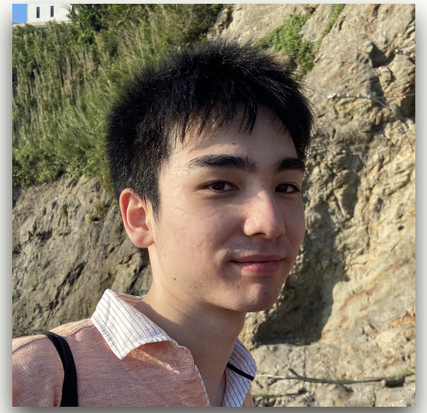
HasChor: Functional Choreographic Programming for All (Functional Pearl)

Gan Shen



Shun Kashiwa

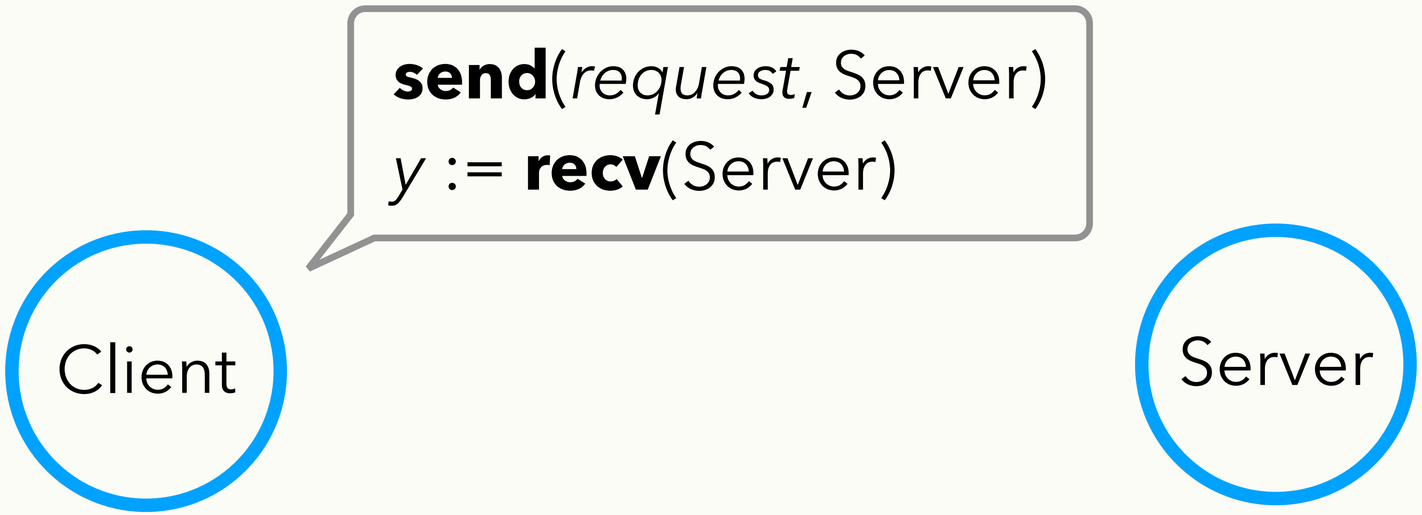
Lindsey Kuper

UC SANTA CRUZ

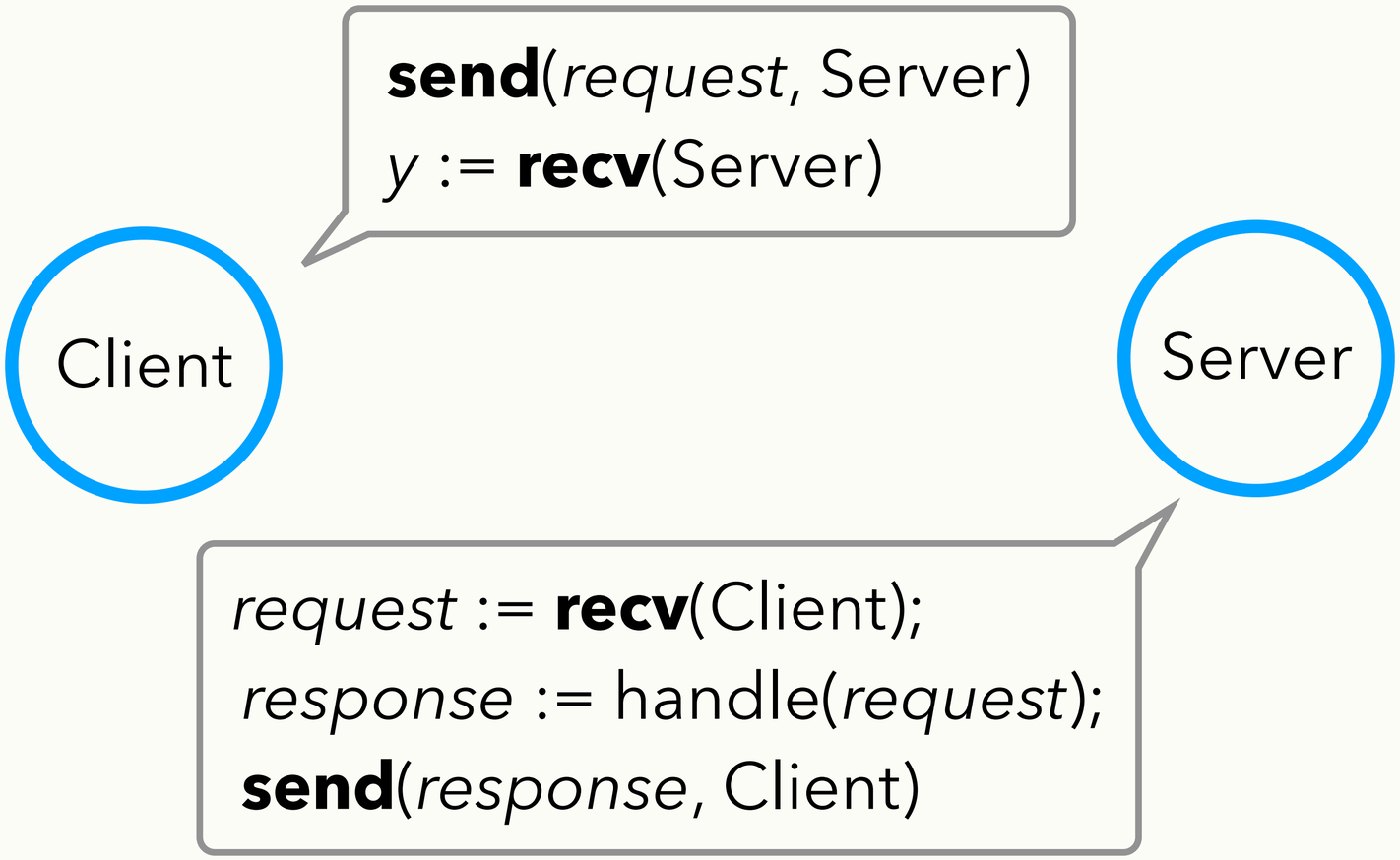
