Welcome to the Parti(tioning) (Functional Pearl): Using Rewrite Rules and Specialisation to Partition Haskell Programs

Welcome to the Parti(tioning) (Functional Pearl)

Using Rewrite Rules and Specialisation to Partition Haskell Programs

Robert Krook
Chalmers University of Technology - Gothenburg University
Sweden
krookr@chalmers.se

Samuel Hammersberg
Gothenburg University
Sweden
samuel.hammersberg@gmail.com

Abstract

Writing distributed applications is hard, as the programmer needs to describe the communication protocol between the different endpoints. If this is not done correctly, we can introduce bugs such as deadlocks and data races. Tierless and choreographic programming models aim to make this easier by describing the interactions of every endpoint in a single compilation unit. When such a program is compiled, ideally, a single endpoint is projected and the code for the other endpoints is removed. This leads to smaller binaries with fewer dependencies, and is called *program partitioning*.

In this pearl, we show how we can use rewrite rules and specialisation to get GHC to partition our Haskell programs (almost) for free, if they are written using the Haste App or HasChor framework.

As an example of why partitioning is useful, we show how an example application can be more easily built and deployed after being partitioned.

CCS Concepts: • **Security and privacy** → *Software security engineering*; • **Software and its engineering** → **Source code generation**; **Application specific development environments**; **Client-server architectures**; **Organizing principles for web applications**.

Keywords: Haskell, Program Partitioning, Choreographic Programming, Tierless Programming, Specialisation, Rewrite Rules

ACM Reference Format:
Robert Krook and Samuel Hammersberg. 2024. Welcome to the Parti(tioning) (Functional Pearl): Using Rewrite Rules and Specialisation to Partition Haskell Programs. In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium (Haskell '24), September 6–7, 2024, Milan, Italy*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3677999.3678276

Haskell '24, September 6–7, 2024, Milan, Italy

ACM ISBN 979-8-4007-1102-2/24/09
https://doi.org/10.1145/3677999.3678276

1 Introduction

Traditional distributed applications are implemented by writing several, separate programs (endpoints) that together implement a communication protocol using transmission primitives such as `send` and `receive`. Correctly matching send and receives is tricky to get right, and potential bugs may render an application unable to perform its function. Common problems are e.g. deadlocks and race conditions.

To ease the development of such code, primarily communications code, we may use specific programming models that are intended to make writing such apps easier. Popular examples of such programming models are *tierless* programming or *choreographic* programming.

Both tierless and choreographic programming models allow a developer to write the code for multiple endpoints in a single compilation unit. The advantages of this are many, but the foremost one is that the compiler has more information at hand during compilation, allowing it to e.g. type check communication protocols.

In both programming models, code can be tagged with a location, indicating which endpoint should execute a specific computation. The low-level communication primitives `send` and `receive` are replaced with high-level operations such as `communicate`. During execution, it is known which endpoint should be projected, and e.g. a call to `communicate` will be replaced with a call to `send`, `receive`, or `no-op`.

While being able to describe multiple endpoints in a single compilation unit already gives us some guarantees from our code, such as the already mentioned freedom from deadlocks and race conditions, actually *partitioning* the code may help us even further. By program partitioning, we refer to the act of compiling a program several times, every time indicating which specific endpoint should be projected, and removing all code intended for other endpoints. A couple of benefits of this are

- When executing code inside a *Trusted Execution Environment* (TEE), it is important to minimize the amount of code that has to go into the TEE, as you have to *trust* this code. The amount of code that goes into the TEE is called the *Trusted Computing Base* (TCB). Typically, the code inside the TEE assumes the role of a server, with the client executing outside of the TEE. By using a tierless or choreographic programming model to write

your application, you eliminate certain classes of bugs. However, without actually partitioning the program, there will be a lot of code going into the TEE that is not necessary. By partitioning the program such that only the necessary parts remain inside the TEE, we can reduce the TCB.

- If our program is proprietary and we do not wish to leak information about its function, partitioning may aid us. If an adversary gains access to the executable of a single endpoint, without partitioning, decompilation might yield a significant amount of information. In contrast, if the program was partitioned, decompilation would only yield information about that particular endpoint, as well as its communication protocol with other endpoints. The application in its entirety might still be relatively safe.
- If one endpoint is intended to run on a particular platform, partitioning may make building and deployment easier. Without partitioning, all the code for all endpoints has to go into the binary. Some dependencies may be absent from certain platforms, making deployment harder. If we partition a program, we will only retain those dependencies that we require for an endpoint, and deployment will be easier.
- The size of executables could decrease if we partition programs. By eliminating code that we know we will not need, we will save space on resource-constrained platforms. In the case of the *Glasgow Haskell Compiler* (GHC), where executables are already very large, partitioning might have very little effect on the overall executable size. If we partition a program and compile it with MicroHaskell[1], however, partitioning would have a larger impact on the size of executables. While this is not something we have experimented with, we discuss it in section 10.

Writing a compiler pass that correctly and thoroughly analyses a program to figure out how to partition it is not impossible, as shown by e.g. GoTEE[2] and J_e[5], but not trivial. The compiler pass becomes large and complicated, and errors can be hard to detect. In the case of GoTEE, modifications to the source language were required to facilitate the desired features.

In this pearl, we look at two methods partitioning Haskell programs. Both methods we show are type-driven, lightweight, and automatic, and require very little help from the developer. We use the optimisation phase of GHC to the max to get what we want, and one of our key insights is that we can use a combination of rewrite rules and specialisation to partition Haskell programs even in cases where the compiler does not have enough information to make the right choice[2].

[1] https://github.com/augustss/MicroHs
[2] The code is available here: https://github.com/Rewbert/HasChor

2 Simple Partitioning à la Haste App

As a first example of program partitioning in Haskell, we will look at the technique used by Haste App[1] to write both the server and client endpoints of web applications. HasTEE[6] later adopted this technique to write the trusted and untrusted endpoints of applications running inside TEEs.

Haste App is used to specify the actions of a single server and a single client. The computations are organised around three monads, called `App`, `Client`, and `Server`. The `App` monad is used for staging, whereas the `Client` and `Server` monads describe the actions of the client and server, respectively.

