*. While they identify some of the same issues in HasChor and other existing choreographic frameworks, they reach solutions different to ours. The primary difference is that they are proposing a completely new framework, while our approach aims to build upon and stay compatible with HasChor. To this end, the notion of multiply-located values (MLVs) is a built-in aspect of MultiChor (both in its formalization and its implementations), while our implementation relies only on HasChor’s public API and generic programming techniques. Secondly, conclaves, which address the problem of overly-broad knowledge of choice (KoC), require tracking the set of a choreography’s participants (“quorum”) at the type-level, making the API more complex. Our solution based on static analysis instead preserves HasChor’s simpler typing. Lastly, in proving the soundness of the formalized semantics, our approach relies on a standard confluence argument, while

MultiChor employs a specially-designed proof by contradiction. A summary of the differences between MultiChor and CloudChor is provided in Table 2 below.

Table 2. Comparison between MultiChor and CloudChor

Feature	MultiChor	CloudChor
MLVs	✓	✓
Granular KoC	Conclaves	Static analysis
API complexity	More complex types (Quorum tracking)	Compatible with HasChor
Proof method	Ad-hoc argument	By confluence

Data clean rooms. Data Clean Rooms (DCRs) have developed as a novel privacy-enhancing technology designed to facilitate the secure exchange and analysis of data in scenarios involving marketing [26] or health data [28]. However, as highlighted by the Federal Trade Commission (FTC) [15], “By default, most services that provide DCRs are not privacy preserving.” This underscores the necessity for meticulous design and implementation to ensure genuine privacy preservation within DCRs—an objective this work moves towards. The work of HasTEE [33] provides an example of DCR implementation with IFC using Intel SGX hardware [20]. A significant contribution of this work is porting GHC¹’s runtime to run inside an SGX enclave. The programming model in question exclusively accommodates two security tiers: public and secret. This setup is designed to function with a solitary server within the SGX enclave, along with an associated client external to it. Consequently, implementing comprehensive DCR protocols that can accommodate multiple clients—as in this paper—or servers becomes challenging. As future work, it would be interesting to adapt the endpoint projection process to be aware of SGX enclaves or similarly-purposed security hardware [3, 4, 9, 19, 25].

8 Final Remarks

By leveraging generic programming techniques, we introduced *multiply-located values*, which represent N-ary products spanning multiple locations. We developed combinators to construct and manipulate these values, allowing for more expressive choreographies and protocols designed for arbitrary numbers of participants. We also introduced an optimized *cond_* operator which selectively propagates branching choices only to relevant locations, based on a partial evaluation technique known as *the trick*. This work also provides a formalization of HasChor’s choreographic programming model, along with the newly-introduced *cond_* operator. We prove *deadlock-freedom* by showing that *endpoint projection* preserves the semantics of choreographies.

