The Remote Monad Design Pattern

Andy Gill Neil Sculthorpe [*] Justin Dawson Aleksander Eskilson
Andrew Farmer Mark Grebe Jeffrey Rosenbluth [†] Ryan Scott James Stanton

Information and Telecommunication Technology Center, University of Kansas, USA
first.last@ittc.ku.edu

Abstract

Remote Procedure Calls are expensive. This paper demonstrates how to reduce the cost of calling remote procedures from Haskell by using the *remote monad design pattern*, which amortizes the cost of remote calls. This gives the Haskell community access to remote capabilities that are not directly supported, at a surprisingly inexpensive cost.

We explore the remote monad design pattern through six models of remote execution patterns, using a simulated Internet of Things toaster as a running example. We consider the expressiveness and optimizations enabled by each remote execution model, and assess the feasibility of our approach. We then present a full-scale case study: a Haskell library that provides a Foreign Function Interface to the JavaScript Canvas API. Finally, we discuss existing instances of the remote monad design pattern found in Haskell libraries.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Language Classifications—Applicative (functional) languages

Keywords Monads, Remote Procedure Call, FFI, Design Pattern.

1. Introduction

Remote Procedure Calls (RPCs) are expensive. This paper presents a way to make them considerably cheaper: the *remote monad design pattern*. Monads [34, 42] provide a general way of structuring composite computations. These monadic computations are first class: they can be passed to functions as arguments, returned from functions as results, and stored in data structures. But monadic computations are not typically executed *remotely*.

This paper investigates the ways that monadic computations can be serialized for the purposes of being sent to remote locations for external execution. The idea is that, rather than directly call a remote procedure, we instead give the remote procedure call a service-specific monadic type, and invoke the remote procedure call using a monadic "send" function.

[*] Now at the Department of Computer Science, Swansea University, UK, n.a.sculthorpe@swansea.ac.uk

[†] Unaffiliated

Haskell'15, September 3-4, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3808-0/15/09...
http://dx.doi.org/10.1145/2804302.2804311

Definition. *A **remote monad** is a monad that has its evaluation function in a remote location, outside the local runtime system.*

By factoring the RPC into sending invocation and service name, we can group together procedure calls, and amortize the cost of the remote call. To give an example, Blank Canvas, our library for remotely accessing the JavaScript HTML5 Canvas, has a `send` function, `lineWidth` and `strokeStyle` services, and our remote monad is called `Canvas`:

```
send        :: Device -> Canvas a -> IO a
lineWidth   :: Double              -> Canvas ()
strokeStyle :: Text                -> Canvas ()
```

If we wanted to change the (remote) line width, the `lineWidth` RPC can be invoked by combining `send` and `lineWidth`:

```
send device (lineWidth 10)
```

Likewise, if we wanted to change the (remote) stroke color, the `strokeStyle` RPC can be invoked by combining `send` and `strokeStyle`:

```
send device (strokeStyle "red")
```

The key idea of this paper is that *remote* monadic commands can be *locally* combined before sending them to a remote server. For example:

```
send device (lineWidth 10 >> strokeStyle "red")
```

The complication is that, in general, monadic commands can return a result, which may be used by subsequent commands. For example, if we add a monadic command that returns a Boolean,

```
isPointInPath :: (Double,Double) -> Canvas Bool
```

we could use the result as follows:

```
send device $ do
   inside <- isPointInPath (0,0)
   lineWidth (if inside then 10 else 2)
   ...
```

The invocation of `send` can also return a value:

```
do res <- send device (isPointInPath (0,0))
   ...
```

Thus, while the monadic commands inside `send` are executed in a *remote* location, the results of those executions need to be made available for use *locally*.

2. Haskell Calling the Real World

When Haskell users want an established graphics library, such as OpenGL, or a web-client library, such as cURL, they typically use a Haskell library that uses the Foreign Function Interface (FFI) capability to link Haskell code with the native capabilities of the desired external library. Rather than reimplement existing code, the rich FFI capability is used to tunnel to C, and onwards to established libraries, such as the C or C++ implementations of OpenGL and cURL. The FFI is well-supported, but there are three conceptual problems to be solved in crossing to native C libraries:

1. Control flow needs to transfer to the correct C function. Given the lowest level of the GHC runtime system is written in C, control-flow transfer is straightforward.
2. The data structures that are arguments and results of calls to C need to be marshalled into the correct format. For example, C strings are not the same as Haskell strings.
3. The abstractions of the target C library may not be idiomatic Haskell abstractions. For example, many C++ APIs assume OO-style class inheritance. Foreign abstractions can be simulated in Haskell, but this raises an obfuscation barrier.

Any time control flow leaves the Haskell eco-system, all three of these concerns need to be addressed. All three are supported by the Haskell FFI for C: There is a way of directly promoting a C function into Haskell-land; there is good support for marshalling Haskell data structures into C structures, as well as automatic memory-management support; and Haskell abstraction capabilities are used to build more Haskell-centric APIs on top of the FFI capability. However, what about functions that cannot be called directly, but must be invoked using a remote procedure call? All three identified FFI issues come into play:

1. The control flow needs to use an RPC mechanism to establish foreign control-flow transfer. This can add considerable costs. A typical mechanism would be to establish a TCP/IP connection with a published remote service.
2. The procedure arguments are sent to the remote location, typically over a TCP/IP channel, the remote command is executed, and the result is communicated back.
3. The remote nature of the call raises issues with presenting remoteness to the API user. What does this remote function look like to a Haskell programmer? Are the costs reflected in the API design?

In this paper we investigate a generalization of the remote procedure call, using monads as a descriptor of remote execution. Specifically, we make the following contributions:

- We document a DSL pattern for bundling RPCs: the remote monad design pattern. This pattern cuts the cost of RPCs in many cases by amortizing the remote aspect, and is a design point between classical monadic APIs, and using deeply embedded DSLs for offline remote execution.
- Toward understanding the remote monad, we give four complete models of remote execution (§3.1, §3.2, §4, §5) and sketch two useful variations of the strongest model (§6, §7). We give a set of remote monad laws (§8), and observe that the pattern has been used in a weak form several times before (§11).
- We explore the design pattern in a real-world large-scale example, called Blank Canvas (§9). We quantify our experiences, and measure performance on some benchmarks, comparing native JavaScript and Blank Canvas, over two operating systems, and two web browsers (§10).

3. Modeling Remote Communication

For illustration purposes, we will remotely control an Internet of Things toaster, which includes a thermometer and a voice synthesis circuit. We can ask the toaster to `say` things, measure the `temperature`, or make `toast` for a given number of seconds. Using a monad called `Remote`, we have:

```
say         :: String -> Remote ()
temperature ::           Remote Int
toast       :: Int    -> Remote ()
```

In the remote monad design pattern, we have a way of running our monad remotely, which we call `send`:

```
send :: Device -> Remote a -> IO a
```

The key step in an efficient implementation of the remote monad is the choice of packaging of commands. Ideally, the entire monadic computation argument of `send` would be transmitted in a single packet; but in general this is not possible.

Towards understanding the choices, we are going to build several ways of interacting with our toaster:

- As a preamble, in §3.1 we build a basic *asynchronous* RPC, and in §3.2 we build a basic *synchronous* RPC.
- In §4 we build the simple version of a remote monad, called a *weak remote monad*, where a monadic API is utilized to *conceptually* bundle commands for remote execution.
- In §5 we build another remote monad, called a *strong remote monad*, where a monadic API is utilized to *actually* bundle commands for more efficient remote execution. *This is our principal model of the key idea of this paper.*
- In §6 we build a *remote applicative functor*, which exploits the restrictions of the applicative functor operations (relative to the more expressive monadic bind) to allow for better bundling.
- Finally, in §7 we use a deep-embedding technique to add support for remote binding of non-local values, again improving the bundling of remote primitives.