To show how to use these monads in action, we will implement a classical example, the bookseller protocol. The application will describe the actions of a client who wishes to purchase a book from a store (the server). The server maintains a state of stock and prices, which we model as regular Haskell data types

```
type Price = Int
type Day = Int
type Store = Map String (Price, Day)
```

The server will keep a value of the `Store` in a mutable reference, to reflect stock changes. To enable the client to query the server for information about its stock, we define the following functions

```
storeLookup :: Server (Ref Store)
            -> String -> Server (Price, Day)
storeLookup secBooks book = do
    booksRef <- secBooks
    books    <- readRef booksRef
    return . fromJust $ Map.lookup book books

getPrice :: Server (Ref Store) -> String -> Server Price
getPrice books book = do
    (price,_) <- storeLookup books book
    return price

getDay :: Server (Ref Store) -> String -> Server Day
getDay books book = do
    (_, deliveryDate) <- storeLookup books book
    return deliveryDate
```

With these functions in place, how can the client communicate with the server? Below follows the rest of the bookseller program

```
-- liftNewRef :: a -> App (Server (Ref a))

bookseller :: App Done
bookseller = do
  remoteRef <- liftNewRef defaultStock
  remoteGetPrice <- inServer (getPrice remoteRef)
  remoteGetDay <- inServer (getDay remoteRef)

```

```
runClient $ do
  liftIO      $ putStrLn "Title of the book to buy"
  userInput <- liftIO  getLine
  price     <- gateway (remoteGetPrice <@> userInput)

  if  price < clientBudget then do
    day     <- gateway (remoteGetDay <@> userInput)
    liftIO  $ putStrLn $ "Delivery date: " <> show day
  else do
    liftIO  $ putStrLn "Price is out of the budget!"
```

Lines 5,6, and 7 happen in the App monad, where staging is performed. We first use `liftNewRef` to create the mutable reference on the server, modeling the current stock. We then indicate that the client will want to invoke `getPrice` and `getDay`, by applying `inServer` to them. The result of `inServer` is a handle from which the client can invoke the corresponding function on the server. The client fully applies the handle by using the `<@>`-operator, after which the `gateway` function performs the remote function invocation. Even though the return type of the remote functions is in the `Server` monad, `gateway` brings the value into the `Client` monad. We emphasize that the client can only invoke remote functions by first acquiring a remote handle from `inServer`.

So, now that we've seen how we can write a program using Haste App, how is it compiled and executed? The trick is that there exist two distinct implementations of the Haste App API. One implements the client functionality, and the other the server functionality, and only one such module is included during compilation. We thus compile the same program twice – once for the client and once for the server. As a concrete example of what these two implementations look like, let us look closer at the `inServer` function.

During execution, the client will drive the computation, whereas the server enters a state where it is ready to receive synchronous requests from the client. This is made possible by *both* client and server executing the App code. When the client calls `inServer`, it generates an identifier that will be used to identify the remote function. The argument to `inServer` is actually discarded

```
inServer :: Remotable a => a -> App (Remote a)
inServer _ = App $ do
  (next_id, remotes) <- get
  put (next_id + 1, remotes)
  return $ Remote next_id []
```

When this remote handle is applied to an argument with `<@>`, the second field (a list) is populated with the arguments. `gateway` will take such a remote handle, serialise it, and transmit it to the server.

How does the server know what to do with such a closure-like value? Well, the implementation of `inServer` for the server generates *the same* identifier and then associates it to the given function in a map.

```
inServer :: Remotable a => a -> App (Remote a)
inServer f = App $ do
  (next_id, remotes) <- get
  put (next_id + 1, (next_id, \bs ->
    let Server n = mkSecure f bs in n) : remotes)
  return RemoteDummy
```

When an identifier and list of arguments arrive, the server will consult its map and fetch the associated function, apply it, and then return the result. Since the server will not apply any remote closures to parameters, a dummy remote is returned instead.

```
onEvent :: [(Int, Method)]
        -> ByteString'
        -> Socket
        -> IO ()
onEvent mapping incoming socket = do
  -- (Int, [ByteString])
  let (identifier, args) = decode incoming
      Just f = lookup identifier mapping
  result <- encode <$> f args
  -- the () type cannot be sent over wire
  let res = handleVoidTy result
  sendLazy socket (B.append (msgSize res) res)
  where
    msgSize r = encode $ B.length r
    -- the () type has msg length 0
    handleVoidTy r = if (B.length r == 0)
                     then encode '\0'
                     else r
```

When the server component is compiled, the client monad is replaced with a dummy monad, and vice versa.

```
-- when compiling the server endpoint
data Client a = Dummy
newtype Server a = Server (IO a)

-- when compiling the client endpoint
newtype Client a = Client (IO a)
data Server a = Dummy
```

This, together with the fact that `inServer` on the client side discards its argument, lets GHC optimise away the code intended for the other endpoint. If we look at the compilation for the server endpoint, every function returning a `Client a` computation will end up as `\_ -> \_ -> ... \_ -> Dummy` before machine code is generated, giving GHC a lot of opportunities to remove dead code.

3 Haste App and Rewrite Rules

Haste App manages to partition Haskell programs by compiling a program twice, replacing the semantics of the API with each compilation. To actually throw away code and partition the program we rely on GHCs optimisations. While it in practice seems that GHC can be trusted to optimise away

the unwanted code, it is a bit shaky. The inner workings of the GHC optimiser can change at any time! We would like to take matters into our own hands (almost) to get stronger guarantees that partitioning actually happens.

One of GHCs optimisation phases is applying rewrite rules. The optimiser applies rewrite rules until no more rules can be applied, and if we can use rewrite rules to partition our program, we might be less at the mercy of GHC than before.

Rewrite Rules. Rewrite rules, as implemented in GHC, is a mechanism for library developers to add domain-specific optimisations to their code, often to do short-cut fusion (eliminate intermediate data structures). The canonical example is that of *map fusion*. Consider the code below

```
map f (map g xs)
```

We first map the function `g` over the input list `xs`, and then we map the function `f` over it. Not only do we traverse the list twice, but we also produce an intermediate list after the first traversal. We can convince ourselves that by composing `f` and `g`, we can write the semantically equivalent

```
map (f . g) xs
```

This version does only one pass over the input list, and the intermediate result is gone. While it is simple for us to understand that these two terms are equivalent, it is difficult for a compiler to automatically figure that out. Using rewrite rules, we can *tell* the compiler that this is an equivalence, and have it apply the optimisation for us!

In GHC, we specify the corresponding rule as

```
{-# RULES "map/map" forall f g xs .
                    map f (map g xs) = map (f . g) xs #-}
```

We are saying that *for any functions f and g, and any input list xs, if you see the term denoted by the left-hand side, you should rewrite it to the right-hand side.* GHC checks that the rewrite rule does not change the type of a term, but does nothing to verify that the rule encodes an equivalence. It trusts you to write correct rules. GHC promises to apply the rule whenever the opportunity presents itself.