¹<https://www.haskell.org/ghc/>

Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

- [1] Maximilian Algehed and Alejandro Russo. 2017. Encoding DCC in Haskell. In *Proc. of the Workshop on Programming Languages and Analysis for Security (PLAS '17)*. Association for Computing Machinery. doi:10.1145/3139337.3139338
- [2] Amazon Web Services. 2023. AWS Clean Rooms Now Generally Available. <https://aws.amazon.com/blogs/aws/aws-clean-rooms-now-generally-available/> Accessed: 2025-02-25.
- [3] AMD. 2017. AMD Secure Encrypted Virtualization (SEV). <https://www.amd.com/en/developer/sev.html> Accessed: 2025-02-17.
- [4] ARM. 2021. ARM CCA will put confidential compute in the hands of every developer. <https://newsroom.arm.com/news/arm-cca-will-put-confidential-compute-in-the-hands-of-every-developer> Accessed: 2025-02-17.
- [5] 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
- [6] Brazilian Government. 2018. Lei Geral de Proteção de Dados (LGPD). Available at: <https://www.gov.br/lgpd>.
- [7] California State Legislature. 2018. California Consumer Privacy Act (CCPA). Available at: <https://oag.ca.gov/privacy/ccpa>.
- [8] Marco Carbone and Fabrizio Montesi. 2013. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *Proc. of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '13)*. Association for Computing Machinery. doi:10.1145/2480359.2429101
- [9] Intel Corporation. 2023. Intel Trust Domain Extensions (TDX) Overview. <https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/overview.html> Accessed: 2025-02-17.
- [10] Luís Cruz-Filipe, Eva Graversen, Lovro Lugović, Fabrizio Montesi, and Marco Peressotti. 2022. Functional Choreographic Programming. In *Theoretical Aspects of Computing – ICTAC 2022: 19th International Colloquium, Tbilisi, Georgia, September 27–29, 2022, Proceedings (ICTAC 2022)*. doi:10.1007/978-3-031-17715-6_15
- [11] Edsko de Vries and Andres Löb. 2014. True sums of products. In *Proc. of the 10th ACM SIGPLAN Workshop on Generic Programming (WGP '14)*. ACM. doi:10.1145/2633628.2633634
- [12] European Union. 2016. General Data Protection Regulation (GDPR). <https://eur-lex.europa.eu/eli/reg/2016/679/oj/eng> Regulation (EU) 2016/679.
- [13] European Union. 2022. Data Governance Act. <https://eur-lex.europa.eu/eli/reg/2022/868/oj/eng> Regulation (EU) 2022/868.
- [14] European Union. 2022. Proposal for a Regulation on harmonised rules on fair access to and use of data (Data Act). <https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A52022PC0068> Proposal COM/2022/68 final.
- [15] Federal Trade Commission. 2024. Data Clean Rooms: Separating Fact from Fiction. <https://www.ftc.gov/policy/advocacy-research/tech-at-ftc/2024/11/data-clean-rooms-separating-fact-fiction> Accessed: 2025-02-17.
- [16] Google Cloud. 2024. Google Cloud Introduces Clean Rooms for Secure Data Collaboration. <https://cloud.google.com/blog/topics/analytics/google-cloud-announces-clean-rooms> Accessed: 2025-02-25.
- [17] Eva Graversen, Andrew K. Hirsch, and Fabrizio Montesi. 2024. Alice or Bob?: Process polymorphism in choreographies. *Journal of Functional Programming* 34 (2024). doi:10.1017/S0956796823000114
- [18] Andrew K. Hirsch and Deepak Garg. 2022. Pirouette: higher-order typed functional choreographies. *Proc. ACM Program. Lang.* 6, POPL (Jan. 2022). doi:10.1145/3498684
- [19] Guernsey D. H. Hunt, Ramachandra Pai, Michael V. Le, Hani Jamjoom, Sukadev Bhattiprolu, Rick Boivie, Laurent Dufour, Brad Frey, Mohit Kapur, Kenneth A. Goldman, Ryan Grimm, Janani Janakiraman, John M. Ludden, Paul Mackerras, Cathy May, Elaine R. Palmer, Bharata Bhasker Rao, Lawrence Roy, William A. Starke, Jeff Stuecheli, Enriquillo Valdez, and Wendel Voigt. 2021. Confidential computing for OpenPOWER. In *Proc. of the European Conference on Computer Systems (EuroSys '21)*. Association for Computing Machinery. doi:10.1145/3447786.3456243
- [20] Intel. 2015. Intel Software Guard Extensions. <https://www.intel.com/content/www/us/en/developer/tools/software-guard-extensions/overview.html> Accessed: 2025-02-17.
- [21] Alex Ionescu and Alejandro Russo. 2025. CloudChor GitHub Repository. <https://github.com/aionescu/cloudchor>
- [22] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. 1993. Partial evaluation and automatic program generation. In *Prentice Hall international series in computer science*. Prentice-Hall, Inc. doi:10.5555/153676
- [23] Oleg Kiselyov and Hiromi Ishii. 2015. Freer monads, more extensible effects. In *Proc. ACM SIGPLAN Symposium on Haskell (Haskell '15)*. Association for Computing Machinery. doi:10.1145/2804302.2804319
- [24] Oleg Kiselyov, Ralf Lämmel, and Kean Schupke. 2004. Strongly typed heterogeneous collections. In *Proc. of the ACM SIGPLAN Workshop on Haskell (Haskell '04)*. Association for Computing Machinery. doi:10.1145/1017472.1017488
- [25] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: an open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems (EuroSys '20)*. Association for Computing Machinery, Article 38, 16 pages. doi:10.1145/3342195.3387532
- [26] Kungang Li, Xiangyi Chen, Ling Leng, Jiajing Xu, Jiankai Sun, and Behnam Rezaei. 2024. Privacy Preserving Conversion Modeling in Data Clean Room. In *Proc. of the ACM Conference on Recommender Systems (RecSys '24)*. Association for Computing Machinery. doi:10.1145/3640457.3688054
- [27] Fabrizio Montesi. 2013. *Choreographic Programming*. Ph.D. Thesis. IT University of Copenhagen. <https://www.fabriziomontesi.com/files/choreographic-programming.pdf>.
- [28] Paul Muntner, Ryan K. Hernandez, Stephen T. Kent, Jennifer E. Brown-ing, David T. Gilbertson, Kimberly E. Hurwitz, Susan S. Jick, Edmond C. Lai, Timothy L. Lash, Keri L. Monda, Kenneth J. Rothman, Brian D. Bradbury, and M. Alan Brookhart. 2024. Staging and clean room: Constructs designed to facilitate transparency and reduce bias in comparative analyses of real-world data. *Pharmacoepidemiology and Drug Safety* 33, 3 (March 2024), e5770. doi:10.1002/pds.5770
- [29] National People's Congress of China. 2021. Personal Information Protection Law (PIPL). Available at: <https://npcobserver.com/pipl/>.
- [30] Markus Pettersson, Johannes Ljung Ekeröth, and Alejandro Russo. 2024. Calculating Function Sensitivity for Synthetic Data Algorithms. In *Proc. of the Symposium on Implementation and Application of Functional Languages (IFL '23)*. Association for Computing Machinery, Article 6. doi:10.1145/3652561.3652567
- [31] Zongyan Qiu, Xiangpeng Zhao, Chao Cai, and Hongli Yang. 2007. Towards the theoretical foundation of choreography. In *Proc. of the 16th International Conference on World Wide Web (WWW '07)*. Association for Computing Machinery. doi:10.1145/1242572.1242704
- [32] Alejandro Russo. 2015. Functional pearl: two can keep a secret, if one of them uses Haskell. In *Proc. of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2015)*. doi:10.1145/2858949.2784756
- [33] Abhiroop Sarkar, Robert Krook, Alejandro Russo, and Koen Claessen. 2023. HasTEE: Programming Trusted Execution Environments with