Programming Distributed Systems

A Client-Server Protocol

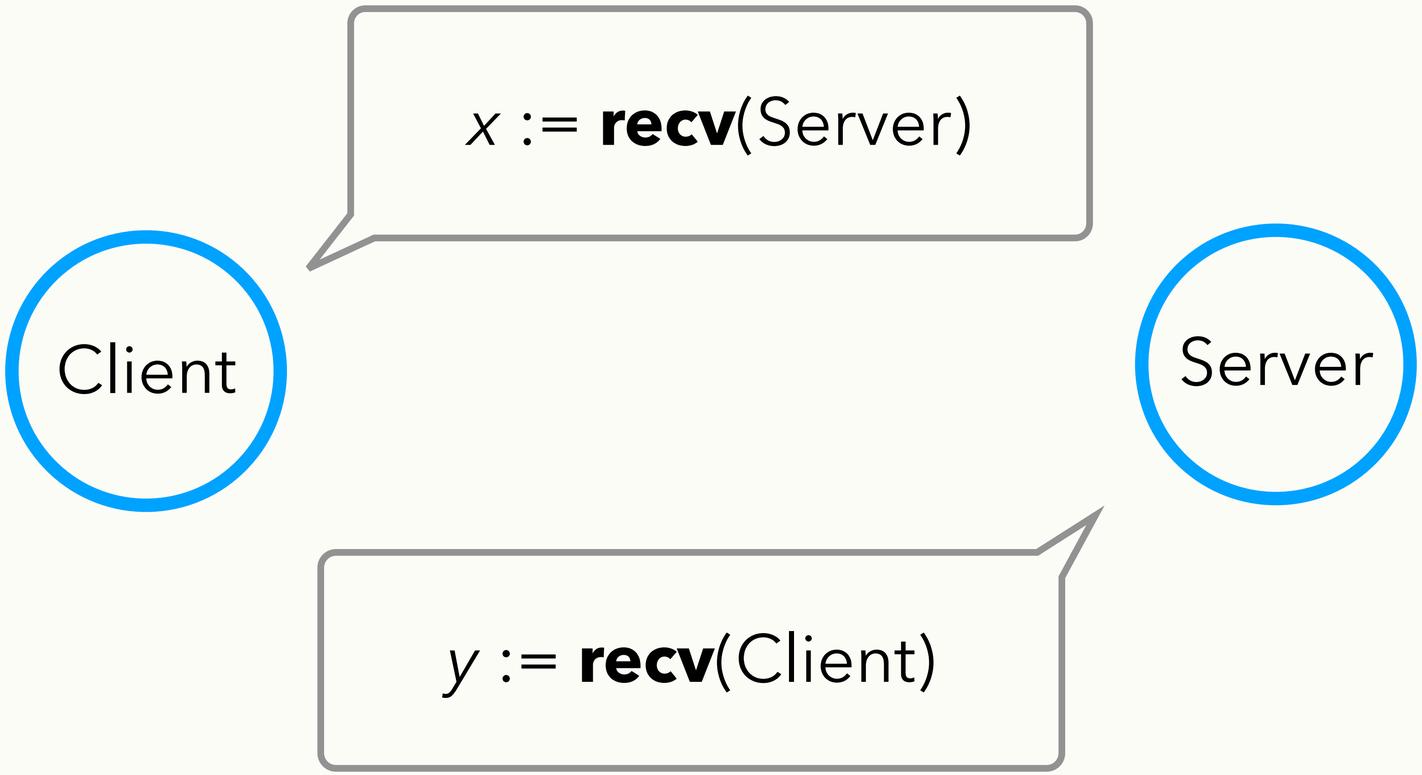
A Client-Server Protocol



A Client-Server Protocol



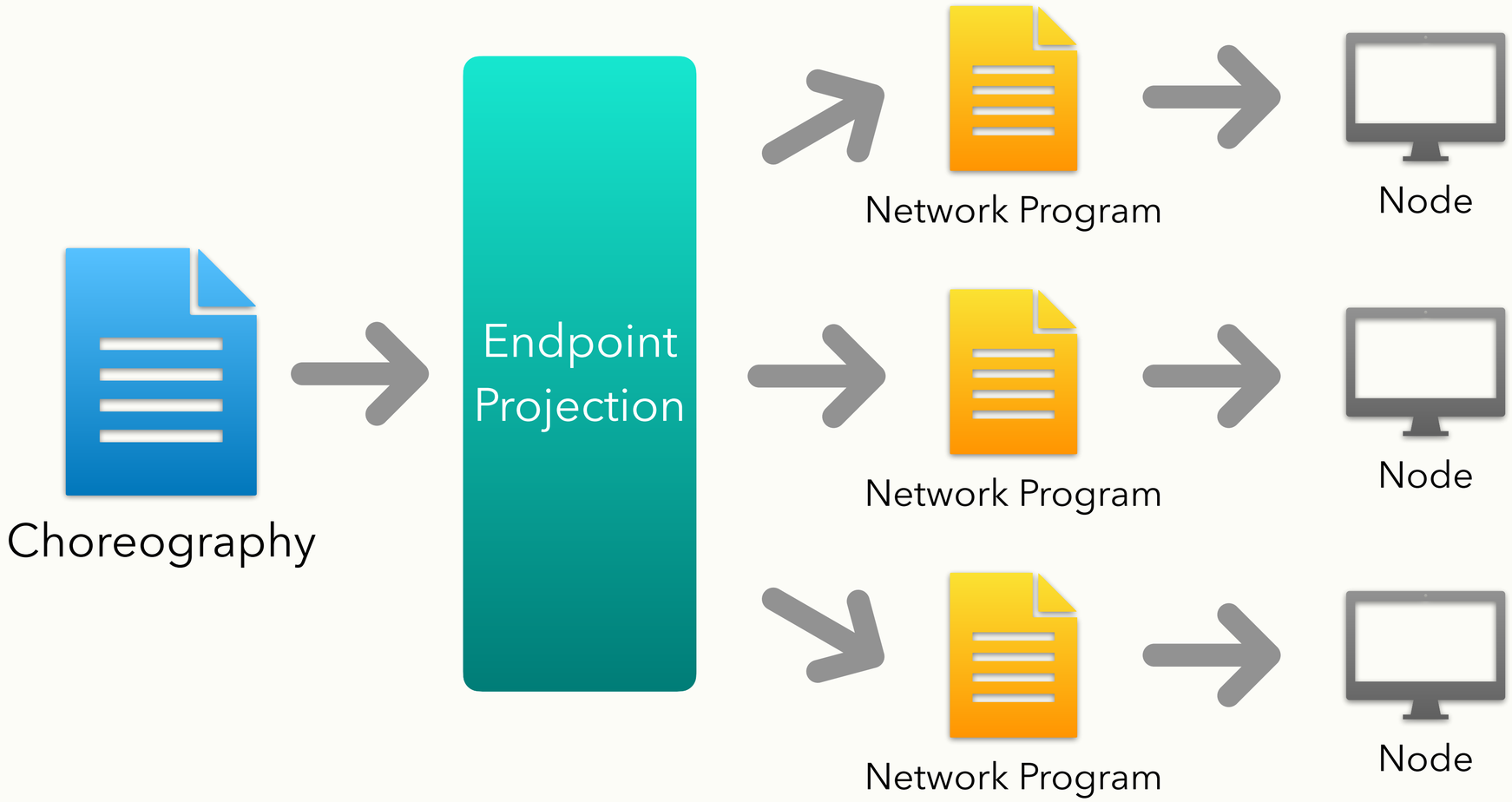
There can be deadlocks!



