The Remote Monad

Dissertation Defense

Justin Dawson
jdawson@ittc.ku.edu

Information and Telecommunication Technology Center
University of Kansas, USA

Sandwiches!

How do you make a sandwich?

Sandwiches!

How do you make a sandwich?

- get out the bread, ham, lettuce, cheese and condiments
- cut lettuce and cheese
- spread condiments on bread
- add remaining ingredients
- put bread and other ingredients away

Time taken: 2:00

Sandwiches!

How do you make ~~a~~ 2 sandwiches?

- get out the bread, ham, lettuce, cheese and condiments
- cut lettuce and cheese
- spread condiments on bread
- add remaining ingredients
- put bread and other ingredients away

Time taken: ~~2:00~~

Sandwiches!

How do you make ~~a~~ 2 sandwiches?

2x

- get out the bread, ham, lettuce, chees and condiments
- cut lettuce and cheese
- spread condiments on bread
- add remaining ingredients
- put bread and other ingredients away

Time taken: ~~2:00~~ 4:00

Sandwiches!

How do you make ~~a~~ 2 sandwiches?

- get out the bread, ham, lettuce, cheese and condiments
- 2x {
  - cut lettuce and cheese
  - spread condiments on bread
  - add remaining ingredients

  }
- put bread and other ingredients away

Time taken: ~~2:00~~ ~~4:00~~ 2:45

Would you like your sandwich toasted?

Bridging to the Internet of Things

- This toaster has artificial intelligence and can make toast, give you the temperature, and in this specific example, most notably talks
- Just as we avoided extra work with making sandwiches we want to avoid the network latency that comes from talking to our toaster

Outline

Remote Procedure Calls

Examples of usage:

- Supercomputing
- Cloud Computing
- Internet of Things

$$[\![ClientMachine]\!] \longmapsto \left[\!\!\left[\begin{matrix} Remote \\ Resource \end{matrix} \right]\!\!\right]$$

Remote Procedure Calls

Examples of usage:

- Supercomputing
- Cloud Computing
- Internet of Things

$$[\![\mathit{ClientMachine}]\!] \longmapsto \left[\!\!\left[\begin{array}{c} \mathit{Remote} \\ \mathit{Resource} \end{array} \right]\!\!\right]$$

Problem:

- RPCs are expensive because networks have latency

(Old) Solution:

- Multiple RPC requests per network transaction
- RPCs therefore amortize the cost of remoteness

New Problem:

- **Need a robust mechanism for bundling RPC calls without obfuscating the RPC API**

Remote Procedure Calls

What is needed for RPCs?

- A remote machine listening for requests
- A local machine that has knowledge of the remote API and protocol to be used
- A network transmission mechanism

Remote Procedure Calls

What is needed for RPCs?

- A remote machine listening for requests
- A local machine that has knowledge of the remote API and protocol to be used
- A network transmission mechanism

```xml
<?xml version="1.0"?>
<methodCall>
   <methodName>circleArea</methodName>
      <params>
         <param>
            <value><double>2.41</double></value>
         </param>
      </params>
</methodCall>
```

Remote Procedure Calls

What is needed for RPCs?

- A remote machine listening for requests
- A local machine that has knowledge of the remote API and protocol to be used
- A network transmission mechanism

```
--> {"jsonrpc": "2.0", "method": "subtract",
      "params": [42, 23], "id": 1}
<-- {"jsonrpc": "2.0", "result": 19, "id": 1}
```

Remote Procedure Calls

What is needed for RPCs?

- A remote machine listening for requests
- A local machine that has knowledge of the remote API and protocol to be used
- A network transmission mechanism

```
--> [
    {"jsonrpc": "2.0", "method": "sum",
     "params": [1,2,4], "id": "1"},
    {"jsonrpc": "2.0", "method": "subtract",
     "params": [42,23], "id": "2"}
    ]
<-- [
    {"jsonrpc": "2.0", "result": 7, "id": "1"},
    {"jsonrpc": "2.0", "result": 19, "id": "2"}
    ]
```

What sets Haskell apart from other languages?