- Haskell. In *Proc. of the 16th ACM SIGPLAN International Haskell Symposium (Haskell 2023)*. ACM, 72–88. doi:10.1145/3609026.3609731
- [34] Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). *Proc. ACM Program. Lang.* 7, ICFP (Aug. 2023). doi:10.1145/3607849
- [35] Snowflake. 2024. Snowflake Clean Rooms Available for Secure Data Collaboration. <https://www.snowflake.com/blog/snowflake-clean-rooms-general-availability/> Accessed: 2025-02-25.
- [36] Deian Stefan, David Mazières, John C. Mitchell, and Alejandro Russo. 2017. Flexible dynamic information flow control in the presence of exceptions. *Journal of Functional Programming* 27 (2017). doi:10.1017/S0956796816000241
- [37] David Terei, Simon Marlow, Simon Peyton Jones, and David Mazières. 2012. Safe Haskell. In *Proc. of the Haskell Symposium (Haskell '12)*. Association for Computing Machinery. doi:10.1145/2364506.2364524
- [38] Nachiappan Valliappan, Robert Krook, Alejandro Russo, and Koen Claessen. 2020. Towards secure IoT programming in Haskell. In *Proc. of the ACM SIGPLAN International Symposium on Haskell (Haskell 2020)*. Association for Computing Machinery. doi:10.1145/3406088.3409027
- [39] W3C. 2004. CWS Choreography Model Overview. <https://www.w3.org/TR/ws-chor-model/>
- [40] Philip Wadler, Wen Kokke, and Jeremy G. Siek. 2022. *Programming Language Foundations in Agda*. <https://plfa.inf.ed.ac.uk/22.08/>

A Soundness Counter-Examples

Frontrunning. The first counter-example from Figure 7 breaks the deadlock-freedom guarantee of HasChor by "frontrunning" network communication. It works by exploiting knowledge about the selected HasChor backend, in this case the HTTP backend. The setup consists of a choreography with three locations (11, 12, and 13), where the resulting projections are executed on the same machine, but on different TCP ports. Each location is assigned its own port (starting at 3000 for 11, 3001 for 12, and so on). The choreography proceeds by running a local computation at 11 (line 3)—which by definition should not affect other locations—followed by a cond that broadcasts the boolean value True from 11 (line 7), and branches on it. If the broadcasted value is True, 12 sends the integer 2 to 13 (line 10); otherwise, the choreography ends (line 8).

However, the local computation at 11 *can* actually affect other locations, and causes location 13 to deadlock. It achieves this by issuing a direct HTTP send operation to 12's port (3001) with the value False (line 4), *before* 11 broadcasts the True value. HasChor's HTTP backend stores received values in FIFO channels, so when 12 executes the cond and performs a Network-level recv, it will read the False value from the channel and branch on it, assuming it was the broadcasted value. This causes it to take the False branch and immediately finish execution without initiating any communication with 13. Meanwhile, location 13 receives the broadcasted True value and branches on it, initiating a Network-level recv from 12 (line 10). This causes 13 to deadlock, thus breaking the soundness guarantees of choreographic programming.

Collusion. The second counter-example presented in Figure 7 allows locations to *collude*, sharing information outside the choreography. As with the previous counter-example, the setup requires a choreography with three locations 11, 12, and 13. The choreography begins by running a local computation at 11 (line 3), followed by a communication of data from 13 to 12 (line 7), and ends with a local computation at 12 (line 9). Assuming the guarantees provided by HasChor, 11 should not have access to the value passed from 13 to 12. This counter-example shows that not to be the case, by exfiltrating that value from 12 to 11. This is achieved through a secondary HTTP connection that is established inside the local computations at 11 (line 4) and 12 (line 10) on a port that is unused by the HTTP backend (in this case 2999). While in this case data is exfiltrated to another participant within the choreography, the same technique could be used to communicate with an external third party. In the case of a protocol such as the data clean room, this could lead to leakage of sensitive or private information.