Choreographic Programming

Choreography

Choreographic Programming



The Client-Server Protocol as a Choreography

Client.*request* ⇝ Server.x;

Server.{ *response* := handle(x) };

Server.*response* ⇝ Client.y

The Client-Server Protocol as a Choreography

An operator for both send and receive

Client.*request* ⇝ Server.x;

Server.{ *response* := handle(x) };

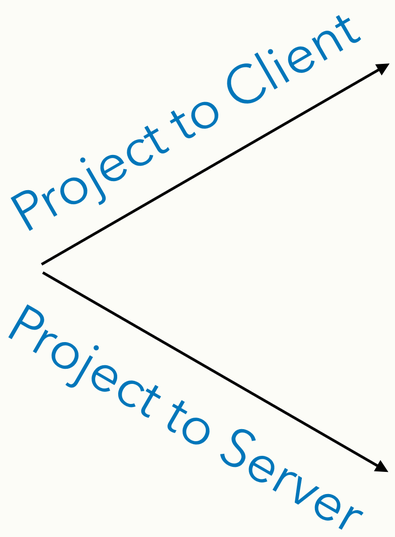
Server.*response* ⇝ Client.y

Endpoint Projection

Client.*request* ⇝ Server.x;

Server.{ *response* := handle(x) };

Server.*response* ⇝ Client.y



send(*request*, Server)
y := **recv**(Server)

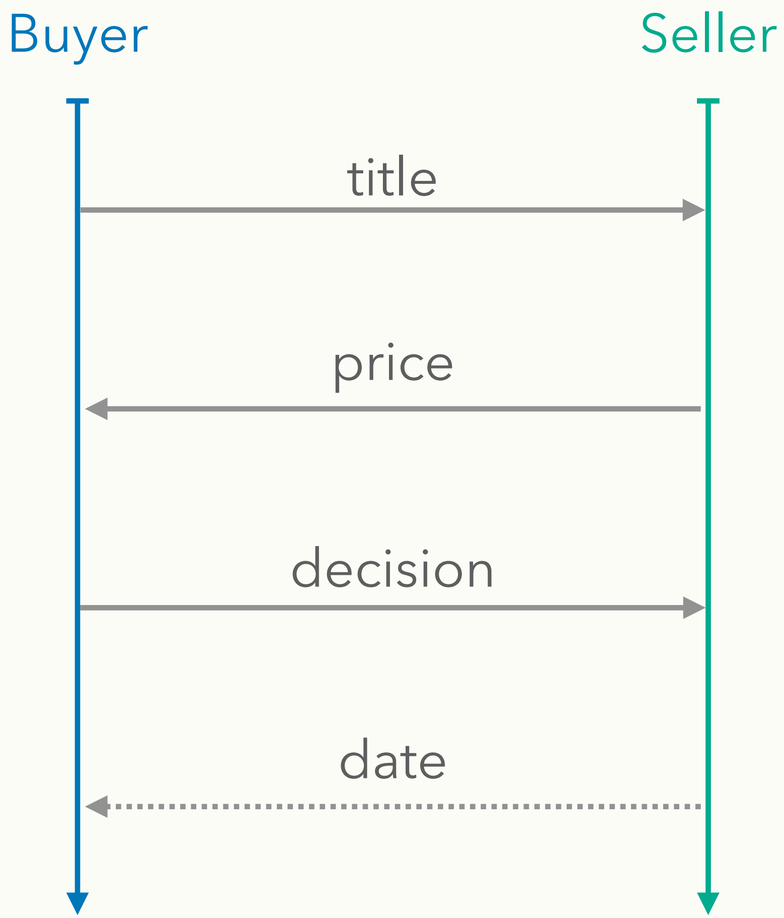
request := **recv**(Client);
response := handle(*request*);
send(*response*, Client)

This Paper

- We present HasChor, an embedded domain-specific language for choreographic programming in Haskell.
- HasChor provides a monadic interface for choreographic programming and integrate well with the existing Haskell ecosystem.
- Our embedding makes heavy use of freer monads, which leads to a natural realization of endpoint projection.

A Tour of HasChor

The Bookseller Protocol



The Bookseller Protocol as a Choreography

```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = …
```

The Bookseller Protocol as a Choreography

The monad for choreographies.

```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = …
```

The Bookseller Protocol as a Choreography

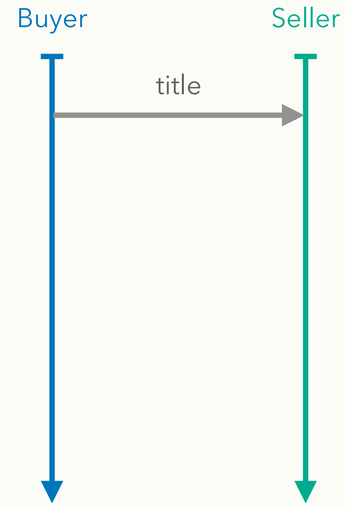
Located values: `a @ l`.

```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = …
```

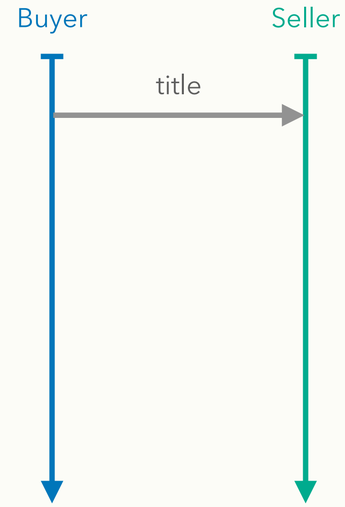
The Bookseller Protocol as a Choreography

The monad for local computation.