Now, one thing to remember is that GHC can only apply the rule if it can recognise the call to `map`. If we are unlucky, another optimisation will have made it impossible for us to apply the rule. If the code for `map` is *inlined* at the call site, our rewrite rule will not be able to fire. In the case of `map` this is no concern since GHC will not inline recursive functions (if it did, it would never stop!), but, as we will see, in the context of Haste App we do have to remember this. We can explicitly tell GHC to not inline a function by a `NOINLINE` pragma.

```
{-# NOINLINE map #-}
```

Another mechanism that can be used to control if and when terms are inlined or rewritten is *phase control*. While it is out of scope for this pearl, we mention it here for the curious reader to look up for themselves.

An important detail of rewrite rules is that *there can be no ambiguity of which rule to apply*. If more than one rule can be applied to a term, GHC is free to choose either of them. This means that we need to be very careful when we define our rules so that only one rule is applicable at a single time. Consider the function below

```
f :: Int -> Bool
f 0 = True
f _ = False
```

We may be tempted to write the two rules

```
{-# RULES
  "f/0"           f 0 = True
  "f/_" forall x . f x = False
#-}
```

and expect the first rule to be applied when we see `f 0`. However, the second rule also matches, and GHC is free to pick either. If our term has very few variants we can emit rules for *all* cases, but this very quickly becomes infeasible.

Haste App using Rewrite Rules. Now, knowing how GHC applies rewrite rules, we can modify our two different Haste App implementations to instead happen via rewrite rules. Circling back to the example of `inServer`, the corresponding rules become

```
{-# NOINLINE inServer #-}
{-# RULES "HASTEERULES inServer/Server" forall f .
      inServer f = App $ do
        (next_id, remotes) <- get
        put (next_id + 1, (next_id, \bs ->
          let Server n = mkSecure f bs in n) : remotes)
        return RemoteDummy #-}
inServer :: Remotable a => a -> App (Remote a)
inServer _ = error "inServer error"
```

for the server, and

```
{-# NOINLINE inServer #-}
{-# RULES "HASTEERULES inServer/Client" forall f .
      inServer f = App $ do
        (next_id, remotes) <- get
        put (next_id + 1, remotes)
        return $ Remote next_id [] #-}
inServer :: Remotable a => a -> App (Remote a)
inServer _ = error "inServer error"
```

for the client. We make the corresponding change for every function in the Haste App API, after which GHC will still be able to partition the program for us. When we compile the client component, in the case of `inServer`, the rewrite rules will have eliminated any call to functions in the `Server` monad, and dead-code elimination can eliminate those functions. This elimination happens because the rewrite rules have `f` appear on the left-hand side, but not on the right-hand side!

GHC Core Plugin for Verification. How can we be sure that partitioning actually happened? The compiler's representation of a program before machine code is emitted is called *core*. We have written a *Core Plugin* which runs after all optimisations have been applied. It traverses the final core and reports an error if it encounters any calls to the API functions of Haste App. If any call remains, the program was not completely partitioned. If partitioning happens as it should, our plugin will report

```
*** Partitioning verification pass running on Main
*** Verification for module Main: done
```

whereas if we remove a rewrite rule and run it again, it will report an error

```
*** Partitioning verification pass running on Main
<no location info>: error:
    Found indications of partitioning not occurring when
    inspecting module: Main

    The problematic symbol that was encountered: inServer
```

Scalability of Haste App. Haste App gives us a nice, simple framework for describing the interactions of a client and a server. What if we want to describe the interactions of a server and *two* clients? Then we would have to add yet another monad, with a third set of semantics. This does not scale very well. We could instead use a more general framework, such as HasChor[8].

4 HasChor

HasChor is a framework for describing *choreographic* programs. Choreographic and tierless programming are very similar, but there are some subtle differences. Primarily, in a choreographic framework there are no tiers, and every endpoint is considered an equal.

Rather than identifying an endpoint by a particular monad, the HasChor API defines endpoints via proxy values whose type parameters are string literals.

```
client :: Proxy "client"
client = Proxy
server :: Proxy "server"
server = Proxy
```

The rest of the API is fairly short, but very expressive

```
data a @ (l :: LocTy) -- values tagged with a location
type Choreo m a       -- choreographic monad

-- computations tagged with a location
locally :: KnownSymbol l
        => Proxy l -- location performing a computation
        -> ((forall a . a @ l -> a) -> m a)
        -> Choreo m (a @ l) -- result located at l

-- send a value from one location to another
(~>) :: (Show a, Read a, KnownSymbol l, KnownSymbol l')
     => (Proxy l, a @ l) -- value tagged with a location
     -> Proxy l'         -- the receiving location
     -> Choreo m (a @ l')

-- synchronised branching on a value
cond :: (Show a, Read a, KnownSymbol l)
     => (Proxy l, a @ l)
     -> (a -> Choreo m b)
     -> Choreo m b
```

How do we ensure that only the client endpoint unwraps and accesses the value inside of a tagged value? Everything happens in the context of a `Choreo` monad. The API function `locally` denotes computations that are local to a specific endpoint, and the `Choreo` monad supplies an unwrap function as a parameter that can be used to unwrap tagged values. The type system ensures that only computations local to a specific endpoint may unwrap values existing on that endpoint, by ensuring that the polymorphic location variable `l` is the same in the first and second argument. The code below, for instance, indicates that the client is reading a line from standard input. The result of the entire call to `locally` has type `Choreo IO (String @ "client")`.

```
client `locally` \_ -> getLine
```

The located string can be transmitted to the server endpoint by using the ~> function.

```
1 do strAtClient <- client `locally` \_ -> getLine
2    strAtServer <- (client, strAtClient) ~> server
3    server `locally` \uw -> putStrLn $ uw strAtServer
```

Here we can also see on line 3 that, after the message has been transmitted to the server on line 2, the server *unwraps* it (inside `locally`) and prints it. If the server tried to do this with `strAtClient`, instead, a type error would have been thrown.

The last API function of HasChor is cond. To motivate the need for cond, consider the code below

```
case x of
  0 -> (client, someval) ~> server
  1 -> do (client, someval)  ~> server
          (client, someval') ~> server
  _ -> ...
```

If both endpoints do not agree on what the value of the variable x is, they might pick different branches. The different branches have different numbers of calls to ~>, so a deadlock would occur if different branches were taken.

If we instead say that we want the endpoints to branch on *one* value, located at a *specific* endpoint, we are describing what cond does. Consider the code below

```
cond (client, xAtClient) $ \x -> case x of
  0 -> (client, someval) ~> server
  1 -> do (client, someval)  ~> server
```

```
4           (client, someval') ~> server
5   _ -> ...
```

cond will broadcast the scrutinee to all other endpoints, where they will use the received value to pick the correct branch. The bookseller example implemented in HasChor can be found in appendix A.

