Free Delivery

Functional Pearl

Jeremy Gibbons
University of Oxford, UK
http://www.cs.ox.ac.uk/jeremy.gibbons/

Abstract

Remote procedure calls are computationally expensive, because network round-trips take several orders of magnitude longer than local interactions. One common technique for amortizing this cost is to *batch* together multiple independent requests into one compound request. Batching requests amounts to serializing the abstract syntax tree of a small program, in order to transmit it and run it remotely. The standard representation for abstract syntax is to use *free monads*; we show that *free applicative functors* are actually a better choice of representation for this scenario.

Categories and Subject Descriptors C.2.4 [*Distributed Systems*]: Distributed applications; D.3.2 [*Language Classifications*]: Functional languages; F.3.2 [*Semantics of Programming Languages*]: Algebraic approaches to semantics

Keywords Remote procedure call, batched request, free monad, free applicative functor

1. Introduction

Distributed computing is hard. Peter Deutsch famously codified a number of *fallacies of distributed computing* (Deutsch 1994) concerning issues such as reliability, cost, and heterogeneity, which together make distributed systems trickier to get right than centralized systems. Distributed computing middleware is usually designed to make remote procedure calls look as much like local calls as possible, in order to hide these issues, and the unwary developer is thereby tempted into falling foul of every one of Deutsch's fallacies.

This paper is specifically concerned with the second of Deutsch's fallacies, that "latency is zero". Good practice in the modular construction of centralized systems is to provide *fine-grained* interfaces, so that each procedure deals with a single cohesive concern. But for a distributed system, this practice has to be weighed against the second fallacy: because of latency, it is much more efficient to provide *coarse-grained* interfaces, handling as much as possible with a single call and hence a single network round-trip. For example, Josuttis (2007) argues that whereas a local service dealing with customer records might provide separate methods to obtain a customer's name, address, and payment details, a remote service ought instead (or additionally) to provide one compound method to obtain all three attributes at once. Similarly, Fowler (2002) describes a REMOTE FAÇADE pattern, which "provides a coarse-grained façade on fine-grained objects to improve efficiency over a network".

But coarse-grained interfaces cause a different problem, characterized by Deutsch's third fallacy, namely that "bandwidth is infinite". What if a client requires just a customer's name and address and not the payment details? Calling separate methods for each attribute wastes a round trip. Calling the compound method to obtain three attributes then discarding one attribute wastes bandwidth. Implementing a dedicated method to return just the two attributes needed by this particular client violates modularity and clutters the interface—the number of different attribute combinations is exponential in the number of attributes. What is the poor distributed system developer to do?

One compromise presents itself: design the remote system to provide a fine-grained interface, promoting modularity, but also support a mechanism to *batch* up multiple small requests into a single compound request, minimizing latency. Of course, this only works in the case that later requests are independent of earlier responses. This idea has been rediscovered several times. Liskov et al. (1988) describe a *stream* abstraction for remote procedure calls, which among other features can batch together a sequence of small independent requests into a single message packet. Bogle and Liskov (1994) adapt this idea to *batched futures*, whereby cross-domain calls are simply collected together at the point of request, but not sent until the client actually needs the results. Ibrahim et al. (2009) introduce *remote batch invocation* as an extension to Java, automatically compounding remote method invocations. These recurring reinventions justify Fowler (2002)'s REMOTE FAÇADE pattern.

More recently, Gill et al. (2015) describe the REMOTE MONAD design pattern in Haskell, whereby remote procedure calls in the *IO* monad can be queued, as long as they are asynchronous and return void ("commands"); the queue is flushed and all requests sent when queueing a synchronous call or one that returns a result (a "procedure"). Gill et al. also point out that applicative functors are a more appropriate abstraction than monads for batched remote invocation, because they precisely capture the constraint that later requests must be independent of earlier responses.

Still, Gill et al.'s construction is more ad hoc than it need be. They implement queueing from first principles, combining the *IO* monad for interaction with the *State* monad for maintaining the queue. Moreover, in order to batch together procedures in the applicative style, they make essential use of Haskell's rather sophisticated recursive **do** notation (Erkök and Launchbury 2000) and lazy evaluation. The contribution of this paper is to show that their sophistication is unnecessary, and that the necessary definitions can essentially be obtained for free—specifically, by exploiting *free applicative functors*.