In each case, we implement the local behavior of the `send` function, as well as give a minimal implementation of remote behavior. In general, in the remote monad design pattern, the remote execution may be in any language. Here, for simplicity, we simulate remote execution from within Haskell, and use the built-in `Show` and `Read` classes to simulate serialization and deserialization. We highlight the dual execution nature of the remote monad by using distinct data structures locally and remotely. Using distinct data structures also allows us to side-step the orthogonal challenge of GADT deserialization.

3.1 An Asynchronous Remote Command Call

This first model will be able to send single commands asynchronously to a remote device. This model is also known as the **Command** design pattern [19].

Definition. *A remote* **command** *is a request to perform an action for remote effect, where there is no result value or temporal consequence.*

We use a deep embedding of the commands that we want to `send` asynchronously. For this simple example, we only consider a single command:

```
data Command = Say String
               deriving Show
```

We derive a `Show` instance so that we can serialize `Command`, in preparation for transmitting it to a remote interpreter.

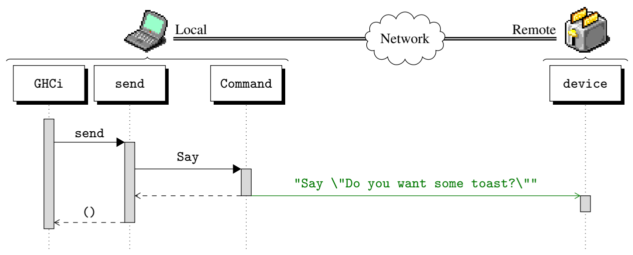


Figure 1: Example of an Asynchronous Remote Command Call

We represent a remote device as a function from `String` to `IO ()`, wrapped in a data structure called `Device`, which models the communication channel between local and remote execution.

```
data Device = Device { async :: String -> IO () }
```

We call the internal function `async` because it represents asynchronous communication. For asynchronous commands, there is no need for a back-channel on which to return values.

We can now define `send`, which serializes a `Command` and then transmits it to a remote device:

```
send :: Device -> Command -> IO ()
send d m = async d (show m)
```

Our Haskell simulation of the remote device requires a representation of commands on the remote device, an execution function for those commands, and a deserialization function that reads those commands. As a convenience, we use the same constructor name for the remote command data type, so that deserialization can be handled by a derived `Read` instance.[1]

```
data RCommand = Say String
                deriving Read

execRCommand :: RCommand -> IO ()
execRCommand (Say str) = putStrLn ("Remote: " ++ str)
```

Our `device`, which simulates the remote interpreter handle, can then be defined as:

```
device :: Device
device = Device (execRCommand . read)
```

This completes our model. Now we can test it by sending a `Say` command, which prints *remotely*:

```
GHCi> send device (Say "Do you want some toast?")
Remote: Do you want some toast?
```

In summary, this model is a direct implementation of *asynchronous* remote calls. Figure 1 shows the interactions in this example invocation using a sequence diagram. We take the liberty of expressing our construction of `Command` as a sequence process, and we give the serialized text sent to the remote device, using green for an asynchronous call.

3.2 A Synchronous Remote Call

In this subsection we build a *synchronous* remote call: a version of `send` that can receive a reply to a remotely transmitted procedure. That is, we will define a model of a remote procedure call.

Definition. *A remote* **procedure** *is a request to perform an action for its remote effects, where there is a result value or temporal consequence.*

[1] This code must be placed in a separate module to avoid a name clash.

When we execute a remote procedure, we either want to get a result back (e.g. the measured temperature), or know that a specific remote action has been completed (e.g. the toast is made). In this model, because we are interested in getting a reply, we represent remoteness using a function from `String` to `IO String`.

```
data Device = Device { sync :: String -> IO String }
```

As with commands, we use a deep embedding of the procedures that we want to `send`. However, as we now expect to receive a result in reply, we use a GADT with a phantom type index denoting the expected result type:

```
data Procedure :: * -> * where
  Temperature ::        Procedure Int
  Toast       :: Int -> Procedure ()
```

As with commands, we provide serialization using the `Show` class:

```
deriving instance Show (Procedure a)
```

For deserialization, we provide an auxiliary function that uses the phantom type index of `Procedure` to determine which `Read` instance should be used to parse the reply from the remote device:

```
readProcedureReply :: Procedure a -> String -> a
readProcedureReply (Temperature {}) i = read i
readProcedureReply (Toast {})       i = read i
```

The two `read` functions are each reading different types, as constrained by the specific `Procedure` constructor.

Now we can write our `send` command:

```
send :: Device -> Procedure a -> IO a
send d m = do
  r <- sync d (show m)
  return (readProcedureReply m r)
```

That is, `send` serializes the `Procedure`, sends it over a synchronous channel, and interprets the reply in terms of the type of the same `Procedure`. This time, `send` is polymorphic in the type parameter of `Procedure` and `IO`.

Our simulated remote `Device` is straightforward to construct. For conciseness we have merged serialization of the result value into the execution function, but these steps could be separated.

```
data RProcedure = Temperature
                | Toast Int
                  deriving Read

execRProcedure :: RProcedure -> IO String
execRProcedure Temperature  = do
  t <- randomRIO (50, 100 :: Int)
  return (show t)
execRProcedure (Toast n)    = do
  putStrLn ("Remote: Toasting...")
  threadDelay (1000 * 1000 * n)
  putStrLn ("Remote: Done!")
  return (show ())

device :: Device
device = Device (execRProcedure . read)
```

This completes our model. Now we can test it by sending a `Temperature` procedure, which returns the temperature *locally*:

```
GHCi> send device Temperature
56
```

We have taken a procedure, transmitted it to a remote interpreter, executed it remotely, and returned with the result — thus we have a basic model of a typical Remote Procedure Call. Figure 2 shows the sequence diagram for this interaction. We use red to highlight the synchronous communication call.

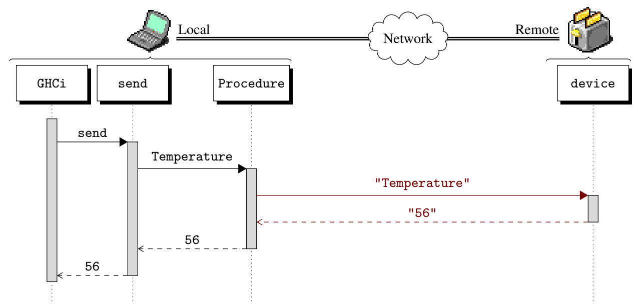


Figure 2: Example of a Synchronous Remote Procedure Call

4. The Weak Remote Monad

Thus far, our arguments to `send` have not been instances of `Monad`. We will now address this. Here we join the asynchronous and synchronous models into a monad called `Remote`. In this section, we implement an initial version of the remote monad that does not amortize the cost of communication. We call this a *weak* remote monad.

Definition. *A* **weak remote monad** *is a remote monad that sends each of its remote calls individually to a remote interpreter.*

In this example, the `Remote` monad is implemented using the reader monad, where the environment is our `Device`, nested around the `IO` monad, using monad transformers [22, 27].

```
newtype Remote a = Remote (ReaderT Device IO a)

deriving instance Monad Remote
```

This gives us access to the specific `Device`, which will allow us to send individual commands to the remote interpreter on the fly. We use `deriving instance Monad` to allow the monadic operators to be used, while hiding the transformer-based operations inside the `Remote` abstraction.