HasChor is not Partitioned. In contrast to Haste App, HasChor programs are not actually partitioned. To understand what we mean by this, let us look at how a HasChor program is executed. We omit certain details that are not important and look at the `Choreo` monad.

```
1  type LocTm = String
2
3  data ChoreoSig m a where
4    Local :: (KnownSymbol l)
5          => Proxy l
6          -> (Unwrap l -> m a)
7          -> ChoreoSig m (a @ l)
8
9    Comm :: ( Show a, Read a
10           , KnownSymbol l, KnownSymbol l')
11         => Proxy l
12         -> a @ l
13         -> Proxy l'
14         -> ChoreoSig m (a @ l')
15
16   Cond :: (Show a, Read a, KnownSymbol l)
17         => Proxy l
18         -> a @ l
19         -> (a -> Choreo m b)
20         -> ChoreoSig m b
21
22 type Choreo m = Freer (ChoreoSig m)
```

HasChor is implemented using *freer monads*[4]. While the full details of freer monads are out of scope for this pearl, we emphasise that freer monads separate the interface of an effectful monad and the implementation. The code above specifies the interface, and we note that the constructors and their types precisely match those of the HasChor API. In fact, the HasChor API is implemented by constructing values of the `ChoreoSig` data type and wrapping them in the `Freer` monad.

```
1  locally l m    = toFreer (Local l m)
2  (~>) (l, a) l' = toFreer (Comm l a l')
3  cond (l, a) c  = toFreer (Cond l a c)
```

To project a single endpoint, we need to move from a high-level API, like the HasCHor API, to a more low-level API that has the primitives `send` and `receive`. For this, we have the `Network` monad

```
1  data NetworkSig m a where
2    Run   :: m a -> NetworkSig m a
3    Send  :: Show a => a -> LocTm -> NetworkSig m ()
4    Recv  :: Read a => LocTm -> NetworkSig m a
5    BCast :: Show a => a -> NetworkSig m ()
6
7  type Network m = Freer (NetworkSig m)
```

Note that there are no more `KnownSymbol` constraints here. A `NetworkSig` computation no longer has any location ambiguities to resolve. The function that takes a `Choreo` computation, as well as the location that should be projected, and returns a `Network` computation, is called *Endpoint Projection*. The implementation is shown below, with some bits omitted for brevity. We ask the reader to focus on the cases in the `handler` function

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

Here is where the magic happens. When a choreographic operation is executed (by `handler`), the given `LocTm` is compared to the endpoints involved in the operation, and a decision is made on whether to do something or not.

A call to `locally` indicates that a computation only happens on a specific endpoint. On line 9 we can see that the given `LocTm` is compared to the location that is supposed to perform the operation. If they are the same, the computation is performed, and otherwise nothing is done. Similarly, when `~>` (`Comm`) is handled, we compare the given `LocTm` against both the sending and receiving location, to figure out if `Comm` should become a `Send`, `Recv`, or a no-op (`Empty`). When `cond` is handled, we either broadcast or receive the value before we apply the continuation to it, based on where the value is located.

A key property of the `epp` function is that all these choices are *made at runtime*. It is *not* known at compile time which endpoint will be projected, meaning that all the code for all the endpoints has to be embedded in the compiled binary.

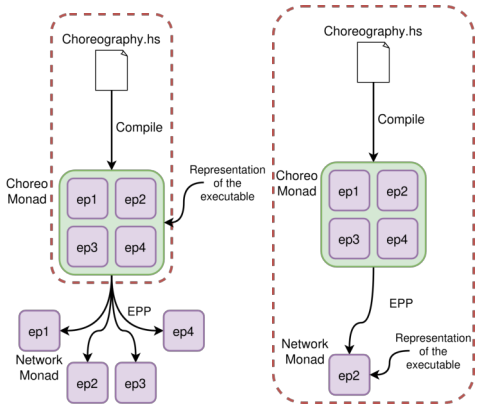

Figure 1. A representation of how HasChor programs are compiled, and of what we would instead like to happen. The diagrams use a dotted line to show what is done during compilation. The compiled binary contains code for all endpoints, and then endpoint projection happens at runtime, where we can project any one of them. What we would like, instead, is to be able to make endpoint projection happen at compile time, yielding an executable that contains code for just one endpoint.

Figure 1 illustrates what the problem is, and what we would like to do instead.

5 HasChor and Rewrite Rules

Well, we saw that in the case of Haste App, we could achieve true partitioning via rewrite rules! Can we apply the same trick to partition HasChor programs? There are two things that we need to do to achieve this.

- We need to *fix* the endpoint we wish to project at compile time, rather than delaying that decision to runtime
- We need to figure out what the rewrite rules *should say*

HasChor uses the `Freer` monad to build up the `ChoreoSig` structure, from which we can project a single endpoint in the `Freer` monad, but with the effects described in the `NetworkSig` structure. What we want our rules to do is to achieve a kind of *short-cut fusion*[3] where we eliminate this intermediate `ChoreoSig` structure, and instead construct the projected `NetworkSig` structure directly. We redefine the `Choreo` monad to be defined as

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

Let us consider the bookseller protocol, with two endpoints. We are going to require two sets of rewrite rules – one specifying the behavior of the seller, and one specifying the behavior of the buyer. Let's consider the semantics of the API functions if we assume that we compile the buyer endpoint. We use the shorthand `Proxy "buyer"` and `Proxy "seller"` to mean `Proxy :: Proxy "buyer"` and `Proxy :: Proxy "seller"`.

```
locally (Proxy "buyer") f  == wrap <$> run (f unwrap)
locally (Proxy "seller") f == return Empty

(Proxy "buyer", v) ~> r == do send (unwrap v) (toLocTm r)
                              return Empty
(s, v) ~> (Proxy "buyer") == wrap <$> recv (toLocTm s)

cond (Proxy "buyer", v) c  == do broadcast (unwrap v)
                                 c (unwrap v)
cond (Proxy "seller", v) c ==
    do x <- recv (toLocTm (Proxy "seller"))
       c x
```

We will encode precisely these equations as rewrite rules. Only the rules for `locally` will be shown here for brevity, while the full set of rules can be found in appendix B.

```
{-# NOINLINE locally #-}
{-# RULES
"CHOREORULES locallybuyer"
    forall (f :: (forall a . a @ "buyer" -> a) -> m a)
           (buyer :: Proxy "buyer") .
           buyer `locally` f = wrap <$> run (f unwrap)

"CHOREORULES locallyseller"
    forall (f :: (forall a . a @ "seller" -> a) -> m a)
           (seller :: Proxy "seller") .
           seller `locally` f = return Empty
#-}
```

The first rule says that if a call to `locally` is seen where the location is known to be `Proxy "buyer"`, it should be rewritten to retain the computation. The second rule says that if the location is known to be `Proxy "seller"`, it should be written to not retain the computation.

