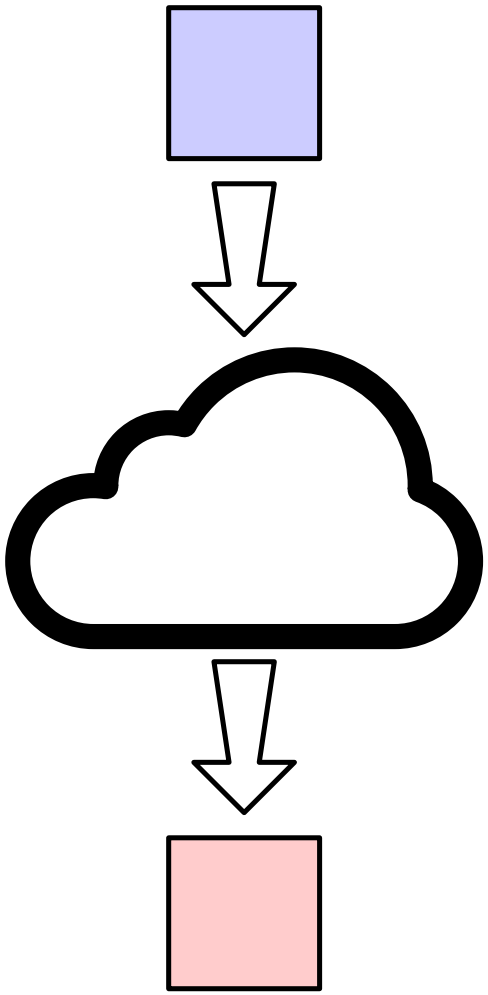
Free Delivery

Jeremy Gibbons

S-REPLS#3, Kent, April 2016

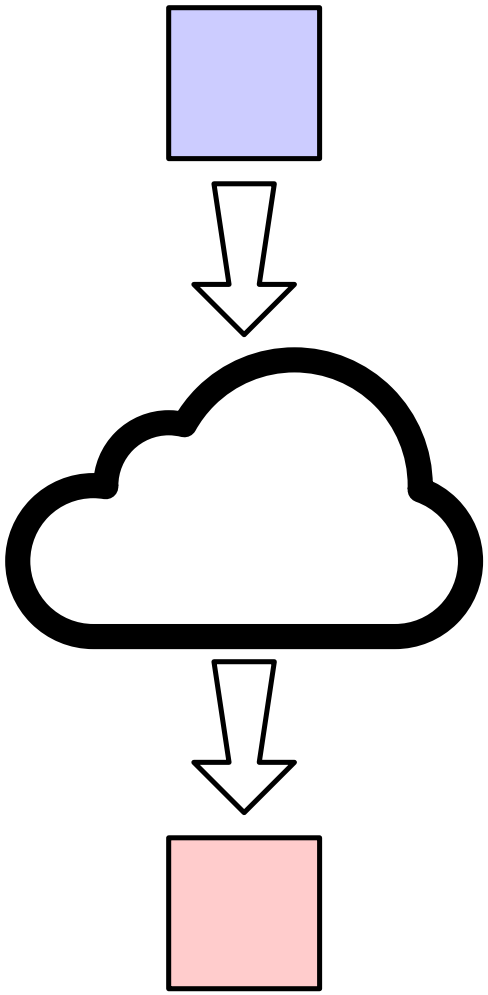
1. Deutsch's "fallacies of distributed computing"

i. the network is reliable
ii. latency is zero
iii. bandwidth is infinite
iv. the network is secure
v. topology doesn't change
vi. there is one administrator
vii. transport cost is zero
viii. the network is homogeneous