This paper is a literate Haskell program; the extracted code is available online at `http://www.cs.ox.ac.uk/jeremy.gibbons/publications/delivery.hs`.

2. Scenario

For purposes of comparison, we adopt Gill et al. (2015)'s 'internet-of-things toaster' example. This is a device that can be instructed to display a message on a built-in screen, to toast for a certain amount of time, and to sense and return its current temperature:

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

Note the essential use of a generalized algebraic datatype: a *Command* includes its argument as part of its value, and expresses its return type via a *phantom type index* (Hinze 2003).

The main point to be made with the choice of example is to contrast the high-level, typed, computationally intensive capabilities available on the client side against the low-level, perhaps untyped, more limited capabilities on the server side. Of course, nothing of significance in what follows would change if a different example were chosen—for example, one with some truly stateful server-side behaviour.

2.1 Serialization

We will need to serialize and deserialize commands for transmission. For simplicity, we will use plain *String*s as the wire format. Serialization is so straightforward that it can be derived automatically:

```
deriving instance Show (Command r)
```

Deserialization is rather harder, because of the GADT: what type should be returned when deserializing a given string? We deliberately do not try to do anything too fancy type-wise here, because deserialization of a command happens on the server side where there are limited capabilities. However, we can at least read the 'name' of a command:

```
data Name = NSay | NToast | NSense
  deriving Show
```

and return that, together with any remaining payload:

```
readCommand :: String → (Name, String)
readCommand s = head $
  [(NSay, s')    | ("Say", s')    ← lex s] ++
  [(NToast, s')  | ("Toast", s')  ← lex s] ++
  [(NSense, s')  | ("Sense", s')  ← lex s]
```

Here, $lex :: String \to [(String, String)]$ is a standard Haskell function to extract the first token from a string, which is assumed to represent an expression. For example,

```
readCommand (show (Toast 3)) = (NToast, "  3")
```

Meanwhile, back on the client side, if we already know the command, we can use its type index to tell us what type of reply to expect from a response string:

```
readReply :: Command r → String → r
readReply (Say   _) s = read s
readReply (Toast _) s = read s
readReply (Sense _) s = read s
```

One could make explicit the association between commands and their names by lifting the type *Name* to a data kind (Yorgey et al. 2012); but this does not seem to prevent all possible ways of mismatching requests and responses, so in the interests of keeping things simple we will refrain from making this refinement.

2.2 Remote Procedure Calls

We will communicate with our toaster by remote procedure calls over TCP. We factorize the code as follows. We represent the server behaviour as a function of type

```
type ServerBehaviour = String → IO String
```

It should accept a serialized request as its argument, and return a serialized response as its result, typically having I/O effects on the toaster in the process, but not itself performing any network interaction. We subject this function to a server-side wrapper

```
server :: ServerBehaviour → IO ()
```

that encapsulates the toaster's network interaction. We identify a complementary wrapper encapsulating the client-side network interaction

```
client :: String → IO String
```

which takes a serialized request as argument and returns a serialized response as result.

Given these *server* and *client* wrapper functions, and a suitable argument $ts :: ServerBehaviour$ representing the toaster's behaviour, we can run the server program

```
mainServer :: IO ()
mainServer = server ts
```

on one machine (representing the toaster), and the client program

```
mainClient :: String → IO ()
mainClient s = do
  response ← client s
  putStrLn ("Received " ++ response)
```

on another. The server and client wrappers are there solely to facilitate the network interaction; in particular, when *mainClient s* is run on the same machine as *mainServer*, then the combined behaviour should be the same as running

```
do
  response ← ts s
  putStrLn ("Received " ++ response)
```

without any network interaction. Since networking per se is not our primary interest, we relegate to Appendix A simple Haskell implementations of the *server* and *client* wrappers, and turn our attention to the *ServerBehaviour*.