When we compile the program, GHC applies the rules until no more rules can be applied. Again, here we rely on our trusted core plugin to verify that partitioning occurred as we expected it to. It seeks occurrences of the symbols `locally`, `~>`, and `cond`.

Now that we have compiled the buyer endpoint, we need to rewrite our rules to encode the seller semantics, and then recompile our program. What a drag!

Template Haskell. Luckily for us, we can use *Template Haskell* [7] (TH) to lessen our burden. With TH we can generate code at compile time. Not only that, but we can also generate compiler pragmas, such as rewrite rules!

We have written TH code that generates all of this boilerplate code for us. A programmer only needs to specify the names of the endpoints, and their location (IP address and port number)

```
$(compileFor 0 [ ("buyer",  ("localhost", 4224))
               , ("seller", ("localhost", 2424)) ]
```

This will generate first the proxies

```
buyer :: Proxy "buyer"
buyer = Proxy
seller :: Proxy "seller"
seller = Proxy
```

and then *all* the rewrite rules that are required to partition the program. The rules will be generated based on the first parameter to `compileFor`, which is an index indicating which of the endpoints in the list should be projected by the rules. To compile and run the bookseller example it is enough to compile the program twice, only changing the `0` to a `1` between the compilations.

6 Location Polymorphic Functions

Great, it seems as if we can also partition HasChor programs using rewrite rules! However, so far we have only tested our method on one example. Let us write another one, using more features from the type system, such as polymorphism.

```
node :: Proxy "node"
node = Proxy

server :: Proxy "server"
server = Proxy

addNumber :: KnownSymbol l => Proxy l -> Int @ l
          -> IORef Int @ "server"
          -> Choreo IO (() @ "server")
addNumber l vAtl ref = do
  vAtServer <- (l, vAtl) ~> server
  server `locally` \uw ->
    modifyIORef (uw ref) (\a -> a + uw vAtServer)

choreography :: Choreo IO (() @ "server")
choreography = do
  n1v <- node `locally` \_ -> return 50
  s1v <- server `locally` \_ -> return 100
  sref <- server `locally` \_ -> newIORef 0
  addNumber node n1v sref
  addNumber server s1v sref
  server `locally` \uw -> do
    v <- readIORef (uw sref)
    putStrLn $ show v
```

This program has two endpoints – `node` and `server`. The function `addNumber` takes a number at any location, and sends it to the server before it is added to the contents of a mutable reference. The function is *location polymorphic*, as it does not explicitly name which endpoint the supplied number resides at. The endpoint is identified via a `KnownSymbol` constraint, rather than via an explicit proxy type.

In the choreography, both `node` and `server` invoke this function, contributing an integer to the final content of the mutable reference.

When we compile this program we are at the mercy of GHC. Partitioning might work, but it might also not. The problem is that *no rule can be applied*. The rules require certain type information to be available to apply a rule.

Consider the rules for `~>` when we compile the `node` endpoint

```
{-# RULES
"CHOREORULES sendfromnode"
    forall v
           r
           (node :: Proxy "node") .
           (node, v) ~> r
                = do (send (unwrap v) (toLocTm r))
                     return Empty

"CHOREORULES sendtonode"
    forall s
           v
           (node :: Proxy "node") .
           (s, v) ~> node = wrap <$> recv (toLocTm s)
#-}
```

For calls to `~>` to be rewritten, the compiler must be able to see that the sender or receiver is `node`. Looking at the call to `~>` in `addNumber`, the type of the sending location is actually `KnownSymbol l => Proxy l`, which we have no rule for! Should we rewrite `~>` to a `send` or a `receive`? If nothing more is done, partitioning will fail. Our plugin verifies that partitioning indeed did not happen as we hoped it would

```
*** Partitioning verification pass running on Main
<no location info>: error:
    Found indications of partitioning not occurring when
    inspecting module: Main

    The problematic symbol that was encountered: ~>
```

However, if GHC chooses to *inline* the body of `addNumber`, it will in many cases be able to infer more type information and apply the appropriate rule.

We can make GHC very eager to inline functions by marking them as inlineable with a pragma, `{-# INLINE addNumber #-}`. However, GHC may *still* choose to not inline a function, so this is not a general solution. Furthermore, some functions will never be inlined. Consider the definition of `map` below

```
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

if we start inlining `map`, we will never stop! It is a recursive function, and recursive functions are never inlineable. What can we do now? Are we beyond saving? We need a way of introducing the missing information, enabling more rules to fire. Luckily for us, we can introduce the missing information with *specialisation*.

Specialisation in GHC. Specialisation in GHC is a feature that lets a programmer tell GHC that it should create specialised versions of certain functions, to reduce indirections. Consider the code below

```
numOp :: Num a => a -> a -> a
numOp x y = x + y * x

main :: IO ()
main = do
  print $ numOp (5 :: Int) 10
  print $ numOp (5.0 :: Double) 10.0
```

The function is overloaded as it works for any type that is a member of the `Num` type class. We invoke it twice at different types, `Int` and `Double`. This kind of polymorphism is implemented by passing in a dictionary at runtime, from which the overloaded operations `(+)` and `(*)` can be fetched. In pseudo-Haskell

```
data Num a = Num
  { add :: a -> a -> a
  , mul :: a -> a -> a
  }

numInt :: Num Int
numInt = Num primIntAdd primIntMul

numDouble :: Num Double
numDouble = Num primDoubleAdd primDoubleMul

numOp :: Num a -> a -> a -> a
numOp dict x y = add dict x (mul dict y x)

main :: IO ()
main = do
  print $ numOp numInt (5 :: Int) 10
  print $ numOp numDouble (5.0 :: Double) 10.0
```

While this is a convenient way of implementing overloading, there is a price to pay at runtime. We have to pass in an extra parameter, and the type-specific functions must be fetched from the dictionary before they can be applied.

If we know that we have a lot of invocations of `numOp` at the type `Int`, we can tell GHC to *specialise* `numOp` at that type. We do this by writing

```
{-# SPECIALISE numOp :: Int -> Int -> Int #-}
```

after which GHC will do two things. First, it will create a copy of `numOp` that is specialised to the type `Int`, and the indirections are removed

```
numOpInt :: Int -> Int -> Int
numOpInt x y = primIntAdd x (primIntMul y x)
```

Second, a rewrite rule is emitted that rewrites calls to `numOp` to `numOpInt`, if the argument type is `Int`.