- strongly typed with automatic inference
- no reassignment
- recursion/map/reduce instead of loops
- explicit side-effects
- determinicity
- expression evaluation instead of sequence evaluation

What sets Haskell apart from other languages?

- strongly typed with automatic inference
- no reassignment
- recursion/map/reduce instead of loops
- explicit side-effects
- determinicity
- expression evaluation instead of sequence evaluation
- **first-class control**

Functional Programming

- Pure Functions + Immutability

```
f(4) => 9
```

- Structures that can construct and compose effect out of pure functions

```
putStr "Hello " *> putStr "World"
```

- Two flavors of effect composition:
  - Applicative Functor
  - Monad (Super Applicative Functor)

```
addPure :: Int -> Int -> Int
addPure x y = x + y
```

```
addPure :: Int -> Int -> Int
addPure x y = x + y
```

```
addIO :: Int -> Int-> IO Int
addIO x y = do
     putStrLn "Writing to file"
     writeFile "tmp.txt" "side-effect"
     return (x + y)
```

Functors - Values wrapped in some context.

data Maybe a = Just a | Nothing

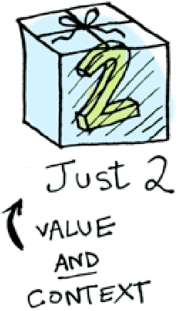


Image Credit: *Aditya Bhargava - adit.io*

Applicative Functors - Wrapped functions applied to wrapped values

Just $(+3) <*> $ Just 2

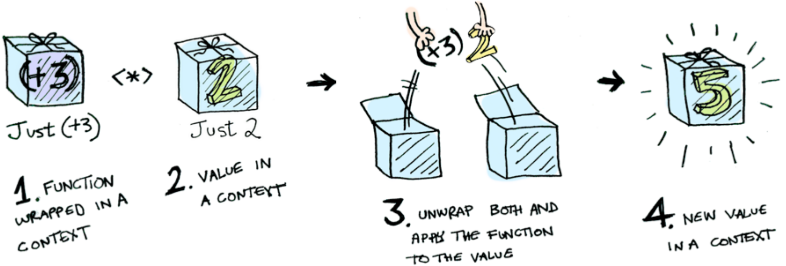


Image Credit: *Aditya Bhargava - adit.io*

Monads

- Used for side-effects
- Can be composed together
- Some require a run function before any side effects occur

```
return  :: (Monad m) => a -> m a
(>>=)   :: m a -> (a -> m b) -> m b
runM    :: m a -> ...
```

Monads

- Used for side-effects
- Can be composed together
- Some require a run function before any side effects occur

```
return   :: (Monad m) => a -> m a
(>>=)    :: m a -> (a -> m b) -> m b
runM     :: m a -> ...
```

Can we execute `runM` remotely?

Haskell Structures

Let's model running a monad remotely in Haskell

Toaster - IO

- Say {String}
- Temperature
- Uptime {String}


```
example :: IO (Int,Double)
example = do say "Hello "
             t <- temperature
             say "World!"
             u <- uptime "orange"
             return (t,u)
```

Toaster - GADT

```
data R where
   Say         :: String -> R ()
   Temperature ::           R Int
   Uptime      :: String -> R Double

say :: String -> R ()
say s = Say s

temperature :: R Int
temperature = Temperature

uptime :: String -> R Double
uptime s = Uptime s
```

KU

Internet of Things

Execution function

```
runR :: forall  a . R a -> IO a
runR (Say s)          = print s
runR (Temperature) = return 23
runR (Uptime s)      = getUptime s
```

runR gives us an interpretation of R in IO

Execution function

```
runR' :: forall  a . R a -> IO a
runR' (Say s)        = void $
  post "http://toaster.com/1234/say" (toJSON s)
runR' (Temperature) =
  get "http://toaster.com/1234?temperature"
runR' (Uptime s)     =
  get "http://toaster.com/1234?uptime=" ++ s
```

runR gives us an interpretation of R in IO

Naming things: Natural Transformation

In mathematics, `R a -> IO a` is called a natural transformation

Definition

A natural transformation arrow

$$F \xrightarrow{\bullet} G \quad \equiv \quad \forall \alpha.\ F\ \alpha \rightarrow G\ \alpha$$

In Haskell:

```
type f ~> g = forall a . f a -> g a

runR :: R ~> IO
```

Batching

We‘ve handled modeling single RPCs, can we incorporate batching?