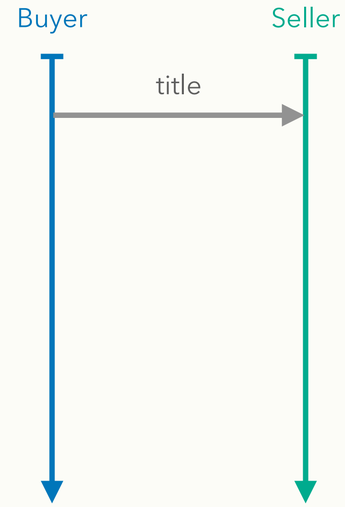
```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = …
```




```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ↝ seller
```



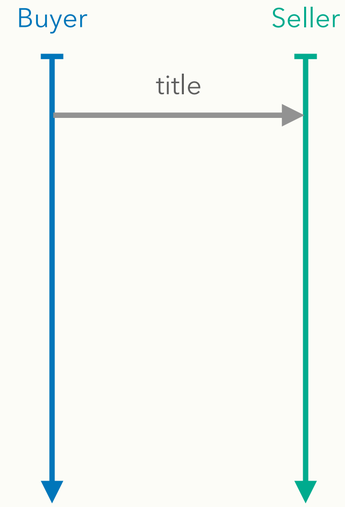

```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ⇝ seller
```



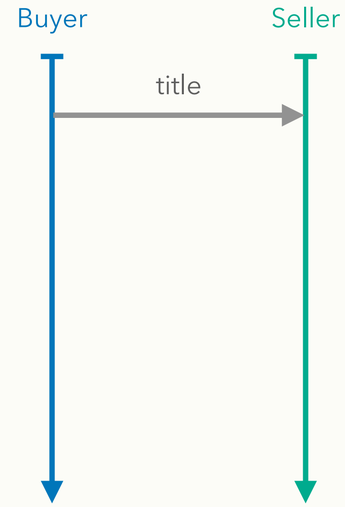

```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ⇝ seller
```

```
locally :: (l :: Loc) → (Unwrap l → m a) → Choreo m (a @ l)
```

```
type Unwrap l = forall a. a @ l → a
```

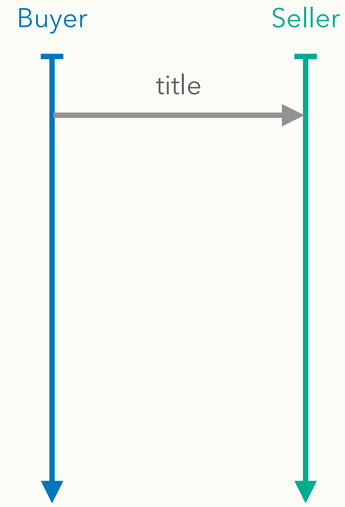



```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ⇝ seller
```

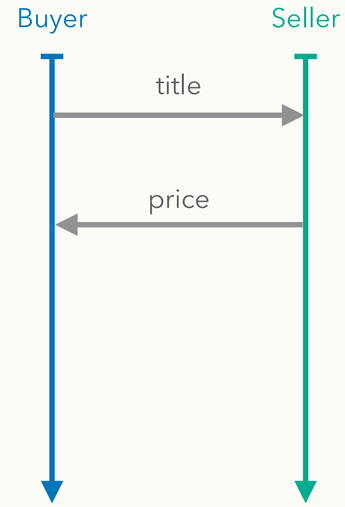



```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ⇝ seller
```

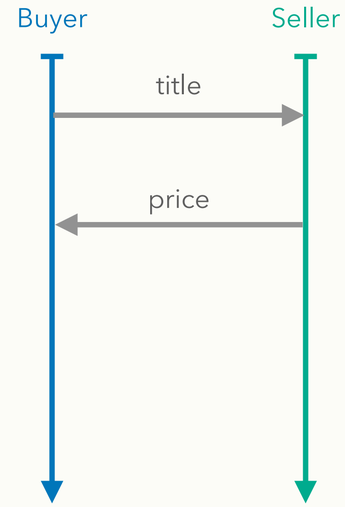
```
(⇝) :: (l :: Loc, a @ l) → (l' :: Loc) → Choreo m (a @ l')
```



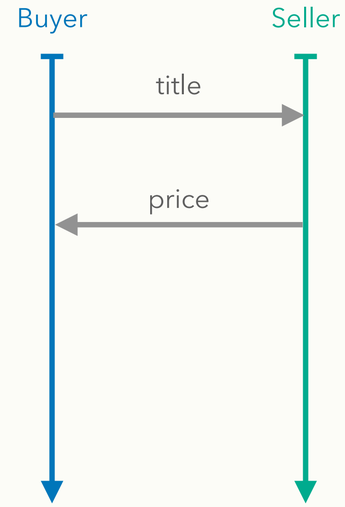

```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ↝ seller
```




```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ↝ seller

  price ← seller `locally` \un → return $ priceOf (un title')
  price' ← (seller, price) ↝ buyer
```

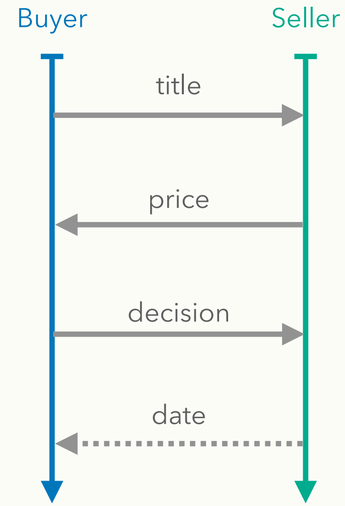



```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ↝ seller

  price ← seller `locally` \un → return $ priceOf (un title')
  price' ← (seller, price) ↝ buyer
```

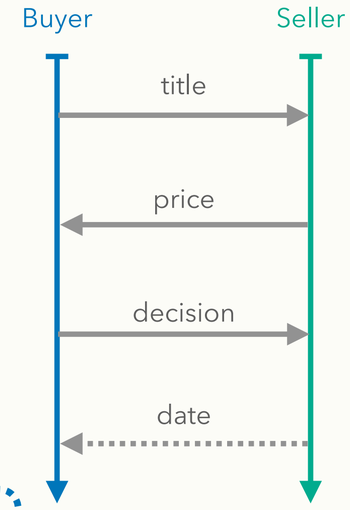



```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ↝ seller

  price ← seller `locally` \un → return $ priceOf (un title')
  price' ← (seller, price) ↝ buyer
```

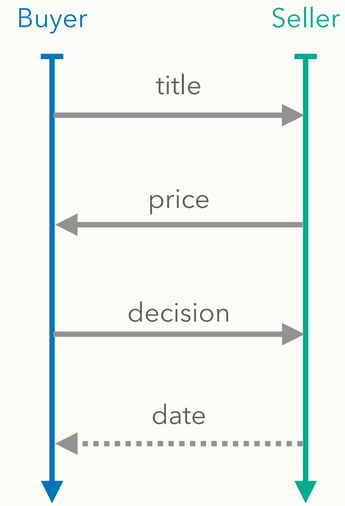
```
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ⇝ seller

  price ← seller `locally` \un → return $ priceOf (un title')
  price' ← (seller, price) ⇝ buyer

  decision ← buyer `locally` \un → return $ (un price') < budget
  cond (buyer, decision) \case
    True → do
      date ← seller `locally` \un → return $ deliveryDateOf (un title')
      date' ← (seller, deliveryDate) ⇝ buyer
      buyer `locally` \un → return $ Just (un date')
    False → do
      buyer `locally` \un → return Nothing
```