2.3 Individual Requests

We now have the ingredients to send individual requests to the toaster. If we define the toaster's behaviour for each of the three commands as follows:

```
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)
```

(for the purposes of illustration, *execSense* simply picks a temperature uniformly at random between 0 and 100, rather than manipulating actual hardware), then we can assemble these into the overall server-side behaviour:

```
execCommand :: ServerBehaviour
execCommand s = case readCommand s of
  (NSay, s')    → do {r ← execSay (read s'); return (show r)}
  (NToast, s')  → do {r ← execToast (read s'); return (show r)}
  (NSense, s')  → do {r ← execSense; return (show r)}
commandServer :: IO ()
commandServer = server execCommand
```

In light of our stated intention that the server side has only limited and low-level computational capabilities, we would typically not use Haskell to program it; we would use something lower-level, like C. Then *commandServer* should be thought of as a Haskell specification for the required C implementation.

However, one might perfectly well use Haskell to program an internet client to interact with the toaster. For example, this client takes a single command, converts it to a string, sends it as a request, then reads the appropriate type of result from the response as determined by the command that was sent:

```
commandClient :: Command r → IO r
commandClient c = do
                    r ← client (show c)
                    return (readReply c r)
```

Having started *commandServer* remotely, one can then interact with the toaster by invoking individual commands locally:

```
*Main> commandClient (Say "Howdy doodly do!")
*Main> commandClient (Toast 3)
*Main> commandClient (Sense ())
78
```

3. Free Monads

We now return to the question of batching up multiple requests. As a first approximation, we can think of this as passing not a single *command* from the client for remote execution on the server, but a whole *program* built from such commands. Of course, this raises the question of whether it is reasonable to expect programs to be mobile in this way: should a toaster have the capability to run a program? This is indeed a worthy question, and we will return to it at the end of this section; but for the time being, let us suspend any disbelief.

The standard technique for representing programs given a syntax of commands is to use *free monads*:

```
data FreeM f a = Var a | Com (f (FreeM f a))
```

For *FreeM f* to be a monad, we need f to be a functor:

```
instance Functor f ⇒ Monad (FreeM f) where
  return = Var
  Var a  >>= k = k a
  Com x >>= k = Com (fmap (>>=k) x)
```

Informally, the free monad for a functor f is given by trees with elements in the leaves and f-structures of children for internal nodes; the unit of the monad constructs a leaf, and the multiplication performs substitution. If f is suitably *Show*able, then so is *FreeM f*:

```
deriving instance (Show (f (FreeM f a)), Show a) ⇒
                  Show (FreeM f a)
```

The datatype *Command* has the right kind to be a functor, but because its type parameter is a phantom type, we cannot complete the definition—given a function f of type $r \to r'$, there is no general way to define *fmap f* :: *Command r* → *Command r'*. However, we can make *Command* into a functor if we combine it with a function from the phantom type (this can be seen as the embodiment of the dual of the Yoneda Lemma (Manzyuk 2013)):

```
data Action a = ∀r.Action (Command r, r → a)
instance Functor Action where
  fmap f (Action (c,k)) = Action (c,f · k)
```

Now we have an appropriate *Functor*, we define *Program*s in terms of the corresponding free monad:

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

We can now define single-step programs (what Plotkin and Power (2003) call *generic effects*) for each of the commands:

```
effect :: Command r → Program r
effect c = do {Com (Action (c, Var))}
say :: String → Program ()
say s = effect (Say s)
toast :: Int → Program ()
toast n = effect (Toast n)
sense :: Program Integer
sense = effect (Sense ())
```

and then we can write composite programs using **do** notation—both those in which requests are independent, like this straight-line program:

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

and those in which later requests depend on earlier responses, like this branching program:

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

But these programs are difficult to serialize and hence to distribute: a program like *branch*, with a *sense* command anywhere other than the final step, is in general infinitely branching. Put another way, there is no good way of defining a *show* function for *Action*s, and hence not for *Program*s either. However, if we restrict attention to *finitary* functors, we can—at least in principle—enumerate all the branches:

```
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
```

Here, the Haskell type class *Bounded* denotes types with a *minBound* and *maxBound*, and *Enum* denotes types with the *succ* function needed for the '..' notation.