First Attempt: `[R a] -> IO [a]`

- All results need to be of the same type
- Lacks composability

This is the space where most other batching RPC libraries reside
Let’s be more systematic

Remote Monad

```
data RM :: * -> * where
  Bind   :: RM a -> (a -> RM b) -> RM b
  Return :: a -> RM a
  Prim   :: R a -> RM a
```

Remote Monad

```
data RM :: * -> * where
  Bind   :: RM a -> (a -> RM b) -> RM b
  Return :: a -> RM a
  Prim   :: R a -> RM a

runRemoteMonad :: (R ~> IO) -> (RM ~> IO)

example :: IO (Int,Double)
example = (run $ runRemoteMonad runR) $ do
    say "Hello "
    t <- temperature
    say "World!"
    u <- uptime "orange"
    return (t,u)
```

Packet Bundling

Notation

$$
\begin{array}{cc}
[\![\begin{array}{c} \mathit{Remote} \\ \mathit{Monad} \end{array}]\!] & [\![\begin{array}{c} \mathit{Remote} \\ \mathit{Monad} \end{array}]\!] \\
\big\downarrow & \big\downarrow \\
[\![\begin{array}{c} \mathit{Weak} \\ \mathit{Packet} \end{array}]\!] & [\![\begin{array}{c} \mathit{Better} \\ \mathit{Packet} \end{array}]\!]
\end{array}
$$

Remote monad evaluator requires a packet evaluator

Serializing Bind

```
prim1 >>= \ x -> ... prim2 ...
```

Serializing Bind

```
prim1 >>= \ x -> ... prim2 ...
```

Definition

Command - a request to perform an action for remote effect, where there is no result value or temporal consequence

Definition

Procedure - a request to perform an action for its remote effect, where there is a result value or temporal consequence

```
cmd >>= \ () -> ... prim2 ...
```

Bundling Strategies

- Weak Bundling – Command | Procedure
- Strong Bundling – Command* Procedure

Can we get a better bundling?

Bundling Strategies

- Weak Bundling – Command | Procedure
- Strong Bundling – Command* Procedure
- Applicative Bundling – (Command | Procedure)*
  - `f <$> prim1 <*> prim2 <*> ...`

Bundling Strategies

- Weak Bundling – Command | Procedure
- Strong Bundling – Command* Procedure
- Applicative Bundling – (Command | Procedure)*
    - `f <$> prim1 <*> prim2 <*> ...`

```
example = do say "Hello "
             t <- temperature
             say "World!"
             u <- uptime "orange"
             return (t,u)
```

KU

Bundling Strategies

- Weak Bundling – Command | Procedure
- Strong Bundling – Command* Procedure
- Applicative Bundling – (Command | Procedure)*
  - `f <$> prim1 <*> prim2 <*> ...`

```
example =
  (,) <$> (say "Hello " *> temperature)
      <*> (say "World!" *> uptime "orange")
```

KU

Bundling Strategies

- Weak Bundling – Command | Procedure
- ~~Strong Bundling – Command* Procedure~~
- Applicative Bundling – (Command | Procedure)*
  - `f <$> prim1 <*> prim2 <*> ...`