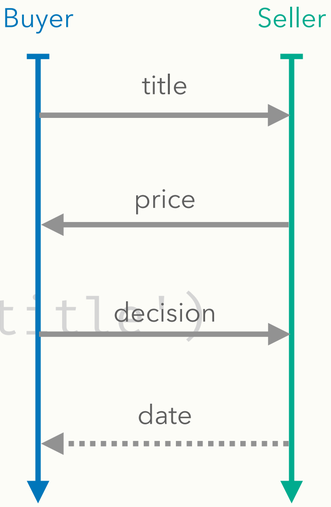
```
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ⇝ seller

  price ← seller `locally` \un → return $ priceOf (un title')
  price' ← (seller, price) ⇝ buyer

  decision ← buyer `locally` \un → return $ (un price') < budget
  cond (buyer, decision) \case
    True → do
      date ← seller `locally` \un → return $ deliveryDateOf (un title')
      date' ← (seller, deliveryDate) ⇝ buyer
      buyer `locally` \un → return $ Just (un date')
    False → do
      buyer `locally` \un → return Nothing
```



```
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ⇝ seller

  price ← seller `locally` \un → return $ priceOf (un title')
  price' ← (seller, price) ⇝ buyer

  decision ← buyer `locally` \un → return $ (un price') < budget
  cond (buyer, decision) \case
    True → do
      date ← seller `locally` \un → return $ deliveryDateOf (un title')
      date' ← (seller, deliveryDate) ⇝ buyer
      buyer `locally` \un → return $ Just (un date')
    False → do
      buyer `locally` \un → return Nothing
```

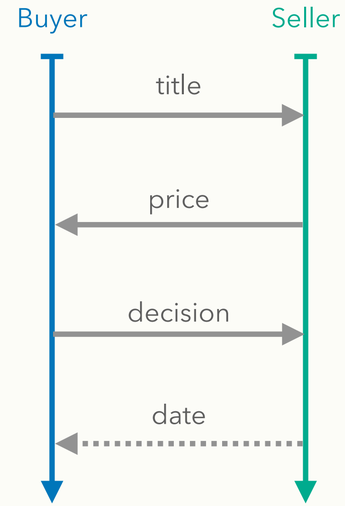


```
decision ← buyer `locally` \un → return $ (un price') < budget
cond (buyer, decision) \case
  True → do
    date ← seller `locally` \un → return $ deliveryDateOf (un title')
    date' ← (seller, deliveryDate) ⇝ buyer
    buyer `locally` \un → return $ Just (un date')
  False → do
    buyer `locally` \un → return Nothing
```

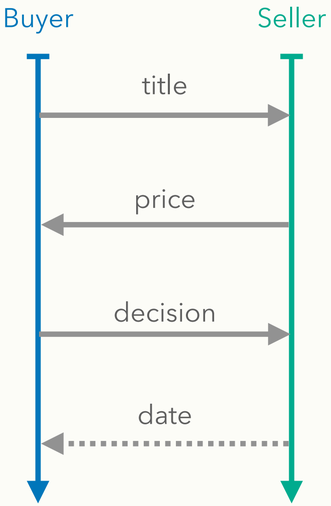



```
cond :: (l :: Loc, a @ l) → (a → Choreo m b) → Choreo m b
```

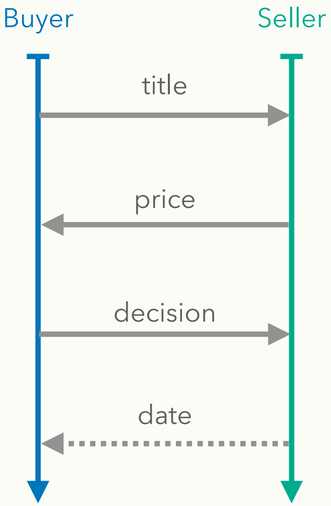
```
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ⇝ seller

  price ← seller `locally` \un → return $ priceOf (un title')
  price' ← (seller, price) ⇝ buyer

  decision ← buyer `locally` \un → return $ (un price') < budget
  cond (buyer, decision) \case
    True → do
      date ← seller `locally` \un → return $ deliveryDateOf (un title')
      date' ← (seller, deliveryDate) ⇝ buyer
      buyer `locally` \un → return $ Just (un date')
    False → do
      buyer `locally` \un → return Nothing
```



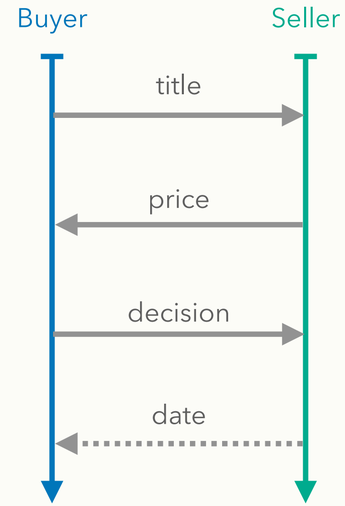
```
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ⇝ seller

  price ← seller `locally` \un → return $ priceOf (un title')
  price' ← (seller, price) ⇝ buyer

  decision ← buyer `locally` \un → return $ (un price') < budget
  cond (buyer, decision) \case
    True → do
      date ← seller `locally` \un → return $ deliveryDateOf (un title')
      date' ← (seller, deliveryDate) ⇝ buyer
      buyer `locally` \un → return $ Just (un date')
    False → do
      buyer `locally` \un → return Nothing
```



```
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ⇝ seller

  price ← seller `locally` \un → return $ priceOf (un title')
  price' ← (seller, price) ⇝ buyer

  decision ← buyer `locally` \un → return $ (un price') < budget
  cond (buyer, decision) \case
    True → do
      date ← seller `locally` \un → return $ deliveryDateOf (un title')
      date' ← (seller, deliveryDate) ⇝ buyer
      buyer `locally` \un → return $ Just (un date')
    False → do
      buyer `locally` \un → return Nothing
```



The Complete Choreography




```
bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = do
  title ← buyer `locally` \un → getLine
  title' ← (buyer, title) ↝ seller

  price ← seller `locally` \un → return $ priceOf (un title')
  price' ← (seller, price) ↝ buyer

  decision ← buyer `locally` \un → return $ (un price') < budget
  cond (buyer, decision) \case
    True → do
      date ← seller `locally` \un → return $ deliveryDateOf (un title')
      date' ← (seller, deliveryDate) ↝ buyer
      buyer `locally` \un → return $ Just (un date')
    False → do
      buyer `locally` \un → return Nothing
```