We cannot support the infinitary *Sense* command that returns an *Integer*, because this type is not *Bounded*; but we could support a finitary sensing command, that returned a value from an enumeration instead:

```
data Temperature = Low | Medium | High
  deriving (Show, Read, Enum, Bounded)
data CommandF :: * → * where
  SayF   :: String → CommandF ()
  ToastF :: Int    → CommandF ()
  SenseF :: ()     → CommandF Temperature
```

We can now serialize a finitary program to send to the server; what should the server do when it receives it? First it should determine whether the program is a plain *Var* or a *Com*posite:

```
readFirstFreeM :: String → (Bool, String)
readFirstFreeM s = head $
  [(True, s')  | ("Var", s') ← lex s] ++
  [(False, s') | ("Com", s') ← lex s]
```

If the program is a *Var*, then the remaining input is the value to be returned. Otherwise, the server should read the first command; depending on what command this is, the server should read the appropriate type of argument, and execute that command. Finally, based on the result of the command, the server should pick the next branch to follow. In particular, for the *SenseF* command, there will be three continuation branches, one for each *Temperature*, and the server reads three serialized subprograms and picks one of them.

```
execProgram :: String → IO String
execProgram s = case readFirstFreeM s of
  (True, s') → return s'
  (False, s') → case readCommandF s' of
    (NSayF, s'')    → do (m, s''') ← readOne s''
                         execSay m
                         execProgram (head (read s'''))
    (NToastF, s'')  → do (n, s''') ← readOne s''
                         execToast n
                         execProgram (head (read s'''))
    (NSenseF, s'')  → do ((), s''') ← readOne s''
                         t ← execSenseF
                         execProgram (read s''' !! fromEnum t)
```

Here, *readOne* is a simple wrapper:

```
readOne :: Read a ⇒ String → IO (a, String)
readOne s = return (head (reads s))
```

(Note that the three *Com*posite cases of *execProgram* are really of the same form; for example, the *Say* command returns a () result, and *fromEnum* () = 0, and so that case could have been written instead

```
do (m, s''') ← readOne s''
   t ← execSay m
   execProgram (read s''' !! fromEnum t)
```

But we have stuck with the clearer and simpler but less regular version above.) Now we can run

```
programServer :: IO ()
programServer = server execProgram
```

on the server, and

```
programClient :: (Show a, Read a) ⇒ ProgramF a → IO a
programClient p = do {r ← client (show p); return (read r)}
```

on the client, and pass a whole program for remote execution.

Still, one might argue that it is a lot to expect of a toaster—even an internet-of-things toaster—for it to be able to parse nested serialized commands, select the appropriate branch, and so on. This brings us back to the question at the start of this section, and the discussion in the introduction: frameworks for batched requests typically insist that later requests should be independent of earlier responses; that is, that there should be no branching in a batch. The *ProgramF* type allows branching, so is too general. We show next how to enforce the no-branching constraint.

4. Free Applicative Functors

Free monads are the standard technique for capturing syntactic descriptions of programs over a particular signature of primitive actions. However, as we have seen, such programs are typically too general to be considered appropriate as batched requests for remote execution. In particular, it is arguably unreasonable to expect a mere toaster to be able to interpret a branching program. Moreover, branching programs can quickly become very large things to transmit, especially if individual commands have high out-degree. We already had to dispense with the infinitary *Sense* that forms a *Command Integer* in favour of the finitary *SenseF* that forms a *CommandF Temperature*; but even then, while an out-degree of 3 may be reasonable, an out-degree of 100 is almost certainly not.

These arguments all lead us to the conclusion that branching programs are inappropriate for batched requests, and that we should restrict attention to straight-line programs. It is fine for individual commands to return results, even results drawn from infinite types like *Integer*; but we should ensure that subsequent commands in the batched request are oblivious to those results. Instead, a sequence of requests is transmitted to the server, invoked in one after the other but independently, yielding a sequence of results that is returned to the client, and those two sequences are combined to match up individual requests with the corresponding results.

This independence of later steps on earlier results is precisely what characterizes the distinction between monads and *applicative functors* (McBride and Paterson 2008). So let us explore what happens if we try to batch up commands using free applicative functors rather than free monads. There are several equivalent definitions of free applicative functors (Capriotti and Kaposi 2014), of which we pick the following:

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

Informally, a value of type $FreeA\ f\ a_0$ is a right-nested sequence of the form $(fs_0, (fs_1, (..., (fs_{n-1}, x)...)))$, where $n \geqslant 0$, each element fs_i has type $f\ (a_{i+1} \to a_i)$, and the rightmost element x has type a_n. Note that the type variable b in the type of *More* is implicitly existentially quantified, as are each of the a_i (for $i > 0$) above.