Packet Bundling Landscape

⟦ Remote Monad ⟧ ↧ ⟦ Weak Packet ⟧

⟦ Remote Monad ⟧ ↧ ⟦ Strong Packet ⟧

⟦ Remote Monad ⟧ ↧ ⟦ Applicative Packet ⟧

⟦ Remote Applicative ⟧ ↧ ⟦ Weak Packet ⟧

⟦ Remote Applicative ⟧ ↧ ⟦ Strong Packet ⟧

⟦ Remote Applicative ⟧ ↧ ⟦ Applicative Packet ⟧

Packet Bundling Landscape

$$\left[\!\left[\begin{array}{c}\textit{Remote}\\\textit{Monad}\end{array}\right]\!\right] \longmapsto \left[\!\left[\begin{array}{c}\textit{Weak}\\\textit{Packet}\end{array}\right]\!\right] \qquad \left[\!\left[\begin{array}{c}\textit{Remote}\\\textit{Monad}\end{array}\right]\!\right] \longmapsto \left[\!\left[\begin{array}{c}\textit{Strong}\\\textit{Packet}\end{array}\right]\!\right] \qquad \left[\!\left[\begin{array}{c}\textit{Remote}\\\textit{Monad}\end{array}\right]\!\right] \longmapsto \left[\!\left[\begin{array}{c}\textit{Applicative}\\\textit{Packet}\end{array}\right]\!\right]$$

$$\left[\!\left[\begin{array}{c}\textit{Remote}\\\textit{Applicative}\end{array}\right]\!\right] \longmapsto \left[\!\left[\begin{array}{c}\textit{Weak}\\\textit{Packet}\end{array}\right]\!\right] \qquad \left[\!\left[\begin{array}{c}\textit{Remote}\\\textit{Applicative}\end{array}\right]\!\right] \longmapsto \left[\!\left[\begin{array}{c}\textit{Strong}\\\textit{Packet}\end{array}\right]\!\right] \qquad \left[\!\left[\begin{array}{c}\textit{Remote}\\\textit{Applicative}\end{array}\right]\!\right] \longmapsto \left[\!\left[\begin{array}{c}\textit{Applicative}\\\textit{Packet}\end{array}\right]\!\right]$$

Stack of evaluators

$$[\![\textit{Remote Monad}]\!] \quad \text{RemoteMonad} \rightsquigarrow \text{IO}$$

$$\downarrow \qquad\qquad \downarrow$$

$$[\![\textit{Applicative Packet}]\!] \quad \text{ApplicativePacket} \rightsquigarrow \text{IO}$$

```
runMonad :: (Monad m) => (ApplicativePacket R ~> m)
                      -> (RemoteMonad R ~> m)
```

KU

Remote Monad & Remote Applicative

```
data RemoteMonad p a where
   Appl :: RemoteApplicative p a ->
           RemoteMonad p a
   Bind :: RemoteMonad p a ->
           (a -> RemoteMonad p b) ->
           RemoteMonad p b
   ...

data RemoteApplicative p a where
   Prim   :: p a -> RemoteApplicative p a
   Ap         :: RemoteApplicative p (a -> b)
              -> RemoteApplicative p a
              -> RemoteApplicative p b
   Pure       :: a -> RemoteApplicative p a
```

Splitting up Monads

```
instance Applicative (RemoteMonad p) where
   pure a               = Appl (pure a)
   Appl f <*> Appl g = Appl (f <*> g)
   f <*> g              = Ap' f g

instance Monad (RemoteMonad p) where
   return   = pure
   m >>= k  = Bind m k
   m1 >> m2 = m1 *> m2
```

KU

Example

```
data R :: * where
  Say          :: String -> R ()
  Temperature  ::           R Int
  Uptime       :: String -> R Double

-- RemoteMonad R a
say :: String -> RemoteMonad R ()
say s = Appl $ Prim (Say s)
...

runR :: R ~> IO
runRPacket :: WeakPacket R ~> IO

send :: RemoteMonad R a -> IO a
send = run $ runMonad runRpacket
```

Other Investigations

- How to handle failure:
  - Alternative Construct `( a <|> b )`
  - Procedure encapsulates failure
  - Alternative Packet
  - Serialize Exceptions
- Remote Monad as a Monad Transformer
- Effects of bundling with `ApplicativeDo` Extension
- Haxl implementation
- Exception Handling