`Device` now needs to support *both* `sync` and `async`. In a real implementation, it would be expected that there would be two entry points into the same communication channel, where the usage specifics depend on if we want to receive a reply.

```
data Device = Device
  { sync  :: String -> IO String
  , async :: String -> IO ()
  }
```

A `send`-style function, customized for our weak `Remote` monad, provides remote execution for `Command`. Observe that each command invokes the remote procedure call immediately.

```
sendCommand :: Command -> Remote ()
sendCommand m = Remote $ do
  d <- ask
  liftIO (async d (show m))
  return ()

say :: String -> Remote ()
say txt = sendCommand (Say txt)
```

For procedures, we also provide a `send`-style function, again customized for our weak `Remote` monad. Again, each procedure invokes the remote procedure call immediately.

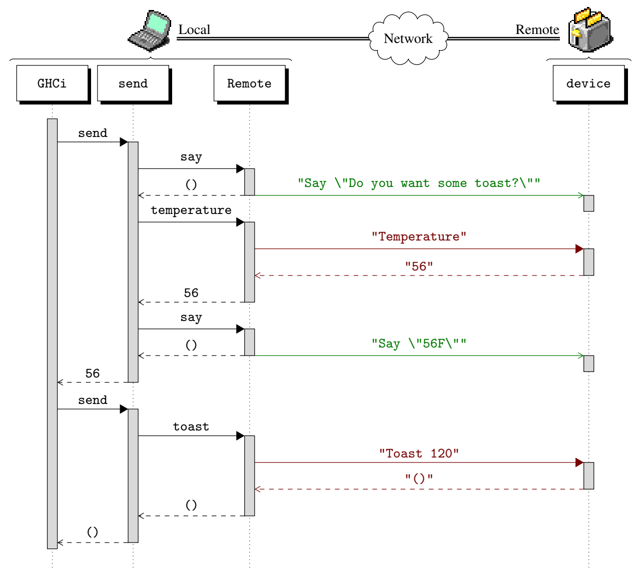


Figure 3: Example of a Weak Remote Monad

```
sendProcedure :: Procedure a -> Remote a
sendProcedure m = Remote $ do
  d <- ask
  r <- liftIO (sync d (show m))
  return (readProcedureReply m r)

temperature :: Remote Int
temperature = sendProcedure Temperature

toast :: Int -> Remote ()
toast n = sendProcedure (Toast n)
```

Our main `send` function is now used to run the `Remote` monad, unboxing the reader function, and applying it to the `Device`:

```
send :: Device -> Remote a -> IO a
send d (Remote m) = runReaderT m d
```

Finally, the virtual remote `Device` is a combination of the synchronous and asynchronous `Device`s:

```
device :: Device
device = Device (execRProcedure . read)
                (execRCommand   . read)
```

Now we can call `send` with a monadic argument, and chains of primitives, connected using the monad, will be executed. We have achieved our original goal: a (weak) remote monad where the primitive commands and procedures are executed in a remote location.

```
GHCi> t <- send device $ do
              say "Do you want some toast?"
              t <- temperature
              say (show t ++ "F")
              return t
Remote: Do you want some toast?
Remote: 56F
GHCi> when (t < 70) (send device (toast 120))
Remote: Toasting...
...sleeping for 120 seconds...
Remote: Done!
```

Figure 3 shows the sequence diagram for this example of the weak remote monad. For every primitive call invoked, there is a remote call to the `device`. Furthermore, the GHCi computation does not terminate until the toast is complete.

5. The Strong Remote Monad

We want to bundle monadic remote calls, and send them as packets of computations to be remotely executed. The strong remote monad does this. We have two classes of primitive remote calls: *commands* that do not require any specific result to be observed, and *procedures* that require a reply from the remote site.

- For commands, which are asynchronous and do not send a reply, we can queue up the command and commit to sending it later.
- For procedures, which are synchronous, we need to transmit them immediately. Thus we first send all outstanding queued commands, then send the procedure, and then await the procedure's result.

At the end of executing a `send`, we flush the queue of any outstanding commands.

This is the key idea behind a strong remote monad: package the sequence of monadic actions into a list of commands, which are (in all but the final case) terminated by procedures. This design assumes that the only primitive remote calls in our remote monad are commands and procedures, and thus there is no way of *locally* pausing or stalling the pipeline of commands. The queuing of commands is simply a bundling strategy for commands and procedures that would be executed immediately after each other anyway.

Definition. *A* **strong remote monad** *is a remote monad that bundles all of its remote calls into packets of commands, punctuated by procedures, for remote execution.*

In the strong remote monad, as well as knowing what `Device` to talk to, we need to queue up the list of to-be-transmitted `Commands`. We use a reader monad for the `Device` (as with the weak remote monad), and a state monad for the command queue. Thus, our (strong) remote monad can be defined as:

```
newtype Remote a =
  Remote (ReaderT Device (StateT [Command] IO) a)
```

We want to send a packet of `Commands`, terminated by a `Procedure`, and also to have a way of sending a packet of `Commands` without a terminating `Procedure`. For the latter case, we simply send `[Command]`. For the former case, we introduce a dedicated data type:

```
data Packet a = Packet [Command] (Procedure a)
                deriving Show
```

Now we can start providing monadic primitives. A utility function, `sendCommand`, appends `Commands` to our "to-be-sent" queue, and `say` supports the remote monad interface:

```
sendCommand :: Command -> Remote ()
sendCommand cmd = Remote (modify (++ [cmd]))

say :: String -> Remote ()
say txt = sendCommand (Say txt)
```

Supporting procedures is more involved. We need to flush the queue of outstanding `Commands`, as well as to actually send a packet containing the `Commands` and procedure to the remote site:

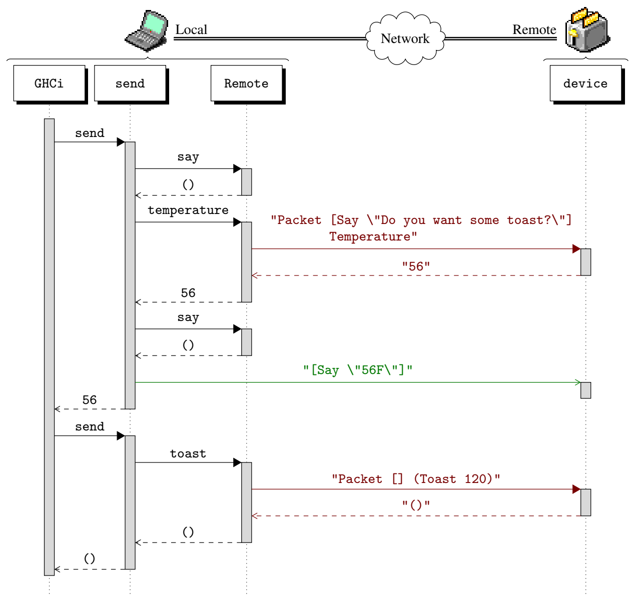


Figure 4: Example of a Strong Remote Monad

```
sendProcedure :: Procedure a -> Remote a
sendProcedure p = Remote $ do
   d <- ask
   cs <- get
   r <- liftIO (sync d (show (Packet cs p)))
   put []
   return (readProcedureReply p r)

temperature :: Remote Int
temperature = sendProcedure Temperature

toast :: Int -> Remote ()
toast n = sendProcedure (Toast n)
```

Next, we provide our `send` function, which runs our inner monad, and flushes any remaining `Commands` to the remote interpreter:

```
send :: Device -> Remote a -> IO a
send d (Remote m) = do
  (r,cs) <- runStateT (runReaderT m d) []
  when (not (null cs)) (async d (show cs))
  return r
```

Finally, we define our simulated remote device, now extended with an execution function for packets:

```
device :: Device
device = Device (execRPacket        . read)
                (mapM_ execRCommand . read)

data RPacket = Packet [RCommand] RProcedure
               deriving Read

execRPacket :: RPacket -> IO String
execRPacket (Packet cs p) = do
        mapM_ execRCommand cs
        execRProcedure p
```

Figure 4 shows the sequence diagram for the strong remote monad, on the same example as used for the weak remote monad. As can been seen, `Packet` combines `Commands`, punctuated by `Procedures`.