Provided that f is a functor, that datatype can be given the structure of an applicative functor, as follows:

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

Now we no longer need to insist that result types of commands are bounded and enumerable, because we are no longer going to enumerate them—they need only be readable:

```
data ActionA a = ∀r.Read r ⇒ ActionA (Command r, r → a)
```

This *ActionA* type is straightforwardly a functor, as *Action* was,

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

and so we can assemble applicative programs from it:

```
type ProgramA a = FreeA ActionA a
```

We can make single-step programs for each of the commands, much as before:

```
effectA :: Read r ⇒ Command r → ProgramA r
effectA c = More (ActionA (c, λr () → r)) (Pure ())

sayA :: String → ProgramA ()
sayA s = effectA (Say s)

toastA :: Int → ProgramA ()
toastA n = effectA (Toast n)

senseA :: ProgramA Integer
senseA = effectA (Sense ())
```

Here's a straight-line program that senses the temperature, toasts for a bit, and senses the temperature again. Later commands are by construction independent of the results of earlier ones.

```
straightA :: ProgramA (Integer, Integer)
straightA = pure (λt () t' → (t,t')) ⊛ senseA ⊛ toastA 3 ⊛ senseA
```

We still can't serialize a whole program; in particular, we can't serialize arbitrary pure functions such as $\lambda t\,()\,t' \to (t,t')$ above. But crucially, now we don't need to—all we need to do is to serialize the commands, not the specific mechanism for collating replies:

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

If we have remembered the program, and retrieved a sequence of responses, we can essentially zip the two together to match up individual requests with their responses:

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

Now all our server needs to be able to do is to unpack a flat sequence of requests, run the requested commands in order, and pack up a flat sequence of responses:

```
execStraight :: String → IO String
execStraight s = do let reqs = read s
                    resps ← sequence (map execCommand reqs)
                    return (show resps)
```

This can be deployed via

```
straightServer :: IO ()
straightServer = server execStraight

straightClient :: ProgramA a → IO a
straightClient p = do r ← client (show (serializeA p))
                      return (deserializeA p (read r))
```

This behaviour is, we argue, not too much to expect of a toaster.

5. Remote Monads for Free

Let us compare our story to Gill et al. (2015)'s REMOTE MONAD design pattern, which inspired this paper. They do not use free monads or free applicative functors; instead, they build the batching mechanism from scratch. They distinguish between an asynchronous *Command*, which returns no result, and a synchronous *Procedure r*, which returns a result of type *r*; these datatypes together are analogous to our *Command r*, since we make no distinction between synchronous and asynchronous requests.

Their remote *Device* is essentially a *String* → *IO r* function, where the return type *r* is () for asynchronous commands and *String* for synchronous procedures; it is analogous to our *execCommand* function. Their individual *Command* and *Procedure* requests can be serialized and sent to a *Device* by their *send* function, analogous to our *commandClient*. Their 'strong remote monad' *Remote* is a combination of *ReaderT Device* for accessing a particular device, *StateT* [*Command*] for maintaining a buffer of unsent commands, and *IO* for actual remote communication. Individual *Command*s get appended to the buffer but not invoked immediately. An individual *Procedure* is combined with any buffered *Command*s into a *Packet*:

```
data Packet r = Packet [Command] (Procedure r)
```

and it is only *Packet*s that are sent across the wire. (There is a separate function to flush the buffer; this could have been avoided, if they had provided a *skip* :: *Procedure* () instead.)

Their datatype *Packet* is a simplification of the free applicative functor. It represents a non-empty sequence of requests, where the last is synchronous and the others all asynchronous; but it is built from scratch. By construction, later requests in a packet are independent of the results of earlier ones—indeed, the earlier ones have no results on which to depend. But this is a stronger constraint that necessary; as we have seen, it would be fine to allow intermediate requests to return results, so long as they do not affect later requests.