Case Studies

Transformations over natural transformations of monads results in a useful API and allows us to model a network stack

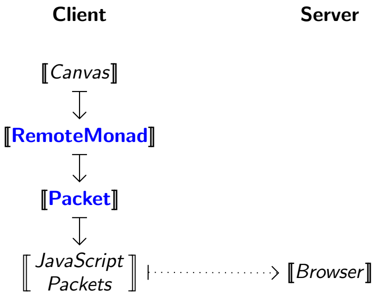
Goal: Show the Remote Monad being used in a variety of situations

$$\left[\!\!\left[\begin{array}{c}\mathit{Remote}\\ \mathit{Monad}\end{array}\right]\!\!\right] \longmapsto [\![\mathit{Packet}]\!]$$

Case Study
blank-canvas

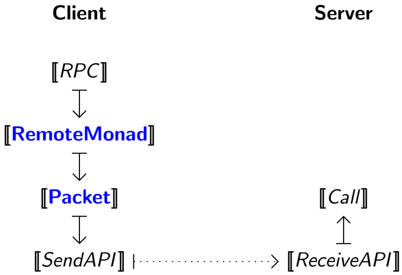
Blank Canvas

- Haskell code to interact and draw on HTML5 Canvas
- Weak, Strong, Applicative bundling
- Created by KU Functional Programming Group including Ryan Scott and David Young as well as other developers from the community



Remote JSON

- JSON-RPC implementation
- Id's used to pair results with requests
- Weak, Strong and Applicative Bundling

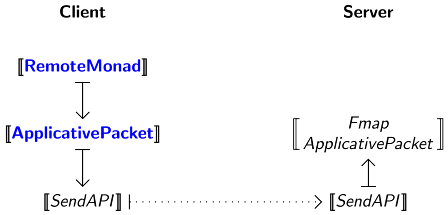


Case Study

remote-binary

Remote Binary

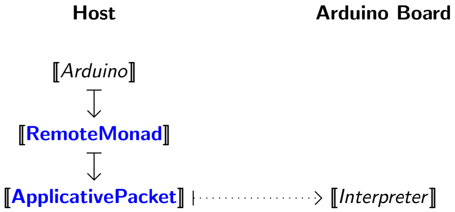
- Serialization to byte strings
- Results start with success/error byte
- Applicative Bundling



Case Study
haskino

Haskino

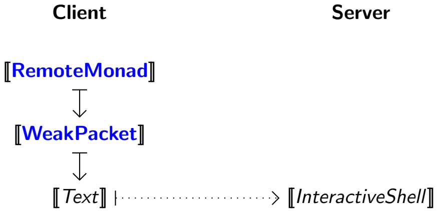
- Created by Mark Grebe
- Haskell programs interacting with an Arduino
- commands sent as bytecode to interpreter
- ported to use remote monad in 10 hours



Case Study
PlistBuddy

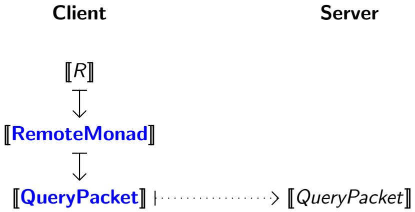
PlistBuddy

- Property List files (.plist)
- interacts with shell program
- Weak Bundling



Haxl

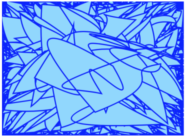
- Read only queries
- Query Bundling
- Optimized to use arbitrarily ordering capability