```
{-# RULES "numOp/Int" numOp = numOpInt #-}
```

The cost of specialisation is that we duplicate code, so the compiled binaries might be larger.

Generalised specialisation. A recent feature of GHC is a generalisation of the existing specialisation mechanism[3]. What specialisation is really doing is fixing one of the type parameters of a function. The new specialisation allows a programmer to fix arbitrary expressions. How does this work? Consider the code below

```
process :: Bool -> Input -> IO Output
process debug input = do
  res <- computeSomething input
  -- imagine there are many debug calls like these
  if debug then putStrLn res else return ()
  computeSomethingElse res
```

Imagine that we have a function that processes some input. The function receives a boolean argument deciding whether debug information should be printed as the input is processed. If we know that many call sites where `process` is applied to the literal boolean `True`. We can say

```
{-# SPECIALISE forall input . process True input #-}
```

Which, again, will do two things. Firstly, a copy of `process` is created where the parameter debug is moved into the right-hand side, and fixed to the value `True`. We can illustrate this by `let`-binding the parameter

```
processTrue :: Input -> IO Output
processTrue input =
  let debug = True in do
    res <- computeSomething input
    -- imagine there are many debug calls like these
    if debug then putStrLn res else return ()
    computeSomethingElse res
```

Secondly, a rewrite rule is emitted to rewrite call sites where the first parameter is `True` to instead call the specialised version.

```
{-# RULES "process/True" forall input .
                process True input = processTrue input #-}
```

Let us consider what happens to a recursive function when we specialise it to concrete terms

```
data Event

renderVerbose :: Event -> String
renderConcise :: Event -> String

render :: Bool -> MVar Event -> IO ()
render b mv = do
  v <- takeMVar mv
```

[3]This feature is implemented but not yet merged into the main branch of GHC. The accepted proposal can be found here: https://github.com/ghc-proposals/ghc-proposals/blob/master/proposals/0493-specialise-expressions.rst

```
  if b
    then putStrLn $ renderVerbose v
    else putStrLn $ renderConcise v
  render b mv
```

`render` will wait until an event is written to an `MVar`. An `MVar` is a mutable reference that can either be empty or hold a value. If a thread tries to read from an empty `MVar`, it is blocked until another thread writes a value to it. Once the thread wakes up an event is ready to be rendered. It is rendered either in great or little detail, after which we recurse. If we would specialise `render` to `True`, the body of the generated function is

```
renderTrue :: MVar Event -> IO ()
renderTrue mv =
  let b = True in do
  v <- takeMVar mv
  if b
    then putStrLn $ renderVerbose v
    else putStrLn $ renderConcise v
  render b mv
```

The function is no longer recursive! However, as the optimiser tackles the right-hand side of `renderTrue`, the generated rewrite rule is going to be applied to the call to `render`, and the new function will be

```
renderTrue :: MVar Event -> IO ()
renderTrue mv =
  let b = True in do
  v <- takeMVar mv
  if b
    then putStrLn $ renderVerbose v
    else putStrLn $ renderConcise v
  renderTrue mv
```

`renderTrue` is now once again a recursive function, like its ancestor `render`. After some further optimisation, the boolean is folded into the test and the test is removed altogether, yielding

```
renderTrue :: MVar Event -> IO ()
renderTrue mv = do
  v <- takeMVar mv
  putStrLn $ renderVerbose v
  renderTrue mv
```

This new generalised specialisation feature, together with the optimisations that GHC does, is something of a *poor man's partial evaluation.*

The older version of specialisation is used to fix types, whereas the new version is used to fix terms. Now, let us see how we can use specialisation to partition HasChor programs!

7 HasChor and Rewrite Rules + Specialisation

We remind ourselves of the code in section 6. We had no rewrite rule to apply when inspecting the call to `~>` in `addNumber`, as not enough type information was available. However, we see that we apply `addNumber` to `node` and `server`. Our key insight is that we can use specialisation to introduce the missing information, enabling more rewrite rules to be applied. The following text only illustrates the effects of specialising the `node` endpoint, but we must duplicate these specialisation pragmas for all possible endpoints the function is invoked at. While this is a bit tedious, we can generate the pragmas using Template Haskell.

If we use the older version of specialisation, we can *specialise* `addNumber` to the types we know we need it at.

```
{-# SPECIALISE addNumber :: Proxy "node" -> Int @ "node"
                         -> IORef Int @ "server"
                         -> Chore IO (() @ "server") #-}
```

This will generate the specialised function

```
addNumberNode :: Proxy "node" -> Int @ "node"
              -> IORef Int @ "server"
              -> Choreo IO (() @ "server")
addNumberNode l vAtl ref = do
  vAtServer <- (l, vAtl) ~> server
  server `locally` \uw ->
    modifyIORef (uw ref) (\a -> a + uw vAtServer)
```

where there are no longer any type ambiguities. Additionally, we can also choose to specialise the program using the new, generalised specialisation. We achieve the same effect by specialising `addNumber` to the term `node` that is generated by our rewrite rules. This term has the necessary type information for our rules to be applied. The specialised function would be

```
-- notice that `node` is not bound by the forall, and
-- refers to the node that is in scope already
{-# SPECIALISE forall v sref . addNumber node v sref #-}

addNumberNode :: Int @ "node"
              -> IORef Int @ "server"
              -> Choreo IO (() @ "server")
addNumberNode vAtl ref =
  let l = node in do
    vAtServer <- (l, vAtl) ~> server
    server `locally` \uw ->
      modifyIORef (uw ref) (\a -> a + uw vAtServer)
```

In this version the type of `l` is known to be `Proxy "node"`, enabling our rewrite rules to fire. It might be slightly more efficient as well, as we have to pass in one argument less. In both cases, a specialised function and a rewrite rule is emitted, which will replace our call to `addNumber` with a call to `addNumberNode` instead.

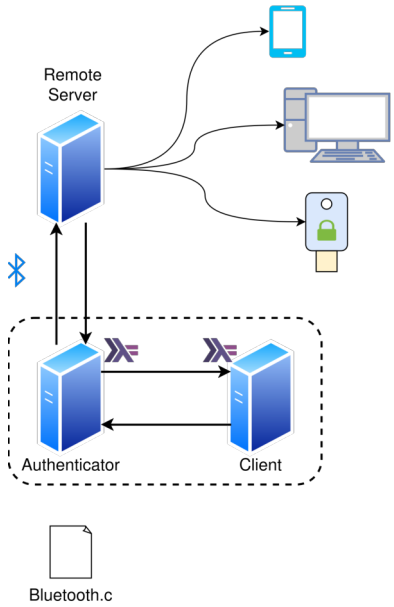

Figure 2. The 3-factor authentication setup. The authenticator and client are endpoints in a HasChor program, whereas the remote server is a completely disjoint program written in C. The remote server can present an authentication token to the client in different ways – by sending an SMS, by sending an email, interacting with a hardware key, or as in this case, printing to the terminal. The border surrounding the authenticator and client is intended to emphasise that these two endpoints are written together in a choreographic Haskell program. They can still execute on different physical machines. The code for `bluetooth.c` can be found in the appendix, as well as the complete C code for the remote server.

It might be a bit bothersome to write all these specialisation pragmas, but they can be generated using TH, enumerating all combinations of endpoints that a function might be applied at[4].

Specialisation, together with the rewrite rules described in section 5, lets us partition HasChor programs, and our plugin verifies that partitioning happened as it should.

8 Example - Third-Party Authentication

As an example of a scenario where partitioning makes building and deploying an application easier, we will implement a choreographic program that uses Bluetooth. The application as a whole implements a 3-factor authentication service, as depicted in figure 2.

The application is intended to authenticate a user against a remote server. The result of the interaction is an access token, granting access to some resource that is not shown here. The client invokes the authenticator, which will establish a bluetooth connection to a remote server. When a connection is established the remote server will present the user with a secret token somehow, e.g. via email or SMS. In this case, the token is printed to a terminal. The client sends the token to the authenticator, which relays it to the remote server. The remote server checks that the token is the same, after which it returns an access token.

The authenticator is going to need to make some foreign calls to C code[5] to establish a Bluetooth connection to the remote server. We will also declare the proxies that denote two endpoints – the authenticator and a client.