6. The Remote Applicative Functor

There is also a remote applicative functor.

Definition. *A* **remote applicative functor** *is an applicative functor that has its evaluation function in a remote location, outside the local runtime system.*

As with monads, there are two classes: the *weak* remote applicative functor, and the *strong* remote applicative functor. Without a bind operator, applicative functors [30] are fundamentally better suited to remoteness than monads are: subsequent applicative computations cannot depend on the results of prior computations, which in our context allows for bundling of procedures. In fact (using our terminology) a strong remote applicative functor is currently used by Facebook to bundle database requests [29].

Any weak remote monad is, by definition, (at least) a weak remote applicative functor, with primitive calls transmitting individually. More interesting is the *strong* remote applicative functor, which we consider here. Exploiting the independence of subsequent calls from the results of prior calls, we are going to bundle *all* the Commands and Procedures together in a single packet, which we represent by a list of Prims (primitive remote calls).

```
data Prim :: * where
  Command   ::           Command     -> Prim
  Procedure :: Show a => Procedure a -> Prim

deriving instance Show Prim
```

Our applicative functor is a wrapper around a monad transformer:

```
newtype Remote a =
  Remote (WriterT [Prim] (State [String]) a)
```

This monad transformer is a combination of the writer monad, for accumulating queued primitive calls, and the state monad, for simultaneously parsing results. We will use lazy evaluation to "tie-the-knot" [2] between these two effects in send. We explicitly provide the instances for Applicative and Functor, but not Monad.

We handle Commands using the underlying writer monad:

```
sendCommand :: Command -> Remote ()
sendCommand cmd = Remote (tell [Command cmd])
```

Handling Procedures is a bit more involved. We use the writer monad to remember the Procedure, then read the result from the state, lazily pulling it off a list. Crucially, there are not any external effects inside our inner monad.

```
sendProcedure :: Show a => Procedure a -> Remote a
sendProcedure p = Remote $ do
   tell [Procedure p]
   ~(r:rs) <- get
   put rs
   return (readProcedureReply p r)
```

We build send using recursive do notation [16, 17]. If our list of Prims contains only Commands, then we send it asynchronously to our remote device. Otherwise, we send it synchronously, and await the list of results from the remote device. We then recursively feed this result list back into the invocation of the applicative functor. Whether synchronous or asynchronous, we bundle the complete list of Prims as a single packet.

```
send :: Device -> Remote a -> IO a
send d (Remote m) = do
  rec let ((a,ps),_) = runState (runWriterT m) r
      r <- if all isCommand ps
             then do async d (show ps)
                     return []
             else do str <- sync d (show ps)
                     return (read str)
  return a
```

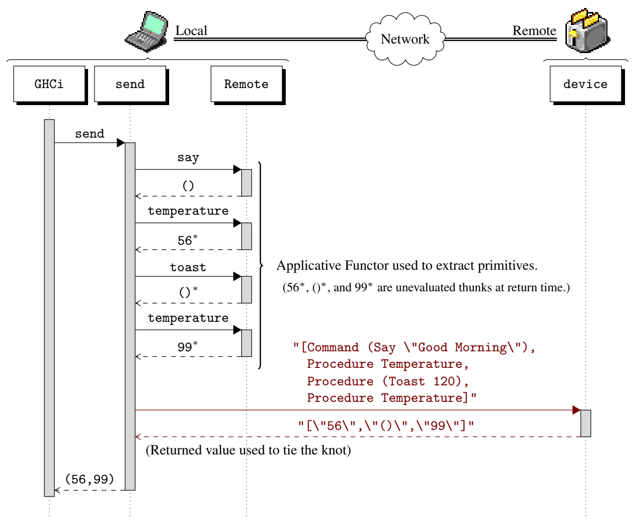

Local
Network
Remote
GHCi
send
Remote
device
send
say
()
temperature
56*
toast
()*
temperature
99*
Applicative Functor used to extract primitives.
(56*, ()*, and 99* are unevaluated thunks at return time.)
"[Command (Say \"Good Morning\"), Procedure Temperature, Procedure (Toast 120), Procedure Temperature]"
"[\"56\",\"()\",\"99\"]"
(Returned value used to tie the knot)
(56,99)

Figure 5: Example of a Strong Remote Applicative Functor

We are able to transmit Procedures as a bundle because we cannot examine the result on individual Procedures until we have the entire applicative functor computation result — a key property of applicative functors.

Our simulated remote device now needs to handle lists of intermingled commands and procedures:

```
data RPrim = Command RCommand
           | Procedure RProcedure
             deriving Read

execRPrims :: [RPrim] -> IO [String]
execRPrims [] = return []
execRPrims (Command c : ps) = do
         execRCommand c
         execRPrims ps
execRPrims (Procedure p : ps) = do
         r  <- execRProcedure p
         rs <- execRPrims ps
         return (r:rs)
```

The device implementation uses execRPrims for both synchronous and asynchronous requests:

```
device :: Device
device = Device (liftM show . execRPrims . read)
                (void      . execRPrims . read)
```

We can now transmit bundles containing both commands and procedures. For example, we can measure the temperature before and after toasting in one request:

```
GHCi> (t1,t2) <- send device $
         liftA2 (,)
                (say "Good Morning" *> temperature)
                (toast 120          *> temperature)
Remote: Good Morning
Remote: Toasting...
...sleeping for 120 seconds...
Remote: Done!
GHCi> print (t1,t2)
(56,99)
```

Figure 5 shows the sequence diagram for this example. Notice how all three Procedures (and the Command) are sent to the toaster in a single packet.

7. Remote Binding

An alternative to combining procedures into one super-procedure with restrictions, would be to have the remote interpreter directly use the result of a procedure. We achieve this by introducing a *remote binding* mechanism, which locally acts like a remote procedure, but remotely acts like a command. To do so, we use some tricks for deeply embedded DSLs. To recap deep embeddings:

- We capture *expressions* by building data-structures that represent the operations inside the expression.
- We capture *functions* by extending our embedding with variables [12], and applying the function to a fresh variable.

As an example, towards building our remote monad model, consider appending the string “F” to a string:

```
f :: StringExpr -> StringExpr
f t = t <> "F"
```

We can capture `f` by using a small data structure for `StringExpr` that includes unique variables.

```
type Id = Int

data StringExpr = Var Id
                | Lit String
                | Append StringExpr StringExpr
```

With suitable `IsString` and `Monoid` instances for `StringExpr`, we can reify the function:

```
GHCi> f (Var 0)
Append (Var 0) (Lit "F")
```

Our remote monad will use the function-capture technique, by generating a new `Id` locally for each binding done remotely.

Definition. *A* **remote binding** *is the combination of a request to perform an action for its remote effects, where there is an interesting result value, and the remote interpreter binding this result value to a name remotely. The remote action is called a* **remote bindee**.

The result of a remote binding can be used immediately, without needing local interaction, because the result value resides remotely. Locally, we generate a new and unique `Id`, allowing the bindee to be bundled with commands. This is the trick — remote binders are a form of remote procedure calls, but because they return handles with known identifiers, we can transmit them to the remote site as remote commands.

We want our `Temperature` procedure to now be a `Bindee`, so that we can directly use the result remotely. We constrain the type of an `Bindee` in the same way we constrained the return type of a procedure: by using a GADT.