These counter-examples show that unrestricted I/O in local computations can break the deadlock-freedom guarantees of choreographic programming, and highlight potential security issues related to information flow.

```

1 frontrun :: Choreo IO ()
2 frontrun = do
3   locally 11 $ \_ -> do
4     httpSend "11" 3001 False
5     exitSuccess
6
7   cond (11, wrap True) $ \case
8     False -> pure ()
9     True -> do
10      (12, wrap 2) ~> 13
11      pure ()

```

```

1 collusion :: Choreo IO ()
2 collusion = do
3   locally 11 $ \unwrap -> do
4     v <- httpRecv 2999
5     print v
6
7   v12 <- (13, wrap 2) ~> 12
8
9   locally 12 $ \unwrap -> do
10    httpSend "12" 2999 $ unwrap v12
11
12 pure ()

```

Figure 7. Examples of unsound choreographies

B Additional Combinators

pointwise. The implementation of $(\sim>.)$ iterates over the list of locations and the list of values in lockstep. At each step, the list of values is unpacked into its head and tail; the head is sent to its corresponding location, and the tail is passed to the next iteration:

```
(l, as) ~>. Nil = pure Nil
(l, as) ~>. (l' :* ls') = do
  a <- locally l $ \unwrap -> pure $ unK $ hd $ unwrap as
  as' <- locally l $ \unwrap -> pure $ tl $ unwrap as
  (:*) <$> ((l, a) ~> l') <*> ((l, as') ~>. ls')
```

Because the list of values to send is itself located at l , we cannot directly pattern-match on it, instead we have to project out its head (hd) and tail (tl) inside separate calls to `locally`.

multilocally. Each local computation receives 2 arguments: a `Proxy` specifying the location it is running on, and an `Unwraps` function which can unwrap a multiply-located value (parameterized in the same ls), picking the corresponding value out of the list.

```
multilocally :: All KnownSymbol ls
=> NP Proxy ls
-> (forall l. KnownSymbol l =>
    Proxy l -> Unwraps ls -> m a)
-> Choreo m (a @@ ls)
```

The implementation of `multilocally` is more involved than the other combinators, as it needs to construct appropriate `Unwraps` functions for each location involved:

```
multilocally ls f = go ls id
where
  go :: forall ls'. All KnownSymbol ls'
    => NP Proxy ls'
    -> (forall a. a @@ ls -> a @@ ls')
    -> Choreo m (a @@ ls')
  go Nil pick = pure Nil
  go (l :* ls) pick = do
    a <- locally l $ \unwrap -> f l $ unwrap . hd . pick
    (a :*) <$> go ls (tl . pick)
```

The top-level function dispatches to the helper `go`, which processes each location l in turn, performing the local computation f at l , and then recursing on the remaining list of locations. To construct the appropriate `Unwraps` functions, `go` uses an accumulator parameter called `pick`, of type $(\forall a. a @@ ls \rightarrow a @@ ls')$.

The function `pick`, when given a list containing values for all locations ls , drops just enough elements from the front of the list, such that the head of its output is the value corresponding to the location `go` is currently processing. When passed to the local computation, `pick` is composed with `hd` (NP's version of `head`) to create a suitable `Unwraps` function for the current location, picking out its value from the multiply-located list. When recursing deeper into the list of locations, `pick` is composed with `tl` (NP's version of `tail`), such that at every iteration it drops one more element

of the list. In the first call to `go`, `pick` is initialized to the identity function `id`, since the target value of the first iteration is already at the head of the list. `go` uses an additional type parameter, ls' to represent the list of locations not yet processed. In the first call to `go`, ls' is instantiated to ls , and every recursive call drops one more element from it.

Sending the results of local computations. `HasChor` provides a combinator $(\sim\sim\sim\sim\sim)$ for the common case where a location wants to perform a local computation and immediately send its result. Its type looks similar to that of $(\sim\sim\sim)$, but instead of a value $a @ l$ it accepts a local computation:

```
(\sim\sim\sim\sim\sim) :: (Show a, Read a, KnownSymbol l, KnownSymbol l')
=> (Proxy l, Unwrap l -> m a) -> Proxy l'
-> Choreo m (a @ l')
```