```
foreign import ccall "allocate_socket_"
    allocateSocket :: IO Int
foreign import ccall "connect_"
    connect :: Int -> IO ()
foreign import ccall "authenticate_"
    authenticate :: Int -> Word32 -> IO Word64
foreign import ccall "close_"
    close :: Int -> IO ()

authenticator :: Proxy "authenticator"
authenticator = Proxy

client :: Proxy "client"
client = Proxy
```

The choreography may include many endpoints which all might want to receive an access token, so we will make the authentication function location polymorphic

```
getAccessToken :: KnownSymbol l
               => Proxy l -> Choreo IO (Maybe Word64 @ l)
```

The authenticator will begin by allocating a socket and connecting to it. The `connect` FFI call knows the Bluetooth address of the remote server and tries to establish a connection.

```
getAccessToken p = do
    authSocket <- authenticator `locally` \_ -> do
        socket <- allocateSocket
        -- upon connection, the remote server is going to
        -- present the user with a secret.
        connect socket
        return socket
```

After successfully connecting to the remote server, the client will be presented with a token somehow. The client will enter this token, after which it will be sent to the authenticator

```
    secret <- p `locally` \_ -> do
        putStr "enter the secret>"
```

[4]The authors have written the code that does everything up until the last stage, actually emitting the pragmas. At the time of writing this pearl, TH does not yet support outputting the new syntax of specialisation pragmas.

[5]The C code that the Haskell code calls use `dlopen` to access the BlueZ Bluetooth stack, in order to avoid declaring this dependency in the Cabal file. The code for this example can be found here: https://github.com/Rewbert/HasChor/tree/rwr/examples/pearlexample

```
3         hFlush stdout
4         getLine
5
6     authSecret <- (p, secret) ~> authenticator
```

The authenticator will call the `authenticate` FFI call with the token, after which the server will reply with either a zero, indicating the token was wrong, or with a 64-bit value, representing the access token. The authenticator will close the socket, severing the connection to the remote server. Finally, the token is sent back to the client, who can now use it to access some guarded resource

```
1     authRes <- authenticator `locally` \uw -> do
2         r <- authenticate
3                 (uw authSocket)
4                 (read (uw authSecret) :: Word32)
5         close (uw authSocket)
6         if r == 0
7           then return Nothing
8           else return (Just r)
9
10    (authenticator, authRes) ~> p
```

Finally, we need to specialise the whole function to the endpoints we will invoke it at

```
1 {-# SPECIALISE getAccessToken client #-}
2 -- potentially more
```

Now, the C code that is called via FFI functions depends on the *BlueZ* Bluetooth stack for the Linux kernel. If the platform for which we are compiling the application does not have access to the `bluetooth.c` file that we've written, or the BlueZ library, compilation is going to fail in the linking stage. With standard HasChor, where no partitioning occurs, we always need to have these resources at hand when we are compiling the program. This is because we do not know until runtime if we will need it or not (when we choose which endpoint to project). The code in `bluetooth.c` uses the `dlopen` function from the C standard library to use the BlueZ Bluetooth stack at runtime.

With partitioning, however, there is a different story. We point out that all interactions with the FFI functions happen on the authenticator endpoint. When we compile the authenticator, we need to provide the `bluetooth.c` source. In contrast, when we compile the client endpoint, *after partitioning there are no calls to the FFI functions left.* We do not need to provide `bluetooth.c` when we compile our code, because it is not needed. The client machine does not need to unnecessarily install the Bluetooth libraries! Not only does this make building simpler, but we also don't need to rely on thousands of lines of code in vain.

This, of course, goes beyond eliminating simple C files. If a package is used only at a single endpoint, all usages of the package occur in calls to `locally`. Once these are partitioned away, GHC will choose to not link in the code for that package during compilation. This has the potential to severely reduce the trusted computing base of an endpoint.

9 When does it Not Work

Firstly, when we want to partition a program to make sure that certain code is not linked in, we need to be very careful with what we export. If a module lacks an export list, everything in that module is going to be exported. In our Bluetooth example, we export only `main`, and that means that we are not exporting the FFI functions. If we did not have this explicit export list, all the FFI functions would be exported as well, and the code would be required during linking.

It might at first glance seem a bit arbitrary which form of specialisation we use to expose more type information for HasChor, and this is for the most part true. When we experimented with the examples that are shipped with HasChor, we did find a difference, however. Consider the following type

```
1 sort ::
2   KnownSymbol a => Proxy a ->
3   KnownSymbol b => Proxy b ->
4   KnownSymbol c => Proxy c ->
5   ([Int] @ a) ->
6   Choreo IO ([Int] @ a)
```

It is a bit non-idiomatic to not collect all constraints in the beginning of the type, but legal nonetheless. If we naively try to specialise this as such

```
1 {-# SPECIALISE sort :: Proxy "primary"
2                     -> Proxy "worker1"
3                     -> Proxy "worker2"
4                     -> ([Int] @ "primary")
5                     -> Choreo IO ([Int] @ "primary") #-}
```

we get a type error, whereas if we naively specialise it to the concrete terms, everything goes well