Gill et al. do introduce a 'strong remote applicative functor', motivated by the same observation that we make that "applicative functors are fundamentally better suited to remoteness than monads are: subsequent applicative computations cannot depend on the results of prior computations", allowing them to batch multiple *Procedure*s into a single *Packet*. However, they make essential use of lazy evaluation and recursive **do** notation in order to match up responses with their requests (their *send* function (Gill et al. 2015, §6) is cyclic). We show that these language features are unnecessary; our approach, with free applicative functors, will work just as well in a simple eager language as a sophisticated lazy one.

Neither we nor Gill et al. provide *implicit* batching of separate requests. In both cases, the programmer has to explicitly partition the program's requests into batches, and it is only the actual assembly of each group of requests into a single batch that happens more or less for free. Other approaches are similar in this respect; for example, Ibrahim et al. (2009) introduce a `batch` construct into Java in order to explicitly group requests into batches, and Cook (2010) declares that "although the result itself is elegant and useful, what is more significant is the realization that the original problems [of implicit batching] cannot be solved using existing programming language constructs and libraries. This work calls into question our assumption that general-purpose programming languages are truly general-purpose."

6. Conclusions

To summarize: remote procedure calls benefit from batching of independent requests in order to reduce round-trips; batched requests are essentially serializations of small programs; the traditional technique for serializing programs is to use free monads; free monads are really too liberal for this problem, because they accommodate (possibly infinitary) branching; free applicative functors, however, represent serialized straight-line programs; free applicative functors therefore have precisely the right restrictions for batching remote requests.

We have discussed the relationship with the REMOTE MONAD pattern (Gill et al. 2015); the novel contributions here are that *free* constructions provide the essence of program serialization, and that batching is fundamentally about *applicative functors* rather than monads. Marlow et al. (2014) present something like Gill et al.'s 'strong remote applicative functor', but they too build from first principles an analogue of the free applicative functor (their datatype *Fetch*). They also describe a proposed Haskell extension *ApplicativeDo* (Marlow 2015), whereby applicative computations can be written in the monadic **do** notation, and the concurrency inherent in independent requests can be reconstructed. This has just been incorporated into GHC (GHC 8.0), and it will be a very convenient provision—our applicative program *straightA* in Section 4 is much less clear than the similar monadic program *straight* in Section 3.

Acknowledgments

This work was partially supported by UK EPSRC grant number EP/K020919/1 on *A Theory of Least Change for Bidirectional Transformations*, and partially undertaken during a Visiting Profes-

sorship at the National Institute of Informatics in Tokyo, hosted by Zhenjiang Hu; I am very grateful for these sources of support. The idea itself was inspired by Gill et al. (2015), who kindly shared and discussed their code with me. I am also grateful to Matthew Pickering, who suggested that I look at free applicatives, to Michał Gajda, who pointed me to the example RPC code used in Section 2.2 and in the Appendix, and to Tim Zakian and Andy Gill and the anonymous reviewers for helpful comments.

References

P. Bogle and B. Liskov. Reducing cross domain call overhead using batched futures. In *Object-Oriented Programming: Systems, Languages and Applications*, pages 314–354. ACM, 1994. doi: 10.1145/191080.191133.

P. Capriotti and A. Kaposi. Free applicative functors. In P. Levy and N. Krishnaswami, editors, *Mathematically Structured Functional Programming*, volume 153 of *EPTCS*, pages 2–30, 2014. doi: 10.4204/EPTCS.153.2.

W. R. Cook. Breaking through to remote data and services. In S. Padmanabhuni, S. K. Aggarwal, and U. Bellur, editors, *India Software Engineering Conference*, pages 161–162. ACM, 2010. doi: 10.1145/1730874.1730877. Abstract of keynote talk.

L. P. Deutsch. The eight fallacies of distributed computing. `https://blogs.oracle.com/jag/resource/Fallacies.html`, 1994.

L. Erkök and J. Launchbury. Recursive monadic bindings. In *International Conference on Functional Programming*, pages 174–185. ACM, 2000. doi: 10.1145/351240.351257.

M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

GHC 8.0. *Glasgow Haskell Compiler Users' Guide, Version 8.0.1*, May 2016. `http://downloads.haskell.org/~ghc/8.0.1/docs/html/`.