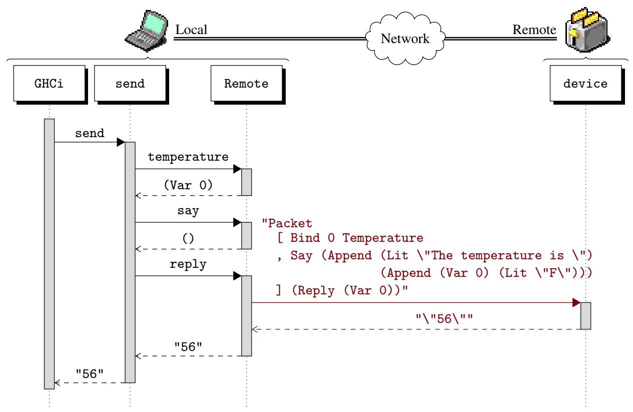
```
data Bindee :: * -> * where
  Temperature :: Bindee StringExpr
```

We now define the `Command` data type such that `Say` takes a `StringExpr` argument, and with a `Bind` command that will remotely bind an `Id` to a `Bindee`:

```
data Command where
  Say  :: StringExpr              -> Command
  Bind :: Id -> Bindee StringExpr -> Command
```

We have our utility functions `say` and `temperature`, and a function `bindBindee` that locally generates a name for the remotely bound value, before utilizing `sendCommand`.

```
say :: StringExpr -> Remote ()
say txt = sendCommand (Say txt)
```



```
"Packet
  [ Bind 0 Temperature
  , Say (Append (Lit \"The temperature is \")
                (Append (Var 0) (Lit \"F\")))
  ] (Reply (Var 0))"
```

Figure 6: Example of Remote Binding in a Strong Remote Monad

```
bindBindee :: Bindee StringExpr -> Remote Id
bindBindee e = do
   i <- newId
   sendCommand (Bind i e)
   return i

temperature :: Remote StringExpr
temperature = do
  i <- bindBindee Temperature
  return (Var i)
```

For returning the results of remote expressions, we provide a `Procedure` that takes a `StringExpr`, remotely executes it, and returns the result:

```
data Procedure :: * -> * where
  Reply :: StringExpr -> Procedure String

reply :: StringExpr -> Remote String
reply e = sendProcedure (Reply e)
```

This completes our set of transmittable primitives. The remaining definitions are almost identical to the strong remote monad model, except that the `Remote` monad now contains a numeric state that is used for fresh name generation, and the remote interpreter has an environment that can be updated. Using the above new machinery, and a suitable remote device, we can now run an example:

```
GHCi> send device $ do
        t <- temperature
        say ("The temperature is " <> t <> "F")
        reply t
Remote: The temperature is 56F
"56"
```

Figure 6 shows the sequence diagram for this example. Observe that the local monadic bind has been transported to the `device` for remote binding.

Remote bindings provide an extension of traditional serialization. While procedures can return anything serializable, remote bindings can return handles to remote things that cannot be realistically transmitted back to Haskell. For example, some languages support objects that, when serialized, lose their identity. With a remote binding, we have a handle to the original object.

```
newtype Object = Object Int
```

Haskell can pass around the abstract but serializable `Object`, and the remote interpreter has an array of objects, indexed by an integer. However, a caveat with remotely bound objects is that, by creating a handle to a remote object, there is no (easy) way to know when you can garbage collect it.

8. The Remote Monad Laws

A remote monad is a monad that has its evaluation function in a remote location, outside the local runtime system. In the remote monad design pattern, this achieved by providing:

- a remote monad — a set of primitive commands and procedures that form a monad; and
- a `send` function that facilitates remote execution of the primitive commands and procedures.

Towards a more systematic understanding of the remote monad, we propose the *remote monad laws*. Consider a `send` specialized to a specific destination or device d, with a remote monad $Remote$, and local monad $Local$. The `send` function is a natural transformation between $Remote$ and $Local$:

$$\texttt{send}_d \quad :: \quad \forall\, a\ .\ Remote\ a \rightarrow Local\ a$$

As part of a design pattern, the `send` function itself says nothing about how the individual commands and procedures within the $Remote$ monad are bundled into packets. Instead, as we have seen, `send` functions can implement different bundling algorithms.

We propose using the monad-transformer `lift` laws [22, 27], also known as the monad homomorphism laws, as our remote monad laws, because we are lifting the $Remote$ computation to the remote site, by $Local$ effect.

$$\texttt{send}_d\ \texttt{(return}\ a\texttt{)} \quad = \quad \texttt{return}\ a \qquad (1)$$
$$\texttt{send}_d\ \texttt{(}m\ \texttt{>>=}\ k\texttt{)} \quad = \quad \texttt{send}_d\ m\ \texttt{>>= (send}_d\ \texttt{.}\ k\texttt{)} \qquad (2)$$

Assuming these laws, the monad laws, and the laws relating functors and applicative functors to monads, the following morphism laws can be derived:

$$\texttt{send}_d\ \texttt{(pure}\ a\texttt{)} \quad = \quad \texttt{pure}\ a \qquad (3)$$
$$\texttt{send}_d\ \texttt{(}m_1\ \texttt{<*>}\ m_2\texttt{)} \quad = \quad \texttt{send}_d\ m_1\ \texttt{<*>}\ \texttt{send}_d\ m_2 \qquad (4)$$
$$\texttt{send}_d\ \texttt{(fmap}\ f\ m\texttt{)} \quad = \quad \texttt{fmap}\ f\ \texttt{(send}_d\ m\texttt{)} \qquad (5)$$

Laws (1) and (3) state that a `send` has no effect for pure computations. Laws (2) and (4) state that packets of remote commands and procedures preserve the ordering of their effects, and can be split and joined into different sized packets without side effects. Law (5) is a reflection of the fact that `send` captures a natural transformation.

8.1 Infinite Remote Monads

From observation of the laws, it is straightforward to split any *finite* sequence of *total Remote* primitives into arbitrarily sized packets. It should be possible to lift the finiteness pre-condition as follows:

- In the weak remote monad, each primitive invokes an individual RPC. Therefore, it is completely reasonable to have an infinite stream of primitives in the $Remote$ monad, in the same way that we can have infinite monadic computations in the `IO` monad. The `hArduino` library, discussed in §11.4, is an example of this.
- In the strong remote monad, it would be possible to have an infinite stream of primitives in the $Remote$ monad, *provided the packets are themselves finite*. If necessary, this could be artificially manufactured by putting an upper bound on the number of consecutive commands before sending a packet.

We leave lifting the finiteness requirement of the remote applicative functor to future work, observing that an applicative functor will likely require `Alternative`, or something similar, to achieve an interesting infinite denotation.

8.2 Networking

Making networks robust is a hard problem. For example, UDP packets are not guaranteed to arrive, or arrive in order, so are unsuitable, without additional infrastructure, for use in a remote monad. The remote monad laws assume the network works. However, having all remote calls wrapped in a `send` allows for `send` to enforce a specific policy; for example, time-out after 3 seconds and raise an exception in the $Local$ monad. Also, the $Remote$ monad could have control structures for exception handling built in. The advantage is that by having a structured way of calling remote services, this gives hooks to include systematic recovery services as part of the remote service API.

8.3 Threading

We have assumed a single-threaded local user and a single-threaded remote service, and the remote monad laws reflect this assumption. We could lift the local single-threaded assumption trivially, by using a lock on invocations of `send`, keeping the remote interactions single threaded. Alternatively, we could allow the local actions to be threaded, and have the thread usage reflect into the remote site. The interleaving semantics will certainly depend on the specific remote monad, and, as when running any side-effecting command on any multi-threaded system, interference patterns need to be considered and accounted for.

9. Extended Example: Blank Canvas

In our toaster models, the remote interpreter was also written in Haskell. This will not be the case in general. Blank Canvas is our Haskell library that provides the complete HTML5 Canvas API, using a strong remote monad. In this section, we describe how we used the remote monad design pattern to build this full scale API that remotely calls JavaScript.

The HTML5 standard, implemented by all modern web browsers, supports a rich canvas element. This canvas element provides a low-level 2-dimensional interface to the in-browser widget. The API provides approximately 30 methods, most of which are void methods called for their effect, and almost 20 settable attributes. Table 1 gives a complete list of the base API.