```
1 {-# SPECIALISE forall . sort primary worker1 worker2 #-}
```

We can of course also just refactor the original type to specify all the constraints in the beginning. This might be the most ideal, as it is more idiomatic.

Finally, GHCs separate compilation may complicate things. If a function and its specialisations are defined in one module, but used in another, all variants of the function must be exported. Dead code elimination will in this case not be able to remove the original function, and it will remain even after linking. Care must be taken with when and where some functions are defined.

10 Conclusions & Future Work

In conclusion, perhaps unsurprisingly so, we can use rewrite rules to partition Haskell programs. Traditional short-cut fusion aims to eliminate intermediate data structures, and

that is precisely what we are trying to do in the case of HasChor. We eliminate the intermediate `ChoreoSig` structure that represents the whole program and use rewrite rules to immediately project a `NetworkSig` structure, modeling just one endpoint, and our plugin helps us verify that partitioning occurred.

We have shown by example that in some situations, partitioning a program can remove dependencies, yielding a program that is easier to build and deploy.

One effect of partitioning programs is that we are removing code, yielding smaller binaries. The binaries produced by GHC are already very large, so this effect is sometimes hardly noticeable. However, the new MicroHaskell compiler has a runtime small enough that it can execute Haskell code on microcontrollers. In this setting, we must make our binaries as small as possible, as memory can be a scarce resource. Today, MicroHaskell does not have enough functionality to partition programs the way that we describe it in this pearl, but the authors are discussing possible ways of making it happen with the MicroHaskell developers.

Acknowledgments

The authors would like to thank Simon Peyton Jones for taking the time to discuss some details of GHC with us. Further thanks go to Koen Claessen and John Hughes for helpful discussions and feedback.

A HasChor Bookseller

Below follow the implementation of the bookseller protocol using HasChor, as taken from the HasChor repository of examples[6]

```
buyer :: Proxy "buyer"
buyer = Proxy

seller :: Proxy "seller"
seller = Proxy

budget :: Int
budget = 100

priceOf :: String -> Int
priceOf "Types and Programming Languages" = 80
priceOf "Homotopy Type Theory"            = 120

deliveryDateOf :: String -> Day
deliveryDateOf "Types and Programming Languages" =
  fromGregorian 2022 12 19
deliveryDateOf "Homotopy Type Theory" =
  fromGregorian 2023 01 01

bookseller :: Choreo IO (Maybe Day @ "buyer")
bookseller = do
  title <- buyer `locally` \_ -> do
    putStrLn "Enter the title of the book to buy"
    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
      deliveryDate <- seller `locally` \un ->
        return $ deliveryDateOf (un title')
      deliveryDate' <- (seller, deliveryDate) ~> buyer

      buyer `locally` \un -> do
        putStrLn $ "The book will be delivered on "
                   ++ show (un deliveryDate')
        return $ Just (un deliveryDate')

    False -> do
      buyer `locally` \_ -> do
        putStrLn "Too expensive"
        return Nothing
```

[6] https://github.com/gshen42/HasChor/blob/main/examples/bookseller-1-simple/Main.hs

B Bookseller Rewrite Rules

Below follows the rewrite rules that implement the semantics for the buyer endpoint in the bookseller choreography.

```
{-# NOINLINE locally #-}
{-# NOINLINE (~>) #-}
{-# NOINLINE cond #-}
{-# RULES
"CHOREORULES locallybuyer"
    forall (f :: (forall a. At a "buyer" -> a) -> m a)
           (buyer :: Proxy "buyer").
           buyer `locally` f = wrap <$> run (f unwrap)

"CHOREORULES locallyseller"
    forall (f :: (forall a. At a "seller" -> a) -> m a)
           (seller :: Proxy "seller").
           seller `locally` f = return Empty

"CHOREORULES sendfrombuyer"
    forall v
           r
           (buyer :: Proxy "buyer").
           (buyer, v) ~> r
               = do (send (unwrap v) (toLocTm r))
                    return Empty
```

```
"CHOREORULES sendtobuyer"
    forall s
           v
           (buyer :: Proxy "buyer") .
           (s, v) ~> buyer = wrap <$> recv (toLocTm s)

"CHOREORULES condbuyer"
    forall v
           c
           (buyer :: Proxy "buyer") .
           (buyer, v) `cond` c = do broadcast (unwrap v)
                                    c (unwrap v)

"CHOREORULES condseller"
    forall v
           c
           ( seller :: Proxy "seller") .
           ( seller , v) `cond` c =
               do x <- recv (toLocTm seller )
                  c x
#-}
```

References

[1] Anton Ekblad and Koen Claessen. 2014. A seamless, client-centric programming model for type safe web applications. In *Proceedings of the 2014 ACM SIGPLAN symposium on Haskell, Gothenburg, Sweden, September 4-5, 2014*, Wouter Swierstra (Ed.). ACM, New York, NY, USA, 79–89. https://doi.org/10.1145/2633357.2633367

[2] Adrien Ghosn, James R. Larus, and Edouard Bugnion. 2019. Secured Routines: Language-based Construction of Trusted Execution Environments. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*, Dahlia Malkhi and Dan Tsafrir (Eds.). USENIX Association, 571–586. https://www.usenix.org/conference/atc19/presentation/ghosn

[3] Andrew John Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Proceedings of the conference on Functional programming languages and computer architecture, FPCA 1993, Copenhagen, Denmark, June 9-11, 1993*, John Williams (Ed.). ACM, 223–232. https://doi.org/10.1145/165180.165214

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

[5] Aditya Oak, Amir M. Ahmadian, Musard Balliu, and Guido Salvaneschi. 2021. Language Support for Secure Software Development with Enclaves. In *34th IEEE Computer Security Foundations Symposium, CSF 2021, Dubrovnik, Croatia, June 21-25, 2021*. IEEE, 1–16. https://doi.org/10.1109/CSF51468.2021.00037

[6] Abhiroop Sarkar, Robert Krook, Alejandro Russo, and Koen Claessen. 2023. HasTEE: Programming Trusted Execution Environments with Haskell. In *Proceedings of the 16th ACM SIGPLAN International Haskell Symposium, Haskell 2023, Seattle, WA, USA, September 8-9, 2023*, Trevor L. McDonell and Niki Vazou (Eds.). ACM, New York, NY, USA, 72–88. https://doi.org/10.1145/3609026.3609731

[7] Tim Sheard and Simon Peyton Jones. 2002. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell, Haskell 2002, Pittsburgh, Pennsylvania, USA, October 3, 2002*, Manuel M. T. Chakravarty (Ed.). ACM, 1–16. https://doi.org/10.1145/581690.581691

[8] Gan Shen, Shun Kashiwa, and Lindsey Kuper. 2023. HasChor: Functional Choreographic Programming for All (Functional Pearl). *Proc. ACM Program. Lang.* 7, ICFP (2023), 541–565. https://doi.org/10.1145/3607849

Received 2024-06-03; accepted 2024-07-05