A. Gill, N. Sculthorpe, J. Dawson, A. Eskilson, A. Farmer, M. Grebe, J. Rosenbluth, R. Scott, and J. Stanton. The Remote Monad design pattern. In B. Lippmeier, editor, *Haskell Symposium*, pages 59–70. ACM, 2015. doi: 10.1145/2804302.2804311.

R. Hinze. Fun with phantom types. In J. Gibbons and O. de Moor, editors, *The Fun of Programming*, Cornerstones in Computing, pages 245–262. Palgrave, 2003. ISBN 1-4039-0772-2.

A. Ibrahim, Y. Jiao, E. Tilevich, and W. R. Cook. Remote batch invocation for compositional object services. In S. Drossopoulou, editor, *European Conference on Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 595–617. Springer, 2009. doi: 10.1007/978-3-642-03013-0_27.

N. M. Josuttis. *SOA in Practice: The Art of Distributed System Design*. O'Reilly, 2007.

B. Liskov, T. Bloom, D. Gifford, R. Scheifler, and W. Weihl. Communication in the Mercury system. In *Hawaii International Conference on System Sciences*, pages 178–187. IEEE, 1988. doi: 10.1109/HICSS.1988.11804.

O. Manzyuk. Co-Yoneda Lemma. `https://oleksandrmanzyuk.wordpress.com/2013/01/18/co-yoneda-lemma/`, Jan. 2013.

S. Marlow. Applicative do-notation. `https://ghc.haskell.org/trac/ghc/wiki/ApplicativeDo`, Mar. 2015.

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. doi: 10.1145/2628136.2628144.

C. McBride and R. Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, 2008. doi: 10.1017/S0956796807006326.

G. Plotkin and J. Power. Algebraic operations and generic effects. *Applied Categorical Structures*, 11(1):69–94, 2003. doi: 10.1023/A:1023064908962.

J. Tibell, maintainer. Network.Socket.Bytestring documentation. `http://hackage.haskell.org/package/network-bytestring-0.1.3.4/docs/Network-Socket-ByteString.html`.

B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Types in Language Design and Implementation*, pages 53–66. ACM, 2012. doi: 10.1145/2103786.2103795.

A. Server and Client Wrappers

The server and client wrappers discussed in Section 2.2 are minor modifications of the example shown in the documentation for the *Network.Socket.ByteString* library (Tibell). To use these definitions, we need the following imports:

```
import Control.Monad (unless,forever)
import Network.Socket hiding (recv)
import Network.Socket.ByteString (recv,sendAll)
import qualified Data.ByteString as S
import qualified Data.ByteString.Char8 as C
```

The server is set up to listen on port 3000:

```
server :: ServerBehaviour → IO ()
server f = withSocketsDo $ do
  (addr: _) ← getAddrInfo
    (Just (defaultHints {addrFlags = [AI_PASSIVE]}))
    Nothing (Just "3000")
  s ← socket (addrFamily addr) Stream defaultProtocol
  bindSocket s (addrAddress addr)
  listen s 1
  forever $ do
    (conn, _) ← accept s
    talk f conn
    sClose conn
```

where *talk f c* repeatedly reads up to 4096 bytes from connection *c*, converts the corresponding bytestring to a string, applies *f* to it, and sends the result back, until it reads null:

```
talk :: (String → IO String) → Socket → IO ()
talk f conn =
do req ← recv conn 4096
  unless (S.null req) $ do
    resp ← f (C.unpack req)
    sendAll conn (C.pack resp)
    talk f conn
```

And for the client, we have

```
client :: String → IO String
client request = withSocketsDo $ do
  (addr: _) ← getAddrInfo
    Nothing (Just "127.0.0.1") (Just "3000")
  s ← socket (addrFamily addr) Stream defaultProtocol
  connect s (addrAddress addr)
  sendAll s (C.pack request)
  response ← recv s 4096
  sClose s
  return (C.unpack response)
```

The hard-wired IP address (here, for demonstration purposes, the loopback address `"127.0.0.1"`) and port `"3000"` in *client* are for the machine on which the server is running and the port on which it is listening.