HasChor Internals

The Monad for Choreographies

Choreo m

- locally
- (⇝)
- cond

The Monad for Network Programs

Choreo m

- locally
- (~>)
- cond

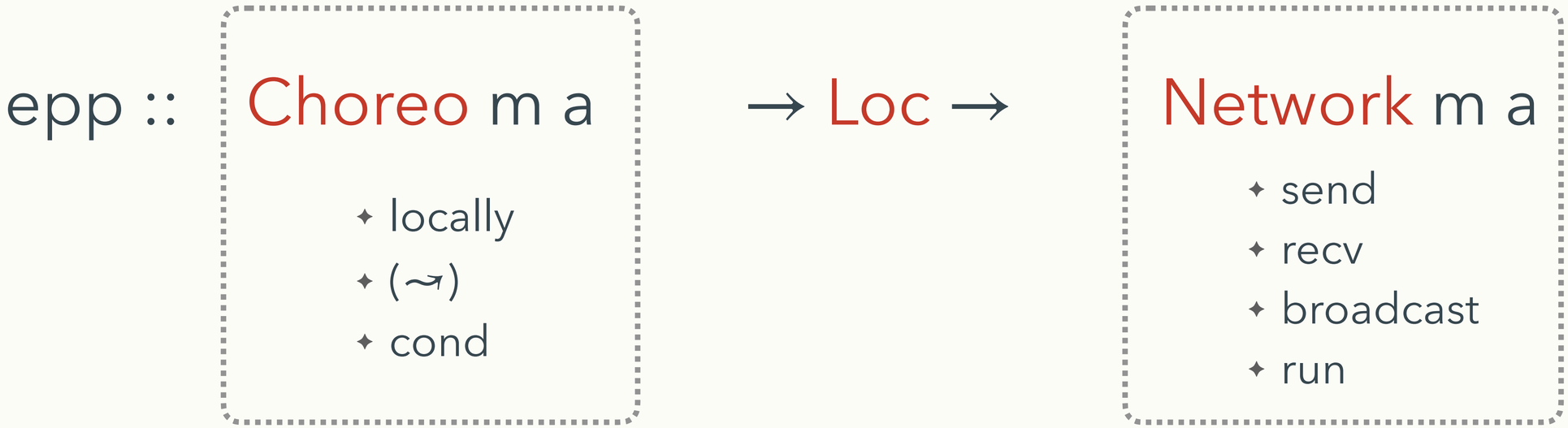
Network m

- send
- recv
- broadcast
- run

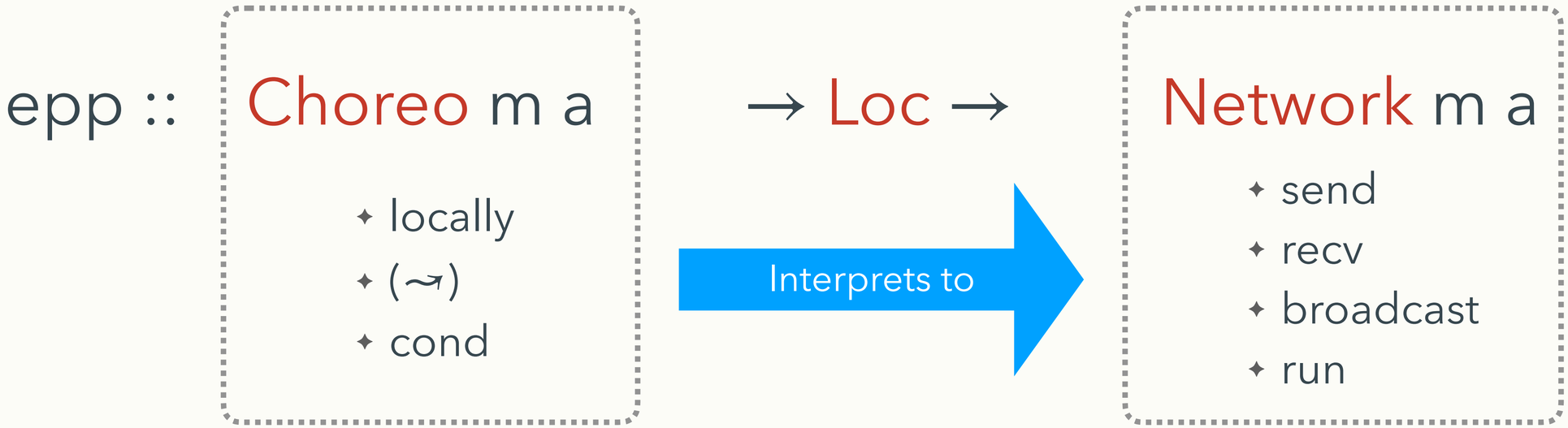
Endpoint Projection Links Them Up

epp :: Choreo m a → Loc → Network m a

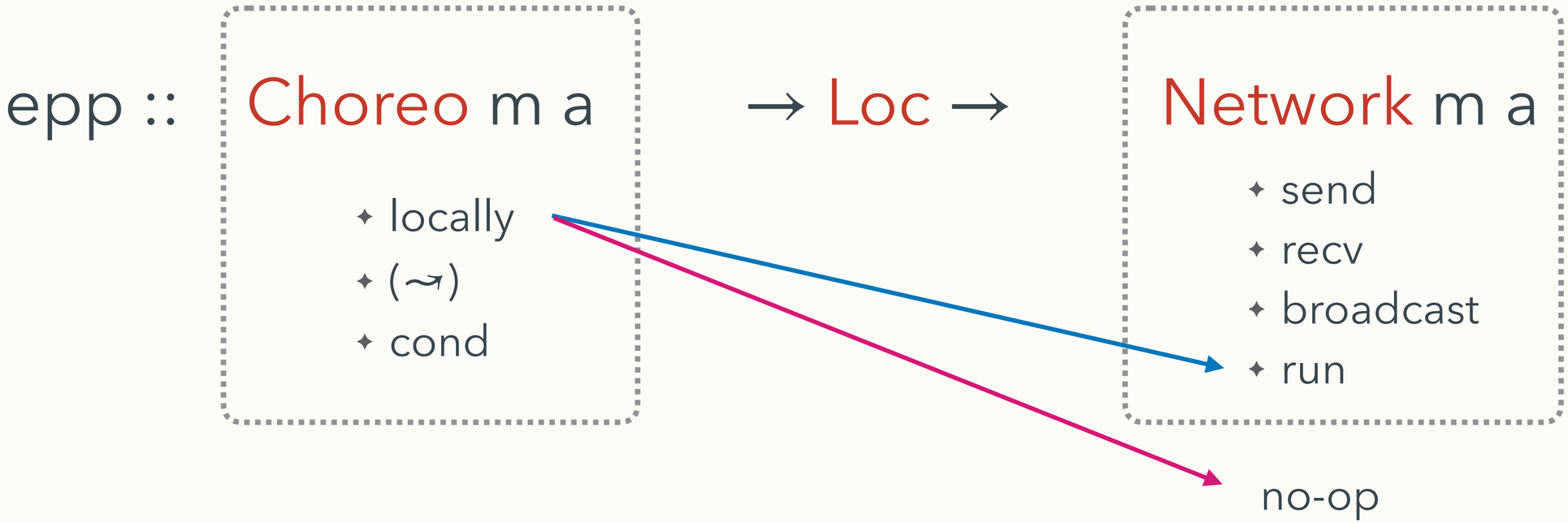
Endpoint Projection Links Them Up



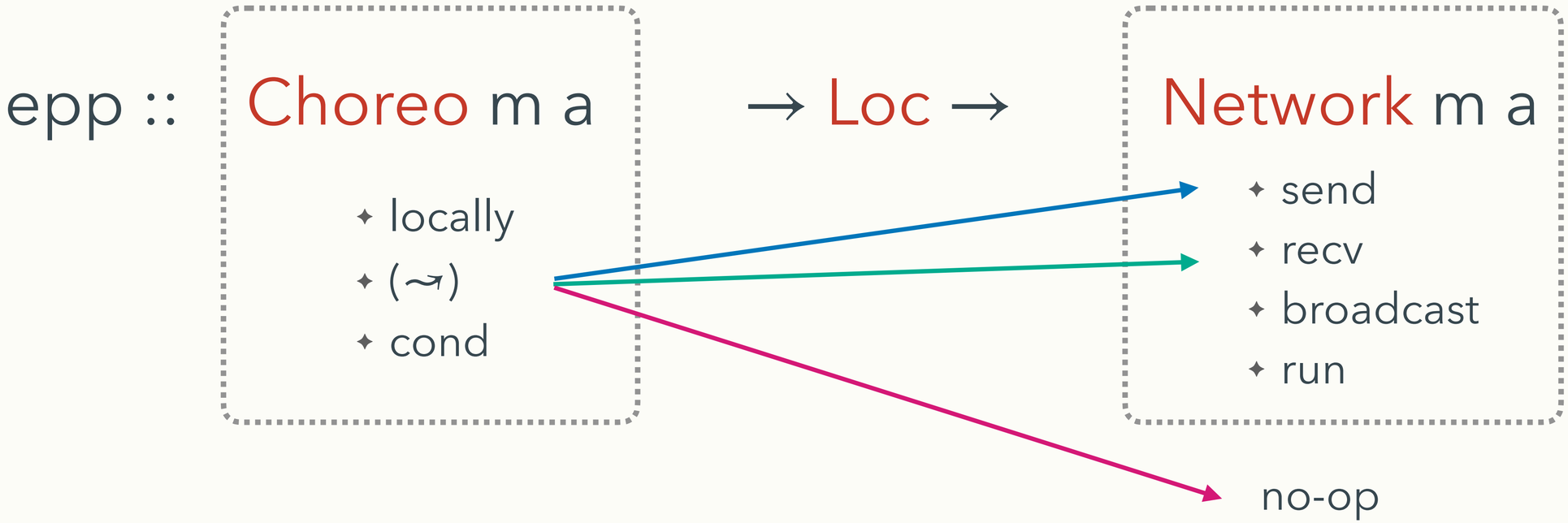
Endpoint Projection Links Them Up



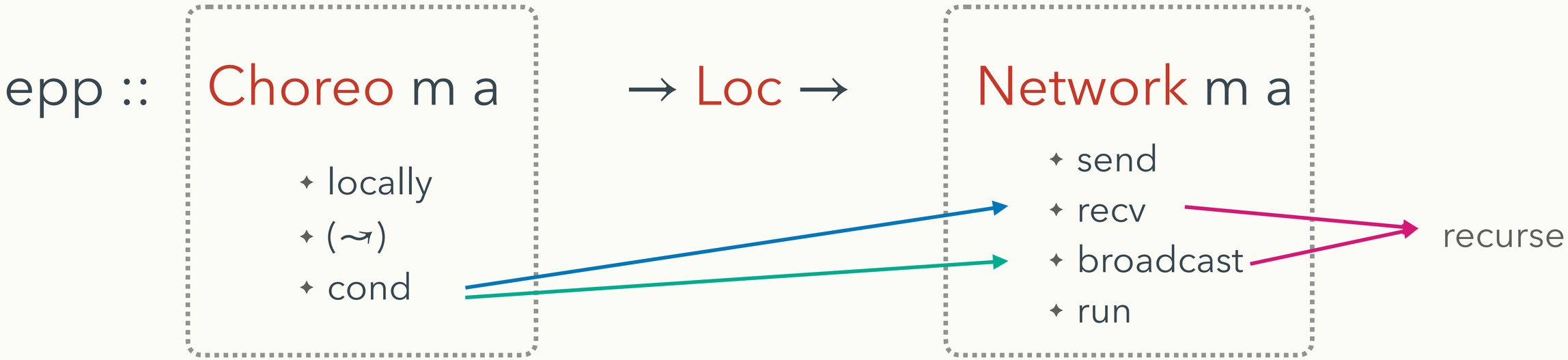
Endpoint Projection Links Them Up



Endpoint Projection Links Them Up



Endpoint Projection Links Them Up



How do we interpret effects in a monad?

How do we interpret effects in a monad?

Freer Monads!

[Kiselyov and Ishii 2015]

Freer Monads, More Extensible Effects

Oleg Kiselyov
Tohoku University, Japan
oleg@okmij.org

Hiromi Ishii
University of Tsukuba, Japan
h-ishii@math.tsukuba.ac.jp

Abstract

We present a rational reconstruction of extensible effects, the recently proposed alternative to monad transformers, as the confluence of efforts to make effectful computations compose. Free monads and then extensible effects emerge from the straightforward term representation of an effectful computation, as more and more boilerplate is abstracted away. The generalization process further leads to freer monads, constructed without the Functor constraint. The continuation exposed in freer monads can then be represented as an efficient type-aligned data structure. The end result is the algorithmically efficient extensible effects library, which is not only more comprehensible but also faster than earlier implementations.