We provide similar variants for the multiply-located combinators: $(\sim\sim\sim\sim\sim*)$ to scatter the result of a local computation, and $(*\sim\sim\sim\sim\sim)$ to gather the results of a multi-local computation. Their implementations simply compute a value using `locally` (respectively `multilocally`) before passing it to the appropriate communication combinator:

```
(\sim\sim\sim\sim\sim*) :: (KnownSymbol l, All KnownSymbol ls, Show a, Read a)
=> (Proxy l, Unwrap l -> m a) -> NP Proxy ls
-> Choreo m (a @@ ls)

(l, f) \sim\sim\sim\sim\sim* ls = do
  a <- locally l f
  (l, a) \sim\sim\sim\sim\sim* ls

(*\sim\sim\sim\sim\sim) :: (KnownSymbol l, All KnownSymbol ls, Applicative m,
  Show a, Read a)
=> (NP Proxy ls, forall l. KnownSymbol l =>
  Proxy l -> Unwraps ls -> m a)
-> Proxy l
-> Choreo m (NP (K a) ls @ l)

(ls, f) *\sim\sim\sim\sim\sim l = do
  a <- multilocally ls f
  a *\sim\sim\sim\sim\sim l
```

C Summary of the CloudChor API

Figure 8 shows the full API of our proposed extensions to HasChor, including multiply-located combinators and the optimized `cond_`. Their full source code is included in the supplementary material [21].

```

type a @@ ls
type Unwraps ls

(~>*) :: (KnownSymbol l, All KnownSymbol ls, Show a, Read a)
      => (Proxy l, a @ l) -> NP Proxy ls
      -> Choreo m (a @@ ls)

(*~>) :: (KnownSymbol l, All KnownSymbol ls, Applicative m
        , Show a, Read a)
      => a @@ ls
      -> Proxy l
      -> Choreo m (NP (K a) ls @ l)

(~>.) :: (KnownSymbol l, All KnownSymbol ls, Applicative m
        , Show a, Read a)
      => (Proxy l, NP (K a) ls @ l)
      -> NP Proxy ls
      -> Choreo m (a @@ ls)

multilocally :: All KnownSymbol ls => NP Proxy ls
              -> (forall l. KnownSymbol l =>
                  Proxy l -> Unwraps ls -> m a)
              -> Choreo m (a @@ ls)

cond_ :: (Bounded a, Enum a, Show a, Read a, KnownSymbol l)
      => (Proxy l, a @ l)
      -> (a -> Choreo m (b @ l))
      -> Choreo m (b @ l)

(~~>*) :: (KnownSymbol l, All KnownSymbol ls, Show a, Read a)
      => (Proxy l, Unwrap l -> m a)
      -> NP Proxy ls
      -> Choreo m (a @@ ls)

(*~~>) :: (KnownSymbol l, All KnownSymbol ls, Applicative m
        , Show a, Read a)
      => (NP Proxy ls,
          forall l. KnownSymbol l
            => Proxy l -> Unwraps ls -> m a)
      -> Proxy l
      -> Choreo m (NP (K a) ls @ l)

cond_' :: (Bounded a, Enum a, Show a, Read a, KnownSymbol l)
      => (Proxy l, Unwrap l -> m a)
      -> (a -> Choreo m (b @ l))
      -> Choreo m (b @ l)

```

Figure 8. The CloudChor API

D Additional Typing Rules and Definitions

Figure 9 shows the full grammar for types, while Figures 10 and 11 list the typing rules for choreographies and network programs, respectively. Lastly, Figure 12 includes the full definition of endpoint-projection (EPP).

Type $\tau ::=$	()	Unit
	Int	Integers
	$\tau_1 \rightarrow \tau_2$	Functions
	$\tau @ l$	Located values
	$m \tau$	Local effects
	Choreo $m \tau$	Choreographies
	Network $m \tau$	Network programs

Figure 9. Grammar for types

T-COMM	$\frac{\Gamma \vdash e : \text{Int} @ l_1}{\Gamma \vdash (l_1, e) \rightsquigarrow l_2 : \text{Choreo } m (\text{Int} @ l_2)}$
T-COND	$\frac{\Gamma \vdash e : \text{Int} @ l \quad \Gamma, x : \text{Int} \vdash c : \text{Choreo } m \tau}{\Gamma \vdash \text{Cond } l \text{ e } x.c : \text{Choreo } m \tau}$
T-COND_	$\frac{\Gamma \vdash e : \text{Int} @ l \quad \Gamma, x : \text{Int} \vdash c : \text{Choreo } m (\tau @ l)}{\Gamma \vdash \text{Cond}_l \text{ e } x.c : \text{Choreo } m (\tau @ l)}$
T-LOCALLY	$\frac{\Gamma, u : (\tau @ l \rightarrow \tau) \vdash e : m \tau}{\Gamma \vdash \text{Locally } l \text{ u.e} : \text{Choreo } m (\tau @ l)}$
T-BIND-CHOREO	$\frac{\Gamma \vdash c_1 : \text{Choreo } m \tau_1 \quad \Gamma, x : \tau_1 \vdash c_2 : \text{Choreo } m \tau_2}{\Gamma \vdash c_1 \gg_C (\lambda x. c_2) : \text{Choreo } m \tau_2}$
T-PURE-CHOREO	$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{pure}_C e : \text{Choreo } m \tau}$

