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 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
Sandwiches!

How do you make 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 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

1. Remote Procedure Calls (RPCs)
2. Introducing Haskell
3. Remote Monad (and Remote Applicative Functors)
4. Case Studies of Remote Monad Usage
5. Performance of Remote Monad in Situ
6. Related Work
7. Conclusion
Remote Procedure Calls

Examples of usage:
- Supercomputing
- Cloud Computing
- Internet of Things

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

Examples of usage:
- Supercomputing
- Cloud Computing
- Internet of Things

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

<?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

```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

```json
--> {"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

```json
--> [  
   {"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"}  
]  
```
Why Haskell?

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
  \[ f(4) \Rightarrow 9 \]
- Structures that can construct and compose effect out of pure functions
  \[ \text{putStr } "Hello\" \,*>\,*\text{putStr } "World" \]
- 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)
Haskell Structures
Applicative Functors

Functors - Values wrapped in some context.

data Maybe a = Just a | Nothing

Image Credit: Aditya Bhargava - adit.io
Haskell Structures

Applicative Functors

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

\[
\text{return} :: (\text{Monad } m) \Rightarrow a \rightarrow m a \\
(\gg\gg=) \quad :: m a \rightarrow (a \rightarrow m b) \rightarrow m b \\
\text{runM} \quad :: m a \rightarrow \ldots
\]
Haskell Structures
Monads

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 \texttt{runM} remotely?
Haskell Structures

Let’s model running a monad remotely in Haskell
Toaster - IO

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

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

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
Execution function

\[
\text{runR} :: \forall a . \text{R } a \rightarrow \text{IO } a \\
\text{runR} (\text{Say } s) = \text{print } s \\
\text{runR} (\text{Temperature}) = \text{return } 23 \\
\text{runR} (\text{Uptime } s) = \text{getUptime } s
\]

runR gives us an interpretation of R in IO
Execution function

\[
\text{runR'} :: \forall a . \text{R} a \rightarrow \text{IO} a
\]

\[
\text{runR'} (\text{Say } s) = \text{void } \$
\]
\[
\text{post } "http://toaster.com/1234/say" (\text{toJSON } s)
\]

\[
\text{runR'} (\text{Temperature}) =
\]
\[
\text{get } "http://toaster.com/1234?temperature"
\]

\[
\text{runR'} (\text{Uptime } s) =
\]
\[
\text{get } "http://toaster.com/1234?uptime=" ++ s
\]

\[
\text{runR gives us an interpretation of R in IO}
\]
In mathematics, \( R \ a \rightarrow IO \ a \) is called a natural transformation.

**Definition**

A natural transformation arrow

\[
F \rightarrow G \equiv \forall \alpha. \ F \alpha \rightarrow G \alpha
\]

In Haskell:

\[
\text{type } f \sim g = \text{forall } a . \ f \ a \rightarrow g \ a
\]

\[
\text{runR :: } R \sim IO
\]
We’ve handled modeling single RPCs, can we incorporate batching?

First Attempt: \([R \ a] \rightarrow IO \ [a]\)

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

```haskell
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

Remote Monad

[ Weak Packet ]

Remote Monad

[ Better Packet ]

Remote monad evaluator requires a packet evaluator
Serializing Bind

\[\text{prim1} \gg= \backslash x \to \ldots \text{prim2} \ldots\]
### Serializing Bind

**Definition**

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

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

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

```haskell
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)*
  - \( f <\$> \text{prim1} <\$\times> \text{prim2} <\$\times> \ldots \)

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

\[
\text{example} = \text{do say} \ "Hello \ \\
\text{t} <- \text{temperature} \\
\text{say} \ "World!" \\
\text{u} <- \text{uptime} "orange" \\
\text{return} (\text{t}, \text{u})
\]
Bundling Strategies

- Weak Bundling – Command | Procedure
- Strong Bundling – Command* Procedure
- Applicative Bundling – (Command | Procedure)*
  - \( f <$> \text{prim1} <*> \text{prim2} <*> ... \)

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

- Weak Bundling – Command | Procedure
- Strong Bundling – Command* Procedure
- Applicative Bundling – (Command | Procedure)*
  - \( f <$> \text{prim1} <*> \text{prim2} <*> \ldots \)
Packet Bundling Landscape

- Remote Monad
  - Weak Packet
  - Remote Applicative
    - Weak Packet
  - Remote Applicative
    - Strong Packet
  - Remote Applicative
    - Applicative Packet
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
Stack of evaluators

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

\[
\begin{array}{c}
\text{Remote Monad} \\
\downarrow \\
\text{Packet}
\end{array}
\]
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

Client

Canvas

RemoteMonad

Packet

JavaScript Packets

Server

Browser
Remote JSON

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

Remote Binary

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

```
Host
  [Arduino]
     ↓
  [RemoteMonad]
     ↓
  [ApplicativePacket] ———> [Interpreter]

Arduino Board
```
PlistBuddy

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

```
Client
  └── [RemoteMonad]
      └── [WeakPacket]
          └── [Text] ───> [InteractiveShell]

Server
```
Case Study

Haxl

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

Client

\[
\begin{align*}
& [R] \\
\downarrow & \quad \text{(RemoteMonad)} \\
\downarrow & \\
[QueryPacket] & \rightarrow [QueryPacket]
\end{align*}
\]

Server
Performance
Command-Centric Benchmarks

- Bezier
- CirclesRandomSize
- CirclesUniformSize
- FillText
- ImageMark
- StaticAsteroids
- Rave
Performance
Procedure-Centric Benchmarks

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()
## Performance

### StaticAsteroids Packet Distribution

<table>
<thead>
<tr>
<th></th>
<th># Packets</th>
<th>Commands per packet</th>
<th>Procedures per packet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak</td>
<td>1x 9992x</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Strong</td>
<td>1x 9992</td>
<td>9992</td>
<td>1</td>
</tr>
<tr>
<td>Applicative</td>
<td>1x 9992</td>
<td>9992</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table:** StaticAsteroids Packet profile from a single test run
# Performance

MeasureText Packet Distribution

<table>
<thead>
<tr>
<th></th>
<th># Packets</th>
<th>Commands per packet</th>
<th>Procedures per packet</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Weak</strong></td>
<td>2002x</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>5x</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td><strong>Strong</strong></td>
<td>2000x</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1x</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1x</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><strong>Applicative</strong></td>
<td>1x</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1x</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td></td>
<td>1x</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

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

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

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

- 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
Related Work
Outside of Haskell

RPCs and batching RPCs:

- B.J. Nelson - PhD Dissertation on RPC
- Shakib et al. - Patent for bundling asynchronous 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