We name our remote monad `Canvas`, because it represents methods on the JavaScript `Canvas`. From examining Figure 1, all methods and all settable attributes can be transliterated into monadic Haskell functions in our remote monad. All methods and attributes are serializable values; there are no callbacks in this API.

Classifying the API as remote commands, bindings and procedures is all about deciding if a specific data structure is local or remote:

- `isPointInPath`, `toDataURL`, and `measureText`, are **procedures**, which return values to the Haskell runtime system, returning `Bool`, `Text` and `TextMetrics` respectively. (`TextMetrics` is simply a wrapper around a `Double`.)
- `createLinearGradient`, `createRadialGradient`, and `createPattern`, are **bindings**, because `CanvasGradient` and `CanvasPattern` are subtypes of an image class, which can only be used remotely.
- `ImageData` is a type that is local to Haskell. This is a design choice. `ImageData` is an array of RGB byte values, intended for pixel-level manipulations of images before calling `putImageData` to render to a canvas. Instead of reflecting an entire deep embedding of arithmetic and array operations, we make `ImageData` a wrapper around a Haskell byte vector, to be constructed Haskell-side, and provided as a serializable argument to `putImageData`. Thus, `getImageData` is a **procedure**.
- Everything else, including setting attributes, are **commands**.

Table 1: JavaScript API for HTML5 Canvas

	TRANSFORMATION
void	`save`()
void	`restore`()
void	`scale`(float x,float y)
void	`rotate`(float angle)
void	`translate`(float x,float y)
void	`transform`(float m11,float m12,float m21,float m22,float dx,float dy)
void	`setTransform`(float m11,float m12,float m21,float m22,float dx,float dy)
	TEXT
void	`fillText`(string text,float x,float y,[Optional] float maxWidth)
void	`strokeText`(string text,float x,float y,[Optional] float maxWidth)
TextMetrics	`measureText`(string text)
	PATHS
void	`beginPath`()
void	`fill`()
void	`stroke`()
void	`clip`()
void	`moveTo`(float x,float y)
void	`lineTo`(float x,float y)
void	`quadraticCurveTo`(float cpx,float cpy,float x,float y)
void	`bezierCurveTo`(float cp1x,float cp1y,float cp2x,float cp2y,float x,float y)
void	`arcTo`(float x1,float y1,float x2,float y2,float radius)
void	`arc`(float x,float y,float radius,float startAngle,float endAngle,boolean d)
void	`rect`(float x,float y,float w,float h)
boolean	`isPointInPath`(float x,float y)

FONTS, COLORS, STYLES AND SHADOWS (ATTRIBUTES)	
`globalAlpha` float	`globalCompositeOperation` string
`lineWidth` float	`lineCap` string
`lineJoin` string	`miterLimit` float
`strokeStyle` any	`fillStyle` any
`shadowOffsetX` float	`shadowOffsetY` float
`shadowBlur` float	`shadowColor` string
`strokeStyle` any	`fillStyle` any
`font` string	`textAlign` string
`textBaseline` string	

	DRAWING
void	`drawImage`(Object image,float dx,float dy, [Optional] ...)
void	`clearRect`(float x,float y,float w,float h)
void	`fillRect`(float x,float y,float w,float h)
void	`strokeRect`(float x,float y,float w,float h)
	STYLE ATTRIBUTES
CanvasGradient	`createLinearGradient`(float x0,float y0,float x1,float y1)
CanvasGradient	`createRadialGradient`(float x0,float y0,float r0,float x1,float y1,float r1)
CanvasPattern	`createPattern`(Object image,string repetition)
	IMAGES
string	`toDataURL`([Optional] string type, [Variadic] any args)
ImageData	`createImageData`(float sw, float sh)
ImageData	`getImageData`(float sx, float sy, float sw, float sh)
void	`putImageData`(ImageData imagedata, float dx, float dy, [Optional] ...)

The taxonomy of remote primitives allowed the design choice to be an informed choice. We maximize the size of our packets going to the JavaScript interpreter, though we flush the commands at the end of every `send`. We also add two meta-primitives to our API: `sync :: Canvas ()`, an empty procedure that ensures the pipeline is flushed; and `async :: Canvas ()`, an empty command that *asynchronously* flushes the pending commands.

We used a deep embedding of the `Canvas` monad in Blank Canvas. This is more of a historical accident: at the time we wrote Blank Canvas we thought the strong remote monad required such an embedding. However, as we can see from our models, it is possible to build a remote monad with a shallow embedding. The `send` function interprets the embedding of the `Canvas` GADT.

To give an example, consider drawing a line using Blank Canvas. All the required primitives are commands. So `send` bundles them up as a single packet, and transmits the following JavaScript (without the whitespace) to the browser, for evaluation:

```
           Haskell                        JavaScript
send context $ do              try{
        moveTo(50,50)            c.moveTo(50,50);
        lineTo(200,100)          c.lineTo(200,100);
        lineWidth 10             c.lineWidth = 10;
        strokeStyle "red"        c.strokeStyle = "red";
        stroke()                 c.stroke();
                               } catch(e) {
                                 alert('...');
                               }
```

For each Blank Canvas procedure primitive, we have a small wrapper, written in JavaScript, that uses currying to accept a unique "transaction" id, in this example the number 9, and the graphics context, `c`.

```
           Haskell                        JavaScript
send context $ do              try{
  isPointInPath(10,20)          IsPointInPath(10,20)(9,c);
                               } catch(e) {
                                alert('...');
                               }
```

The function `$.kc.reply`, from the package `kansas-comet` [21], sends a JavaScript value back to the Haskell program. The first argument is the transaction id, and the second argument is the returned (JavaScript) value. We then embed JavaScript's `isPointInPath` inside a Blank Canvas supporting function.

```
function IsPointInPath(x,y) {
    return function (u,c) {
        $.kc.reply(u,c.isPointInPath(x,y));
    }
}
```

For bindings, we allocate a global variable, and remember the unique number inside the proxy object.

```
         Haskell                      JavaScript
send context $ do            try{
   grd <- ⟪ BINDEE ⟫           var v_42 = ⟪ BINDEE ⟫
   ...                         ...
                             }catch(e){
                               alert('...');
                             }
```

In summary, the JavaScript generated is a direct transliteration of the function calls made in Haskell, using the remote monad as the guiding design pattern.

10. Blank Canvas in Practice

Out of the box, Blank Canvas is pac-man complete — it is a platform for simple graphics, classic video games, and building more powerful abstractions that use graphics. The key question is how well does Blank Canvas perform in practice. In this section, we give both empirical and anecdotal evidence.

10.1 Empirical Evidence

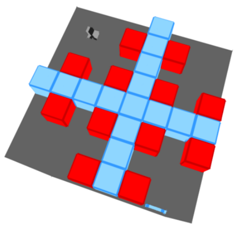
An important question is the cost of using the remote monad design pattern, albeit over a machine-local network. At first glance the cost of using the remote monad to implement Blank Canvas seem prohibitive. Blank Canvas transliterates each packet to a `Text` string, and then sends the packet over a network, where it is parsed by the JavaScript virtual machine. Normally, in a JavaScript application, the JavaScript program is parsed and compiled once, and the inner loops are run many times. But a loop in Blank Canvas will repeatedly transmit the contents of the loop body, requiring re-parsing and re-compilation each time. Even with the use of the remote monad design pattern, we expect to take a significant performance hit, above and beyond the cost of networking.

In order to quantify our ideas, we have measured the performance of Blank Canvas on several benchmark programs and compared them to native JavaScript. We have two classes of benchmarks: command benchmarks, which simply render to the canvas; and procedure benchmarks, where the inner loop of the benchmark invokes some form of query that requires a round-trip from server, to client, and back to the server. The JavaScript versions of the benchmarks were written in idiomatic JavaScript. The Blank Canvas tests were run using `criterion` [36], and the JavaScript tests used the `criterion` mechanism to estimate confidence.

