MapReduce as a Monad

by Julian Porter ⟨julian.porter@porternet.org⟩

June 4, 2011

MapReduce is an increasingly popular paradigm for building massively parallel applications. It originated in ideas from functional programming, but soon migrated to a more imperative approach. In the process, popular implementations have become massively complicated to run. For example, HADOOP requires considerable detailed knowledge on the part of the application's author to ensure that their code plugs properly into the MapReduce framework.

I show how MapReduce can be implemented as a form of monad in Haskell. Why a monad? First, it means processing code is purely about processing; it need not know anything about the MapReduce framework. Second, MapReduce reduces to repeated composition of monadic functions with $\gg=$*, so the programmer can trivially mix MapReduce with other paradigms. Third, it allows a massive generalisation of MapReduce. And fourth, it's an interesting exercise in monadic programming.*

About MapReduce

MapReduce consists of two stages, named **map** and **reduce**. In the map stage, the following happens:

1. A central unit takes data in the form of key/value pairs and divides it into a number of chunks, which it distributes to one or more **mappers**.
2. Each mapper transforms its input list of key/value pairs into an output list of key/value pairs.

In the reduce stage, the following happens:

1. A central unit sorts the output of the map stage by key, dividing it into chunks, one for each unique key value in the set, and distributes the chunks to **reducers**.
2. Each chunk is sent to a reducer, which transforms its input key/value pairs into an output list of key/value pairs.

This process is repeated until the desired result is achieved.

The mapper and reducer can be many to many functions, so one input pair may correspond to no output pairs or to many. Also, the type of the output from these functions need not be the same as that of the input.

In practice, there is an intermediate stage called **partitioning**, which is a second reduce stage. Therefore, anything we say about reduce applies to it as well. Note that as my approach removes the distinction between map and reduce, partitioning fits in naturally as just another such step.

The idea

Transformer functions

Consider reduce. Reduce collects data from the map stage and then partitions it into a number of sets, one per key value. Then the reducer is applied to each set separately, producing new value / key pairs read for the next step. Restating this, what happens is that for each key value k, reduce extracts the pairs with key value equal to k and then applies the reducer to them:

$$
\begin{aligned}
&\mathit{reduceF} :: (\mathit{Eq}\ b) \Rightarrow b \rightarrow ([s'] \rightarrow [(s'', c)]) \rightarrow ([(s', b)] \rightarrow [(s'', c)]) \\
&\mathit{reduceF}\ k\ \mathit{reducer} = (\lambda ss \rightarrow \mathit{reducer}\ \$\ \mathit{partition}\ k\ ss) \\
&\quad \mathbf{where} \\
&\quad \mathit{partition}\ k\ ss = \mathit{fst}\ \langle\$\rangle\ \mathit{filter}\ (\lambda(_, k') \rightarrow k \equiv k')\ ss
\end{aligned}
$$

The same is true of map. In map value / key pairs are distributed randomly across mappers, which then transform their chunk of the data to produce value / key pairs. This is just *reduceF* with random key values.

Therefore map and reduce can be fitted within the same conceptual framework, and we can treat the mapper and reducer as instances of a more general concept, the **transformer**, which is a function which takes a list of values and transforms them to produce a list of value / key pairs:

$$[s] \rightarrow [(s', a)]$$

Generalising transformers

There is no reason why *reduceF* (and its equivalent for map) should take the specific form set out above of a wrapper around a transformer. For a given *tranformer* we can define:

$$
\begin{aligned}
&\mathit{reduceF'} :: (\mathit{Eq}\ a) \Rightarrow a \rightarrow ([(s, a)] \rightarrow [(s', b)]) \\
&\mathit{reduceF'} = (\mathit{flip}\ \mathit{reduceF})\ \mathit{transformer}
\end{aligned}
$$

But now the obvious next step is to generalise from *reduceF* to any function

$$\mathit{transformF} :: a \to ([(s, a)] \to [(s', b)])$$

so we generalise from a function which selects values and feeds them to the transformer, to a general function which combines the selection and transformation operations.

Now consider what happens when we compose successive stages of MapReduce. As we have just seen, the outcome of each stage is a map from value / key pairs to value / key pairs. Then in applying the next stage, we take *transformF* and combine it with that map to obtain a new map from value / key pairs to value / key pairs:

$$([(s, a)] \to [(s', b)]) \to (b \to ([(s', b)] \to [(s'', c)])) \\ \quad \to ([(s, a)] \to [(s'', c)])$$

This is highly suggestive of the rule for monadic bind:

$$(\mathit{Monad}\ m) \Rightarrow m\ a \to (a \to m\ b) \to m\ b$$

where m is a function of type $([(s, a)] \to [(s', b)])$.[1] Say I define a type

$$\textbf{newtype}\ m\ s\ a\ s'\ b = \mathit{MR}\ ([(s, a)] \to [(s', b)])$$

Then the composition of MapReduce stages becomes:

$$m\ s\ a\ s'\ b \to (b \to m\ s'\ b\ s''\ c) \to m\ s\ a\ s''\ c$$

where s, s', s'' are data types and a, b, c are key types. This certainly looks very much like a "monad" of some description.