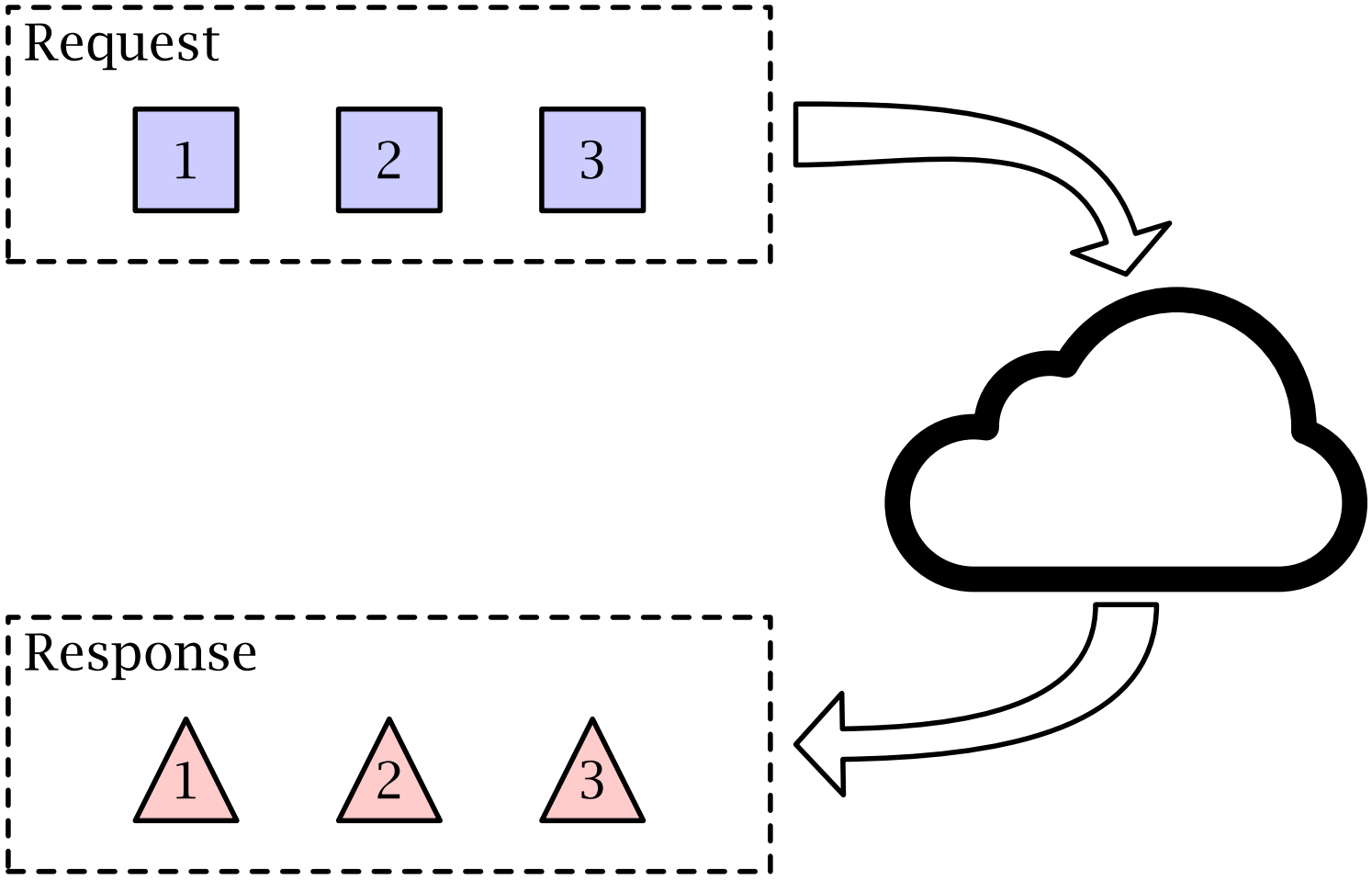


1. Deutsch's "fallacies of distributed computing"

i. the network is reliable
ii. **latency is zero**
iii. bandwidth is infinite
iv. the network is secure
v. topology doesn't change
vi. there is one administrator
vii. transport cost is zero
viii. the network is homogeneous



Fighting latency via batching



2. IoT toaster

```
data Command :: * → * where
  Say   :: String → Command ()
  Toast :: Int    → Command ()
  Sense :: ()     → Command Integer

data Name = NSay | NToast | NSense
```

(http://zurtech.deviantart.com/)

Remote procedure calls

Serialization and deserialization:

$$
\begin{aligned}
&\mathit{show} &&:: \mathit{Command}\ r \to \mathit{String} \\
&\mathit{readCommand} &&:: \mathit{String} \to (\mathit{Name}, \mathit{String}) \\
&\mathit{readReply} &&:: \mathit{Command}\ r \to \mathit{String} \to r
\end{aligned}
$$

Server-side and client-side wrappers:

$$
\begin{aligned}
&\mathit{server} :: (\mathit{String} \to \mathit{IO}\ \mathit{String}) \to \mathit{IO}\ () \\
&\mathit{client} :: \mathit{String} \to \mathit{IO}\ \mathit{String}
\end{aligned}
$$

Implementing the toaster

```
execSay :: String → IO ()
execSay s = putStrLn s

execToast :: Int → IO ()
execToast n = do putStr ("Toasting...")
                 threadDelay (1000000 * n)
                 putStrLn ("done!")

execSense :: IO Integer
execSense = randomRIO (0, 100)

execCommand :: String → IO String
execCommand s = case readCommand s of ...
```

3. Free monads

```
class Functor m ⇒ Monad m where     -- a variation
  return :: a          → m a
  join   :: m (m a) → m a

data FreeM :: (∗ → ∗) → ∗ → ∗ where
  Var :: a                  → FreeM f a
  Com :: f (FreeM f a) → FreeM f a

instance Functor f ⇒ Monad (FreeM f) where
  return = Var
  join (Var x)    = x
  join (Com xs) = Com (fmap join xs)
```

For example, a *FreeM Pair* is a binary tree.