We ran our benchmarks on both Chrome and Firefox, and on both Linux and OSX. The relative performance of our command benchmarks varied widely, depending on browser and benchmark, but on average, the cost of using Haskell and the Blank Canvas API was between a factor of 2 and 10 relative to native JavaScript. This is surprising and encouraging! However, the performance cost of our procedure benchmarks was significantly higher, up to a factor of 600 relative to native JavaScript. We expect that adding a strong remote applicative functor capability would significantly reduce the cost of the procedure benchmarks, because it would also allow Blank Canvas to bundle procedures. We plan to perform a more systematic study of the costs of the remote monad, and the benefits of the remote applicative functor, in the future.

10.2 Anecdotal Evidence

Blank Canvas has now been used by the students in four separate instances of our functional programming class. Students find it easy to understand, given the analog between the `IO` monad and the remote `Canvas` monad, with students often choosing to use Blank Canvas for their end-of-semester project. To give two examples, one end-of-semester project was Omar Bari and Dain Vermaak's Isometric Tile Game, that can be rotated in 3D in real-time; another project was Blankeroids, a playable asteroids clone, written by Mark Grebe, on top of Yampa [9] and `yampa-canvas` [39]. Both are shown here with the students' permission.



In summary, Blank Canvas, using the remote monad, is a viable way for students to draw pictures and write games in Haskell. Rendering graphics uses many more commands than procedures, so it (retrospectively) turns out to be an ideal candidate for the remote monad. Given that we have access to whole new capabilities, we consider the overheads of using the remote monad reasonable.

11. Related Work

Once the remote monad design pattern is understood, many instances of its use, or of related patterns, can be observed in the wild. In this section we discuss some of the existing instances, and present the type of the `send` analog, with the natural transformation highlighted in red. This list is not intended to be comprehensive, but rather to give a flavor of existing uses, or close uses, of the remote monad and related ideas.

11.1 Terminal Interaction

- The `ncurses` [33] package provides two levels of weak remote monads, in order to provide an API for moving the cursor around a terminal, and other terminal-specific services. The outer `send` has the type

  ```
  runCurses :: Curses a -> IO a
  ```

  and the inner `send` has the type

  ```
  updateWindow :: Window -> Update a -> Curses a
  ```

  This package compiles commands into ANSI-style command sequences to be execute directly on the terminal.

11.2 Database Access

- In the Haxl DSL, Marlow [29] uses the properties of applicative functors to issue database queries in parallel. The `send` function has the type:

  ```
  runHaxl :: Env u -> GenHaxl u a -> IO a
  ```

  Haxl is a DSL with a weak remote monad, with a strong remote applicative functor.

- `mongoDB` [23] is an API into the popular MongoDB NoSQL database that uses a weak remote monad. The `send` function has the type:

  ```
  access :: MonadIO m => Pipe
                      -> AccessMode
                      -> Database
                      -> Action m a -> m a
  ```

11.3 Browser / Server

- The rather ingenious Haste.App library [10, 11] is an instance of the remote monad design pattern. The Haste.App uses two Haskell compilers, a Haskell to JavaScript compiler for the local monad, and GHC for the remote monad evaluator, with the glue code between the two generated artifacts being automatically generated. The `send` function has the type:

  ```
  onServer :: Binary a => Remote (Server a) -> Client a
  ```

  The `Remote` is a wrapper to help stage the difference between the client and server compilations. The `Server` monad is the remote monad. The remoteness here is running on the server; the main program runs on the client browser.

- Sunroof is a compiler for a monadic JavaScript DSL [4, 20], developed by the University of Kansas. The compiler, which is considerably more involved than Blank Canvas, could be used stand-alone, as a part of a strong remote monad. There are three separate `send` commands:

  ```
  asyncJS ::          SunroofEngine -> JS t () -> IO ()
  syncJS  :: (...) => SunroofEngine -> JS t a
                                    -> IO (ResultOf a)
  rsyncJS :: (...) => SunroofEngine -> JS t a  -> IO a
  ```

  Using what we have learned from studying the remote monad design pattern, these three `send` functions could be combined into a single `send`.

11.4 Embedded Systems

- Levent Erkök's `hArduino` package [14] uses the weak remote monad design pattern. The send-command `withArduino` is a one-shot operation, and does not have the ability to return values. Instead, the monad represents the whole computation to be executed.

  ```
  withArduino :: Bool -> FilePath -> Arduino () -> IO ()
  ```

- Ben Gamari's `bus-pirate` package [18] allows access to the I^{2}C or SPI embedded device protocols via a serial port, using monadic primitives. The `send` command implements the remote monad directly.

```
runBusPirate :: FilePath -> BusPirateM a
                         -> IO (Either String a)
```

- The University of Kansas used an (as yet unpublished) strong remote applicative functor in the design of λ-bridge, an interface between Haskell and an FPGA, built on top of the open Wishbone [35] bus protocol.

```
send :: Board -> BusCmd a -> IO (Maybe a)
```

  The applicative functor structure allows multiple reads and writes to be combined into a bus transaction, and executed on an FPGA.

11.5 GUIs

- Heinrich Apfelmus' `threepenny-gui` [1] uses a user-interface element monad `UI` as a weak remote monad.

```
runUI :: Window -> UI a -> IO a
```

11.6 GPGPUs

- The popular `accelerate` package [7] uses a **remote array** design pattern to program a GPGPU. The `send` function takes an `Acc`, and returns a pure result.

```
runIn :: Arrays a => Context -> Acc a -> a
```

  This version of `send` does not use a local monad, but it can be considered to use the `Identity` monad. The `send` function can be purely functional specifically because the computations that are performed remotely are purely functional. Like other uses of the remote design pattern, `Acc` encodes the computation to be performed, in this case on the GPGPU. `Acc` is a DSL which provides array-based operators; such as `zipWith` and `map`. There is no applicative apply, or monadic bind, for `Acc`.

11.7 SMT solvers

- Levent Erkök's `sbv` package [15] provides access to SMT solvers, and uses a weak remote monad as its lowest level API.

```
runSymbolic' :: SBVRunMode -> Symbolic a -> IO (a, Result)
```

11.8 Games

- Douglas Burke's `mcpi` package [5] uses a weak remote monad to allow Haskell users to interact with a Minecraft server.

```
runMCPI :: MCPI a -> IO a
```

11.9 Other Related Works

This work is the marriage of Domain Specific Languages and Remote Procedure Calls. Both have a long and rich history.

Sun Microsystems implemented the first widely-distributed RPC implementation [31, 32]. The ideas were based on Birrell and Nelson's seminal work in this area [3]. All modern operating systems include RPC capabilities, building on these initial implementations. There is now an array of high-level libraries and frameworks for almost any modern language, offering RPC services such as D-Bus [28], which operates within a single machine, and Java's Jini, now called Apache River (`http://river.apache.org/`). The Haskell cloud-computing effort [13] includes support for RPCs, and can be considered such a middleware solution. There are also many low-level protocols for executing RPCs, such as JSON-RPC (`http://www.jsonrpc.org/`).

Domain Specific Languages are Haskell's forte. There are shallowly embedded DSLs [25, 26], with direct execution, and deeply embedded DSLs [12], with a generated structure representing the computation and a paired evaluator. There are also tagless-final DSLs [6], where there is no early commitment to a specific embedding, but instead class overloading in used to specify the API. Several Haskell DSLs, including Feldspar [37] and Obsidian [8], reify the structure of monadic code [40, 41] to generate external imperative code. Other DSLs, including CoPilot [38] and Ivory [24], use a shallow embedding of statements to capture the monadic structure. All of these DSLs could leverage the remote monad design pattern.