Figure 10. Typing rules for choreographies

T-SEND	$\frac{\Gamma \vdash e : \text{Int}}{\Gamma \vdash \text{Send } e \text{ l} : \text{Network } m ()}$
T-RECV	$\Gamma \vdash \text{Recv } l : \text{Network } m \text{ Int}$
T-BCAST	$\Gamma \vdash \text{BCast } e \text{ ls} : \text{Network } m ()$
T-RUN	$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{Run } e : \text{Network } m \tau}$
T-BIND-NETWORK	$\frac{\Gamma \vdash p_1 : \text{Network } m \tau_1 \quad \Gamma, x : \tau \vdash p_2 : \text{Network } m \tau_2}{\Gamma \vdash p_1 \gg_N (\lambda x. p_2) : \text{Network } m \tau_2}$
T-PURE-NETWORK	$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{pure}_C e : \text{Network } m \tau}$

Figure 11. Typing rules for network programs

$$\llbracket (l_1, e) \rightsquigarrow l_2 \rrbracket_l = \begin{cases} \text{pure}_N e & \text{if } l = l_1 = l_2 \\ \text{Send } (\text{unwrap}_{l_1} e) l_2 \gg_N \lambda_. \text{Empty} & \text{if } l = l_1 \neq l_2 \\ \text{Recv } l \gg_N \lambda x. \text{pure}_N (\text{Wrap}_{l_2} x) & \text{if } l = l_2 \neq l_1 \\ \text{pure}_N \text{Empty} & \text{otherwise} \end{cases} \quad (1)$$

$$\llbracket \text{Cond } l_1 e a.c \rrbracket_l = \begin{cases} \text{BCast } (\text{unwrap}_{l_1} e) \text{locs}(c) \gg_N \lambda_. \llbracket c[\text{unwrap}_{l_1} e / a] \rrbracket_l & \text{if } l = l_1 \\ \text{Recv } l \gg_N \lambda x. \llbracket c[x / a] \rrbracket_l & \text{otherwise} \end{cases} \quad (2)$$

$$\llbracket \text{Cond_} l_1 e a.c \rrbracket_l = \begin{cases} \text{BCast } (\text{unwrap}_{l_1} e) \text{freeLocs}(c) \gg_N \lambda_. \llbracket c[\text{unwrap}_{l_1} e / a] \rrbracket_l & \text{if } l = l_1 \\ \text{Recv } l \gg_N \lambda x. \llbracket c[x / a] \rrbracket_l & \text{if } l \in \text{freeLocs}(c) \\ \text{pure}_N \text{Empty} & \text{otherwise} \end{cases} \quad (3)$$

$$\llbracket \text{Locally } l_1 u.e \rrbracket_l = \begin{cases} \text{Run } e[\text{unwrap}_{l_1} / u] \gg_N \lambda x. \text{pure}_N (\text{Wrap}_{l_1} x) & \text{if } l = l_1 \\ \text{pure}_N \text{Empty} & \text{otherwise} \end{cases} \quad (4)$$

$$\llbracket c \gg_C f \rrbracket_l = \llbracket c \rrbracket_l \gg_N \llbracket f \rrbracket_l \quad (5)$$

$$\llbracket \text{pure}_C e \rrbracket_l = \begin{cases} \text{pure}_N \text{Empty} & \text{if } (e \longrightarrow^* \text{Wrap}_{l_1} e') \wedge l \neq l_1 \\ \text{pure}_N e & \text{otherwise} \end{cases} \quad (6)$$

$$\llbracket \Sigma; c \rrbracket = (l_1 \triangleright \Sigma[l_1]; \llbracket c \rrbracket_{l_1}) \parallel \dots \parallel (l_n \triangleright \Sigma[l_n]; \llbracket c \rrbracket_{l_n}) \quad (7)$$

Figure 12. Endpoint projection of choreographies

E Additional Proofs

Theorem 1 (DEADLOCK-FREEDOM).

$$\begin{aligned} \forall C, S, m, \tau. (\emptyset \vdash C : \text{Choreo } m \tau) \wedge (\llbracket C \rrbracket \rightsquigarrow_S^* S) \\ \implies (S \text{ final}) \vee (\exists S'. S \rightsquigarrow_S^* S') \end{aligned}$$

Proof.

Case 5.7 (S is final). Goal is reached.

Case 5.8 (S is not final). Apply SOUNDNESS with C and S , producing a choreography C' such that $S \rightsquigarrow_S^* \llbracket C' \rrbracket$.

The goal is reached by choosing $S' = \llbracket C' \rrbracket$.