Turning commands into a functor

Command has the right kind to be a functor, but has no *fmap*.
Use the *co-Yoneda trick*!

```
data Action a = ∀ r. Action (Command r, r → a)

instance Functor Action where
  fmap f (Action (c, k)) = Action (c, f · k)
```

Then we can define programs:

```
type Program a = FreeM Action a
```

A *FreeM Action* has branching structure matching the *Command*s.

With only boolean commands, *Action* ≃ *Pair* and *Program*s are binary trees:

```
data Command :: ∗ → ∗ where
  Branch :: () → Command Bool
```

Some monadic programs

```
say :: String → Program ()
say s = do { Com (Action (Say s, Var)) }

toast :: Int → Program ()
toast n = do { Com (Action (Toast n, Var)) }

sense :: Program Integer
sense = do { Com (Action (Sense (), Var)) }
```

composed in various ways:

```
straight :: Program ()
straight = do { say "hello"; toast 3; say "goodbye" }

branch :: Program ()
branch = do { t ← sense; if t < 80 then toast 3 else say "hot" }
```

Finitary actions

But these programs are difficult to serialize, and hence to distribute.

Restrict attention to *finitary* branching commands:

```
data ActionF a = ∀ r.(Bounded r, Enum r) ⇒
                 ActionF (Command r, r → a)

instance Show a ⇒ Show (ActionF a) where
  show (ActionF (c, k)) = show c ++ "  " ++
    show [show (k r) | r ← [minBound .. maxBound]]

type ProgramF a = FreeM ActionF a
```

Now we *can* serialize programs.

But should we?

4. Free applicatives

With monadic programs, later computations may depend on earlier results...

```
class Functor f ⇒ Applicative f where
  pure :: a → f a
  (⊛)  :: f (a → b) → f a → f b

data FreeA :: (∗ → ∗) → ∗ → ∗ where
  Pure :: a →                          FreeA f a
  More :: FreeA f (b → a) → f b → FreeA f a

instance Functor f ⇒ Applicative (FreeA f) where
  pure = Pure
  Pure f     ⊛ y = fmap f y
  More h x ⊛ y = More (fmap flip h ⊛ y) x
```

Eg, a *FreeA Pair* consists of an n-ary function and n pairs of arguments.

Some applicative programs

Now commands need only have *Read*able results:

$$\textbf{data}\ \mathit{ActionA}\ a = \forall r.\mathit{Read}\ r \Rightarrow \mathit{ActionA}\ (\mathit{Command}\ r, r \to a)$$
$$\textbf{type}\ \mathit{ProgramA}\ a = \mathit{FreeA}\ \mathit{ActionA}\ a$$

and programs are necessarily straight-line:

$$\mathit{sayA} :: \mathit{String} \to \mathit{ProgramA}\ ()$$
$$\mathit{sayA}\ s = \mathit{More}\ (\mathit{Pure}\ \mathit{id})\ (\mathit{ActionA}\ (\mathit{Say}\ s, \mathit{id}))$$

$$\mathit{toastA} :: \mathit{Int} \to \mathit{ProgramA}\ ()$$
$$\mathit{toastA}\ n = \mathit{More}\ (\mathit{Pure}\ \mathit{id})\ (\mathit{ActionA}\ (\mathit{Toast}\ n, \mathit{id}))$$

$$\mathit{senseA} :: \mathit{ProgramA}\ \mathit{Integer}$$
$$\mathit{senseA} = \mathit{More}\ (\mathit{Pure}\ \mathit{id})\ (\mathit{ActionA}\ (\mathit{Sense}\ (), \mathit{id}))$$

$$\mathit{straightA} :: \mathit{ProgramA}\ (\mathit{Integer}, \mathit{Integer})$$
$$\mathit{straightA} = \mathit{pure}\ (\lambda t\ ()\ t' \to (t, t')) \circledast \mathit{senseA} \circledast \mathit{toastA}\ 3 \circledast \mathit{senseA}$$

Serialization and deserialization

We can't serialize whole programs, but we don't need to—only the sequence of commands:

```
serializeA :: ProgramA a → [String]
serializeA (Pure _)                      = []
serializeA (More p (ActionA (c, _))) = show c : serializeA p
```

which we can then zip with a sequence of results:

```
deserializeA :: ProgramA a → [String] → a
deserializeA (Pure a) [] = a
deserializeA (More p (ActionA (c, k))) (s : ss)
    = deserializeA (p ⊛ Pure (k (readReply c s))) ss
```

5. Conclusions

- *batched invocation* is a matter of distributing programs
- *free monads* are the obvious tool
- but they are *too general*
- *free applicatives* have precisely the right power
- new *ApplicativeDo* proposal is super-convenient

- inspired by Andy Gill's "The Remote Monad Design Pattern"
- thanks to Matt Pickering for helpful suggestions
- paper under review—watch this space!

`http://www.cs.ox.ac.uk/jeremy.gibbons/`