12. Conclusion and Future Work

This work was inspired by the question of what would an online evaluator of a deeply embedded DSL outside the Haskell heap look like. However, as we have seen, other DSL technologies can be used to generate the remote packets; the remote monad ideas are orthogonal to specific DSL implementation technologies. We thought we were writing a paper about deeply embedded DSLs and interesting GADT encodings. Instead, we discovered a design pattern and a small language for RPC APIs.

We have built a number of remote monads and remote applicative functors. Aside from the examples already documented above, we have also reimplemented the Minecraft API found in `mcpi`, but with a strong remote monad, and a general JSON-RPC framework in Haskell. In particular, the JSON-RPC protocol supports multiple batched calls, as well as individual calls. Currently, the user needs to choose between the monadic and applicative functor API. We will use this prototype as a test-bench to explore the combination of simultaneously being a strong remote monad and a strong remote applicative functor. The recent push to add applicative functor `do`-notation as a GHC-extension is something that we can take advantage of here.

Our simulated remote interpreters used regular data structures, rather than sharing the GADT used by `send`. This was for clarity, but also to avoid the complications of reifying GADTs. This is not a fundamental limitation: we have also implemented all our models using GADTs for the remote procedures, using a wrapper GADT called `Transport` to enable deserialization.

```
data Transport (m :: * -> *)
  = forall a . (Show a) => Transport (m a)
```

By providing a `Read` instance for `Transport`, we can hide the phantom type index from the `Read` instance, while ensuring that we have a `Show` instance for the phantom type. We expect that any full-scale implementation would use similar techniques when the remote interpreter uses Haskell.

A remote monad is a sweet spot between an RPC and a deeply embedded DSL, offering the possibility of rich FFIs for little effort and runtime cost. The remote monad design pattern allows a progression from weak, to strong, to deep embedding, giving a gentler pathway to implement deeply embedded DSLs. We have used the remote monad design pattern many times, and hope others will find the pattern as useful as we have.

Acknowledgments

We would like to thank Heinrich Apfelmus, Conal Elliott, and Levent Erkök for their useful discussions and insights, and the anonymous reviewers for their useful and constructive comments. This material is based upon work supported by the National Science Foundation under Grant No. 1117569 and Grant No. 1350901. Aleksander Eskilson, Ryan Scott and James Stanton were supported by the NSF REU initiative. The icons in the sequence diagrams were created by Tomoyuki Miyano, and made available online, at `http:/iconarchive.com/`.

References

[1] H. Apfelmus. Hackage package `threepenny-gui-0.6.0.2`, 2015.

[2] R. Bird. Using circular programs to eliminate multiple traversals of data. *Acta Informatica*, 21(3):239–250, 1984.

[3] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *Transactions on Computer Systems*, 2(1):39–59, 1984.

[4] J. Bracker and A. Gill. Sunroof: A monadic DSL for generating JavaScript. In *International Symposium on Practical Aspects of Declarative Languages*, volume 8324 of *LNCS*, pages 65–80. Springer, 2014.

[5] D. Burke. Hackage package `mcpi-0.0.1.2`, 2014.

[6] J. Carette, O. Kiselyov, and C. Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(05):509–543, 2009.

[7] M. M. T. Chakravarty, R. Clifton-Everest, G. Keller, S. Lee, B. Lever, T. L. McDonell, R. Newtown, and S. Seefried. Hackage package `accelerate-0.15.1.0`, 2015.

[8] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive array constructs in an embedded GPU kernel programming language. In *Workshop on Declarative Aspects and Applications of Multicore Programming*, pages 21–30. ACM, 2012.

[9] A. Courtney, H. Nilsson, and J. Peterson. The Yampa arcade. In *Haskell Workshop*, pages 7–18. ACM, 2003.

[10] A. Ekblad. Hackage package `haste-compiler-0.4.4.4`, 2015.

[11] A. Ekblad and K. Claessen. A seamless, client-centric programming model for type safe web applications. In *Haskell Symposium*, pages 79–89. ACM, 2014.

[12] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *Journal of Functional Programming*, 13(3):455–481, 2003.

[13] J. Epstein, A. P. Black, and S. Peyton Jones. Towards Haskell in the cloud. In *Haskell Symposium*, pages 118–129, 2011.

[14] L. Erkok. Hackage package `hArduino-0.9`, 2014.

[15] L. Erkok. Hackage package `sbv-4.4`, 2015.

[16] L. Erkök and J. Launchbury. Recursive monadic bindings. In *International Conference on Functional Programming*, pages 174–185. ACM, 2000.

[17] L. Erkök and J. Launchbury. A recursive do for Haskell. In *Haskell Workshop*, pages 29–37. ACM, 2002.

[18] B. Gamari. Hackage package `bus-pirate-0.6.2`, 2015.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[20] A. Gill and J. Bracker. Hackage package `sunroof-server-0.2.1`, 2014.

[21] A. Gill and A. Farmer. Hackage package `kansas-comet-0.3.1`, 2014.

[22] A. Gill and R. Paterson. Hackage package `transformers-0.4.3.0`, 2015.

[23] T. Hannan. Hackage package `mongoDB-2.0.5`, 2015.

[24] P. C. Hickey, L. Pike, T. Elliott, J. Bielman, and J. Launchbury. Building embedded systems with embedded DSLs. In *International Conference on Functional Programming*, pages 3–9. ACM, 2014.

[25] P. Hudak. Modular domain specific languages and tools. In *International Conference on Software Reuse*, pages 134–142. IEEE Press, 1998.

[26] D. Leijen and E. Meijer. Domain specific embedded compilers. In *Conference on Domain-Specific Languages*, pages 109–122. ACM, 1999.

[27] S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Symposium on Principles of Programming Languages*, pages 333–343. ACM, 1995.

[28] R. Love. Get on the D-BUS. *Linux Journal*, 2005(130):3, 2005.

[29] S. Marlow, L. Brandy, J. Coens, and J. Purdy. There is no fork: An abstraction for efficient, concurrent, and concise data access. In *International Conference on Functional Programming*, pages 325–337. ACM, 2014.

[30] C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18:1–13, 2008.

[31] S. Microsystems. RPC: Remote procedure call protocol specification. Technical report, RFC 1050, Apr. 1988.

[32] S. Microsystems. RPC: Remote procedure call protocol specification: Version 2. Technical report, RFC 1057, June 1988.

[33] J. Millikin. Hackage package `ncurses-0.2.11`, 2014.

[34] E. Moggi. Computational lambda-calculus and monads. In *Symposium on Logic in Computer Science*, pages 14–23. IEEE Press, 1989.

[35] OpenCores Organization. *Wishbone B4: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores*, 2010. URL `http://opencores.org/opencores,wishbone`.

[36] B. O'Sullivan. Hackage package `criterion-1.1.0.0`, 2015.

[37] A. Persson, E. Axelsson, and J. Svenningsson. Generic monadic constructs for embedded languages. In *Symposium on Implementation and Application of Functional Languages*, volume 7257 of *LNCS*, pages 85–99. Springer, 2012.

[38] L. Pike, N. Wegmann, S. Niller, and A. Goodloe. A do-it-yourself high-assurance compiler. In *International Conference on Functional Programming*, pages 335–340. ACM, 2012.

[39] N. Sculthorpe. Hackage package `yampa-canvas-0.2`, 2014.

[40] N. Sculthorpe, J. Bracker, G. Giorgidze, and A. Gill. The constrained-monad problem. In *International Conference on Functional Programming*, pages 287–298. ACM, 2013.

[41] J. Svenningsson and B. J. Svensson. Simple and compositional reification of monadic embedded languages. In *International Conference on Functional Programming*, pages 299–304. ACM, 2013.

[42] P. Wadler. Comprehending monads. In *Conference on LISP and Functional Programming*, pages 61–78. ACM, 1990.