As an illustration of the new library, we show three surprisingly simple applications: non-determinism with committed choice (LogicT), catching IO exceptions in the presence of other effects, and the semi-automatic management of file handles and other resources through monadic regions.

We extensively use and promote the new sort of 'laziness', which underlies the left Kan extension: instead of performing an operation, keep its operands and pretend it is done.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs—Control primitives; F.3.3 [*Logics and Mean-*

the widely used monad transformer library (MTL). They are based on Moggi's original idea of "monads with a hole", adding to it the lifting of monad operations through the transformer stack. The second approach combines monads through a quite complicated coproduct [24], whose simplification has lead to the free monad popularized in Data types à la carte [30]. The third, presented just before monad transformers, looked at effects as an interaction and introduced side-effect–request handlers [7]. That idea of effect handlers, generalizing exception handlers, was picked up in [4, 26], and developed into the language Eff. In Haskell, it was implemented as extensible effects [21] and [17].

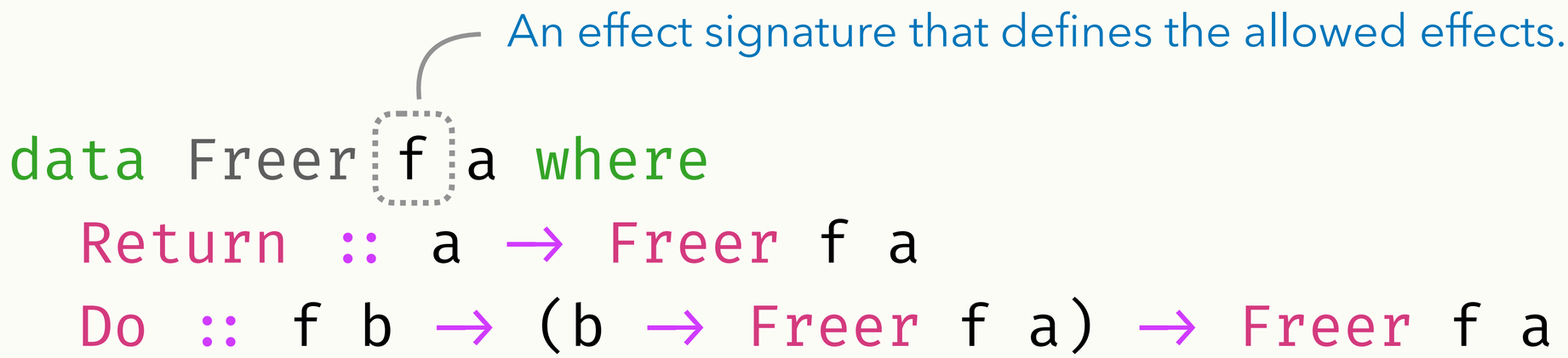
We present, in §2, a unifying view: we derive the free monad and extensible effects by progressively abstracting the straightforward term representation of an effectful computation. Extensible effects emerge as the combination of the ideas of free monads and open union. The unifying, rational reconstruction is not only edifying: it pointed to the further generalization in §2.4: freer monads, free even from the Functor constraint. Freer (or, free-er, for emphasis) monad is an algebraic data type that is a monad by the very construction, just like list is a monoid by construction.