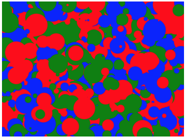
Performance

Command-Centric Benchmarks

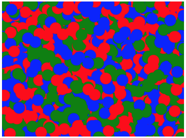
Bezier



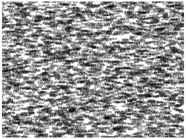
CirclesRandomSize



CirclesUniformSize

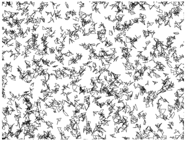


FillText



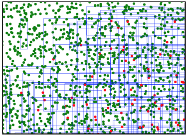
ImageMark

StaticAsteroids

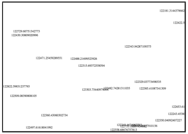


Rave

IsPointInPath



MeasureText



ToDataURL

Performance

Example: StaticAsteroids

```
benchmark :: CanvasBenchmark
benchmark ctx = do
  xs  <- replicateM 1000 $ randomXCoord ctx
  ys  <- replicateM 1000 $ randomYCoord ctx
  dxs <- replicateM 1000 $ randomRIO (-15, 15)
  dys <- replicateM 1000 $ randomRIO (-15, 15)
  send ctx $ do
           clearCanvas
           sequence_ [showAsteroid (x,y) (mkPts (x,y) ds)
                     | x <- xs
                     | y <- ys
                     | ds <- cycle $ splitEvery 6 $ zip dxs dys
                     ]
showAsteroid :: Point -> [Point] -> Canvas ()
showAsteroid (x,y) pts = do
  beginPath()
  moveTo (x,y)
  mapM_ lineTo pts
  closePath()
  stroke()
```

StaticAsteroids Packet Distribution

	# Packets	Commands per packet	Procedures per packet
Weak	1x	0	1
	9992x	1	0
Strong	1x	9992	1
Applicative	1x	9992	1

Table: StaticAsteroids Packet profile from a single test run

Performance

MeasureText Packet Distribution

	# Packets	Commands per packet	Procedures per packet
Weak	2002x	0	1
	5x	1	0
Strong	2000x	0	1
	1x	2	1
	1x	3	1
Applicative	1x	0	1
	1x	2	2000
	1x	3	1

Table: MeasureText Packet profile from a single run of the test

Performance

Benchmark	Weak (ms)	Strong (ms)	Applicative (ms)
Bezier	113.7	71.4	80.0
CirclesRandomSize	138.5	52.2	59.6
CirclesUniformSize	134.9	48.5	55.6
FillText	150.4	75.6	87.4
ImageMark	184.7	70.2	76.0
StaticAsteroids	374.3	112.4	128.2
Rave	48.8	20.9	26.0
IsPointInPath	447.8	359.1	199.3
MeasureText	682.9	689.2	142.8
ToDataURL	211.1	208.2	238.9

Table: Performance Comparison of Bundling Strategies (Chrome v64.0.3282.186)

- Weak - Globally slower
- Strong - fastest in non interaction
- Applicative - fastest with interactions but additional overhead cost when compared to Strong (Only noticeable when sending packets of the same composition)

Possibility of a hybrid packet between the Strong and Applicative

RPCs and batching RPCs:

- B.J. Nelson - PhD Dissertation on RPC
- Shakib et al. - Patent for bundling asyncronous calls with synchronous RPC
- Bogle et al. - Batched futures, batches as transactions
- Gifford et al. - RPCs as remote pipes, buffered sends
- Alfred Spector - No response for Asynchronous calls

Related Work

Haskell

Haxl - Facebook

- Uses Applicative Functor to split monad
- Procedures are read-only
- Optimized for parallelism

Free Delivery - Jeremy Gibbons

- Free Applicative Functors
- Applicative bundling

Cloud Haskell

- Distributed system using Erlang-style messages
- GHC Static pointers used for server functions

Contributions

Investigations

- Remote choices and failure handling
- Relationship between Haxl and Remote Monad
- Applicative packet optimization for blank-canvas

Publications/Talks

- Haskell Symposium 2015 paper
- IFL 2016 - invited talk
- Haskell Symposium 2017 paper

Open Source Libraries

- remote-monad library
- remote-json library
- remote-binary library

Future Work

- Remote Monad-Transformer
- Local IO
- Use of GHC static keyword Template Haskell
- Is there a better packet than applicative?

Conclusion

- We can systematically bundle primitives in an environment with first-class control
- We examined the properties of remote primitives yielding different bundling strategies
- We observe that we can model network stacks by chaining natural transformations together
- We conclude that applicative functors make a great packet structure