□

Theorem 2 (COMPLETENESS).

$$\begin{aligned} \forall C, C', m, \tau. (\emptyset \vdash C : \text{Choreo } m \tau) \wedge (C \rightsquigarrow_C C') \\ \implies (\llbracket C \rrbracket \rightsquigarrow_S^* \llbracket C' \rrbracket) \end{aligned}$$

Proof. By structural induction on the reduction semantics (\rightsquigarrow_C).

Case 5.9 (C-local). Definitions:

$$\begin{aligned} \forall l. \llbracket C \rrbracket_l &= e \\ \forall l. \llbracket C' \rrbracket_l &= e' \end{aligned}$$

Reduction:

Apply S-DISJOINT (S-SINGLE (N-LOCAL)) once for every location.

Case 5.10 (C-pure-bind). Definitions:

$$\begin{aligned} C &= \text{pure}_C e \gg_C f \\ C' &= f e \end{aligned}$$

$$\begin{aligned} \forall l. \llbracket C \rrbracket_l &= \text{pure}_N e \gg_N \llbracket f \rrbracket_l \\ \forall l. \llbracket C' \rrbracket_l &= \llbracket f \rrbracket_l e \end{aligned}$$

Reduction:

Apply S-DISJOINT (S-SINGLE (N-PURE-BIND)) once for every location.

Case 5.11 (C-bind). Definitions:

$$\begin{aligned} C &= c_1 \gg_C f \\ C' &= c'_1 \gg_C f \end{aligned}$$

$$\begin{aligned} \forall l. \llbracket C \rrbracket_l &= \llbracket c_1 \rrbracket_l \gg_N \llbracket f \rrbracket_l \\ \forall l. \llbracket C' \rrbracket_l &= \llbracket c'_1 \rrbracket_l \gg_N \llbracket f \rrbracket_l \end{aligned}$$

Reduction:

Applying the induction hypothesis with c_1 and c'_1 produces a system reduction trace from $\llbracket c_1 \rrbracket$ to $\llbracket c'_1 \rrbracket$. The steps from this reduction trace can be re-applied under N-BIND to each location in $\llbracket C \rrbracket$.

Case 5.12 (C-locally). Definitions:

$$\begin{aligned} C &= \text{Locally } l_1 u.e \\ C' &= \text{pure}_C (\text{Wrap}_{l_1} e') \\ \Sigma' &= \Sigma[l_1 \triangleright \sigma'_1] \\ \forall l. \llbracket C \rrbracket_l &= \text{Run } e[\text{unwrap}_{l_1} / u] \gg_N \lambda x. \text{pure}_N (\text{Wrap}_{l_1} x) \\ &= p_1 \\ \forall l \neq l_1. \llbracket C \rrbracket_l &= \text{pure}_N \text{Empty} \\ \forall l. \llbracket C' \rrbracket_l &= \text{pure}_N (\text{Wrap}_{l_1} e') \\ \forall l \neq l_1. \llbracket C' \rrbracket_l &= \text{pure}_N \text{Empty} \end{aligned}$$

Reduction:

Apply S-DISJOINT (S-SINGLE (N-LOCAL)) at location l_1 . Other locations $l_n \neq l_1$ do not need to reduce.

Case 5.13 (C-comm). Definitions:

$$\begin{aligned} C &= (l_1, \text{Wrap}_{l_1} v) \rightsquigarrow l_2 \\ C' &= \text{pure}_C (\text{Wrap}_{l_2} v) \end{aligned}$$

$$\begin{aligned} \llbracket C \rrbracket_{l_1} &= \text{Send } (\text{unwrap}_{l_1} (\text{Wrap}_{l_1} v)) l_2 \gg_N \lambda_. \text{Empty} \\ &= p_1 \\ \llbracket C \rrbracket_{l_2} &= \text{Recv } l_1 \gg_N \lambda x. \text{pure}_N (\text{Wrap}_{l_2} x) = p_2 \\ \forall l \neq l_1 \neq l_2. \llbracket C \rrbracket_l &= \text{pure}_N \text{Empty} \\ \llbracket C' \rrbracket_{l_1} &= \text{pure}_N \text{Empty} = p'_1 \\ \llbracket C' \rrbracket_{l_2} &= \text{pure}_N (\text{Wrap}_{l_2} v) = p'_2 \\ \forall l \notin \{l_1, l_2\}. \llbracket C' \rrbracket_l &= \text{pure}_N \text{Empty} \\ p'_1 &= \text{pure}_N () \gg_N \lambda_. \text{Empty} \\ p'_2 &= \text{pure}_N v \gg_N \lambda x. \text{pure}_N (\text{Wrap}_{l_2} x) \end{aligned}$$

Reduction:

Apply S-DISJOINT (S-SINGLE (N-LOCAL)) at l_1 (to reduce $(\text{unwrap}_{l_1} (\text{Wrap}_{l_1} v)) \longrightarrow v$).