Besides intellectually satisfying, the freer monads are more economical with memory, avoiding rebuilding of the request data structure on each bind operation. Mainly, by exposing the continuation the freer monads made it easier to represent it differently, as a type-aligned sequence data structure [31], which improved the

Freer Monads

```haskell
data Freer f a where
  Return :: a → Freer f a
  Do :: f b → (b → Freer f a) → Freer f a
```

Freer Monads



Freer Monads

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

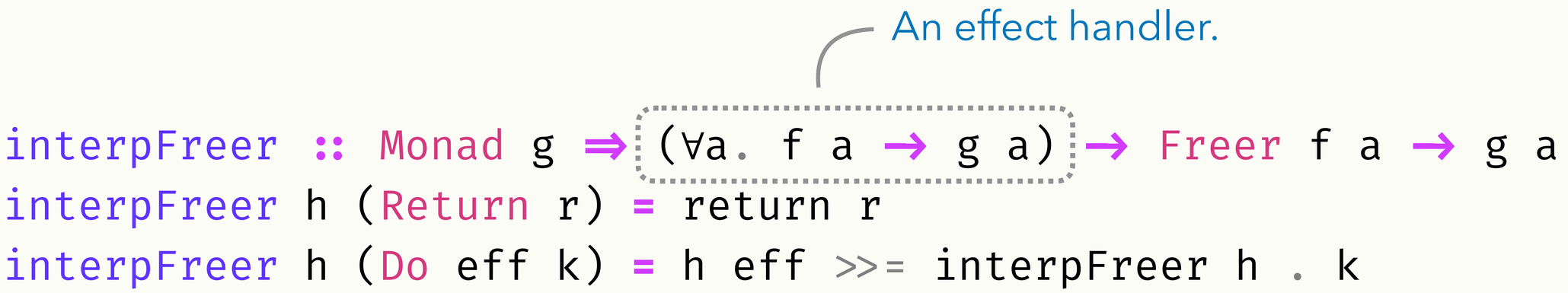
A pure computation.

An effectful computation.

Interpreting Freer Monads

```
interpFreer :: Monad g ⇒ (∀a. f a → g a) → Freer f a → g a
interpFreer h (Return r) = return r
interpFreer h (Do eff k) = h eff >>= interpFreer h . k
```

Interpreting Freer Monads



Interpreting Freer Monads

```
interpFreer :: Monad g ⇒ (∀a. f a → g a) → Freer f a → g a
interpFreer h (Return r) = return r
interpFreer h (Do eff k) = h eff >>= interpFreer h . k
```

Handle the effect, bind the result to the continuation, and recurse.

Choreo and Network are Freer Monads

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

Endpoint Projection = Interpreting Effects, Finally

```haskell
epp :: Choreo m a → Loc → Network m a
epp c l' = interpFreer handler c
  where
    handler :: ChoreoSig m a → Network m a
    handler (Local l m)
      | l == l'   = wrap <$> run (m unwrap)
      | otherwise = return Empty
    handler (Comm s a r)
      | s == l'   = send (unwrap a) r >> return Empty
      | r == l'   = wrap <$> recv s
      | otherwise = return Empty
    handler (Cond l a c)
      | l == l'   = broadcast (unwrap a) >> epp (c (unwrap a)) l'
      | otherwise = recv l >>= \x → epp (c x) l'
```

Endpoint Projection = Interpreting Effects, Finally

```haskell
epp :: Choreo m a → Loc → Network m a
epp c l' = interpFreer handler c
  where
    handler :: ChoreoSig m a → Network m a
    handler (Local l m)
      | l == l'   = wrap <$> run (m unwrap)
      | otherwise = return Empty
    handler (Comm s a r)
      | s == l'   = send (unwrap a) r >> return Empty
      | r == l'   = wrap <$> recv s
      | otherwise = return Empty
    handler (Cond l a c)
      | l == l'   = broadcast (unwrap a) >> epp (c (unwrap a)) l'
      | otherwise = recv l >>= \x → epp (c x) l'
```

More in the Paper :-)

- Swappable message transport backends for the `Network` monad
  - HTTP, Threads and `MVar`s
- Some type shenanigans
  - Approximate dependent types using `Proxy` and `DataKinds` GHC extension
- Advanced choreographic programming features for free
  - Higher-order choreographies, location polymorphism
- Case study on replicated in-memory key-value stores

Conclusion

- We present HasChor, the first embedded domain-specific language for choreographic programming.
- We show freer monads can be used to implement endpoint projection in a natural way.

Email from a User

“Hi,

I tried out HasChor yesterday, made the world's laziest rsync clone. You guys made something really special. **I've never used freer monads and didn't fully grasp what's happening in the type system but it's kind of intuitive anyway.** ~> moves things. locally does stuff well... locally and if the compiler stops you then your value probably isn't on the right machine.

I don't mean to call your stuff simple or obvious. It's not. **What I mean to say is HasChor itself is super accessible.**”

— Jack Wines

Thank you!

HasChor is available at https://github.com/gshen42/HasChor.

Endpoint Projection = Interpreting Effects, Finally

```haskell
epp :: Choreo m a → Loc → Network m a
epp c l' = interpFreer handler c
  where
    handler :: ChoreoSig m a → Network m a
    handler (Local l m)
      | l == l'   = wrap <$> run (m unwrap)
      | otherwise = return Empty
    handler (Comm s a r)
      | s == l'   = send (unwrap a) r >> return Empty
      | r == l'   = wrap <$> recv s
      | otherwise = return Empty
    handler (Cond l a c)
      | l == l'   = broadcast (unwrap a) >> epp (c (unwrap a)) l'
      | otherwise = recv l >>= \x → epp (c x) l'
```

Choreographic Programming

- A single program—a *choreography*—that describes the complete behavior of an entire distributed system
- Send and receive are combined into one operator that guarantees match-up
- A compilation process for generating individual programs to be deployed on each node

Freer Monads

An effectful computation that returns a result of type a.

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