Apply S-DISJOINT (S-SYNC (N-BIND (N-SEND)) (N-BIND (N-RECV))) at l_1 and l_2 .

Apply S-DISJOINT (S-SINGLE (N-PURE-BIND)) at l_1 .

Apply S-DISJOINT (S-SINGLE (N-PURE-BIND)) at l_2 .

Case 5.14 (C-cond). Definitions:

$$C = \text{Cond } l_1 (\text{Wrap}_{l_1} v) a.c_1$$

$$C' = c_1[v / a]$$

$$\llbracket C \rrbracket_{l_1} = \text{BCast } (\text{unwrap}_{l_1} (\text{Wrap}_{l_1} v)) \text{ locs}(C)$$

$$\ggg_N \lambda_. \llbracket c_1[\text{unwrap}_{l_1} (\text{Wrap}_{l_1} v) / a] \rrbracket_l$$

$$\forall l \neq l_1. \llbracket C \rrbracket_l = \text{Recv } l \ggg_N \lambda x. \llbracket c_1[x / a] \rrbracket_l$$

$$\llbracket C' \rrbracket_{l_1} = \llbracket c_1[\text{unwrap}_{l_1} (\text{Wrap}_{l_1} v) / a] \rrbracket_{l_1}$$

$$\forall l \neq l_1. \llbracket C' \rrbracket_l = \llbracket c_1[\text{unwrap}_{l_1} (\text{Wrap}_{l_1} v) / a] \rrbracket_l$$

Reduction:

Apply S-DISJOINT (S-SINGLE (N-LOCAL)) at l_1 (to reduce $(\text{unwrap}_{l_1} (\text{Wrap}_{l_1} v)) \longrightarrow v$).

Then, repeat for each location l_i :

Apply S-DISJOINT (S-SYNC (N-BIND N-BCAST-CONS) (N-BIND N-RECV)) at l_1 and l_i (to synchronize a send and a recv).

Apply S-DISJOINT (S-SINGLE (N-PURE-BIND)) at l_i .

Finally:

Apply S-DISJOINT (S-SINGLE (N-BCAST-NIL)) at l_1 .

Apply S-DISJOINT (S-SINGLE (N-PURE-BIND)) at l_1 .

Case 5.15 (C-cond₋). Definitions:

$$C = \text{Cond}_- l_1 (\text{Wrap}_{l_1} v) a.c_1$$

$$C' = c_1[v / a]$$

$$\llbracket C \rrbracket_{l_1} = \text{BCast } (\text{unwrap}_{l_1} (\text{Wrap}_{l_1} v)) \text{ freeLocs}(C)$$

$$\ggg_N \lambda_. \llbracket C[\text{unwrap}_{l_1} (\text{Wrap}_{l_1} v) / a] \rrbracket_l$$

$$\forall l \in \text{freeLocs}(c_1). \llbracket C \rrbracket_l = \text{Recv } l \ggg_N \lambda x. \llbracket C[x / a] \rrbracket_l$$

$$\forall l \notin \text{freeLocs}(c_1). \llbracket C \rrbracket_l = \text{pure}_N \text{ Empty}$$

$$\llbracket C' \rrbracket_{l_1} = \llbracket c_1[\text{unwrap}_{l_1} (\text{Wrap}_{l_1} v) / a] \rrbracket_{l_1}$$

$$\forall l \in \text{freeLocs}(c_1). \llbracket C' \rrbracket_l = \llbracket c_1[\text{unwrap}_{l_1} (\text{Wrap}_{l_1} v) / a] \rrbracket_l$$

$$\forall l \notin \text{freeLocs}(c_1). \llbracket C' \rrbracket_l = \llbracket c_1[\text{unwrap}_{l_1} (\text{Wrap}_{l_1} v) / a] \rrbracket_l$$

$$= \text{pure}_N \text{ Empty}$$

(by lemma EPP-UNINVOLVED)

The network reductions in this case are identical to case C-COND, but only for locations included in $\text{freeLocs}(c_1)$.

By lemma EPP-UNINVOLVED (defined below), all other locations project to $\text{pure}_N \text{ Empty}$ in both C and C' , and thus need no reduction.

□

Lemma 1 (EPP-UNINVOLVED).

$$\forall C, l. (\Gamma \vdash C : \text{Choreo } m (\tau @ l)) \wedge (l \notin \text{freeLocs}(C))$$

$$\implies (\llbracket C \rrbracket_l = \text{pure}_N \text{ Empty})$$

Proof. By structural induction on the choreography C , witnessing the fact that all choreographic primitives are endpoint-projected to $\text{pure}_N \text{ Empty}$ at locations that they don't involve. □

PEPM '26, January 11–17, 2026, Rennes, France

Alex Ionescu and Alejandro Russo

Received 2025-10-24; accepted 2025-11-28