Using a monad

The goal is then for a single round of generalised MapReduce to be expressible as:

$$\mathit{state} \ggg= \mathit{mapF} \ggg= \mathit{reduceF}$$

where *mapF* and *reduceF* are the generalised transformers and state is a monad containing the initial state. This approach must contain MapReduce as a special case. Therefore there should be a function

$$\mathit{wrapMR} :: (\mathit{Eq}\ a) \Rightarrow ([s] \to [(s', b)]) \to (a \to m\ s\ a\ s'\ b)$$

that wraps a transformer in the monad so that MapReduce can be expressed as:

$$\circ \circ \circ \ggg= \mathit{wrapMR}\ \mathit{mapper} \ggg= \mathit{wrapMR}\ \mathit{reducer} \ggg= \circ \circ \circ$$

[1] In fact, this is very similar to the definition of the *State* monad, which is a function of type $(s \to (s, a))$. The main differences are that the value type changes on application of the function, and also that we need to keep track of the types of the keys.

Aside: do we need all those indices?

Having four indices on m certainly seems rather clumsy. Is there anything we can do to reduce them? For reference, I write out again the bind rule:

$$m\ s\ a\ s'\ b \rightarrow (b \rightarrow m\ s'\ b\ s''\ c) \rightarrow m\ s\ a\ s''\ c$$

Note that indices are paired, so objects can be bound if the second value / key pair of the first object equals the first value / key pair of the second.

Therefore, the obvious simplification is to treat value / key pairs as opaque types, so we can rewrite as

$$m\ p\ q \rightarrow (q \rightarrow m\ q\ r) \rightarrow m\ p\ r$$

which is (almost) a monad. There are two problems which turn out to be related:

1. The first argument of the middle function is b, not (s', b), and so we had to generalise this to q.
2. If we write out the simplified version in terms of the functions that comprise m we get:

   $$([p] \rightarrow [q]) \rightarrow (q \rightarrow ([q] \rightarrow [r])) \rightarrow ([p] \rightarrow [r])$$

   which has lost sight of the distinction between values and keys

The first problem is survivable, but together with the second it becomes insurmountable. The critical fact about MapReduce is that we take the data set and then, for each key, select a transformer apply it to the whole data set. But now we cannot do that, as all we can see is the opaque value / key pair, and selecting transformers based on key *and* value loses the element of aggregation critical to MapReduce.

There are various (rather ugly) ways we could get round this, e.g.

```
class Keyable k v | k → v where
  key :: k → v
class Monad' m p q where
  return :: q → m p q
  (>>=) :: (Keyable q v) ⇒ m p q → (v → m q r) → m p r
```

However, this is clearly just reintroducing the key value by the back door, and is also rather confusing, so we will, regrettably, stick with the four-index version.

Implementation

The Monad' and MapReduce types

The monad is defined as:

```
class Monad' m where
  return :: a → m s x s a
  (>>=) :: (Eq b) ⇒ m s a s' b → (b → m s' b s'' c) → m s a s'' c
```

The generalised transformers are of a type which is an instance of *Monad'*:

```
newtype MapReduce s a s' b = MR {runMR :: ([(s, a)] → [(s', b)])}
```

The monadic operations

The return operator is fairly trivial: *return x* just forces the key value of the output to be x. Bind is the critical part of the design:

```
1  bindMR :: (Eq b) ⇒ MapReduce s a s' b → (b → MapReduce s' b s'' c)
2     → MapReduce s a s'' c
3  bindMR f g = MR (λs →
4    let
5      fs = runMR f s
6      gs = map g $ nub $ snd <$> fs
7    in
8    concat $ map (λg' → runMR g' fs) gs)
```

Note that the set of key values is extracted from the output of f in line 5 and used, via g, to generate a set of transformers in line 6: one per key value. Each of these functions is given the whole data set output by f. It is left to the functions themselves to determine which parts of the data to process. This enables varying strategies:

1. MapReduce, where one function is applied to sets based on key values
2. Key-dependent functions are applied to all of the data
3. A mix of the two (possibly even within one processing chain)

The crucial factor for making this parallel is the *map* in line 8. It is here that multiple instances of the transformer are created, one per key value. Therefore in practical implementations, map will be replaced with some parallel processing mechanism.

Relation to MapReduce

I said above that a method was needed for wrapping a traditional transformer in the monad. The following code does just that, turning a transformer to a monadic function:

$$\begin{aligned}&\mathit{wrapMR} :: (\mathit{Eq}\ a) \Rightarrow ([s] \rightarrow [(s', b)]) \rightarrow (a \rightarrow \mathit{MapReduce}\ s\ a\ s'\ b)\\&\mathit{wrapMR}\ f = (\lambda k \rightarrow \mathit{MR}\ (g\ k))\\&\quad \textbf{where}\\&\quad g\ k\ ss = f\ \$\ \mathit{fst}\ \langle\$\rangle\ \mathit{filter}\ (\lambda s \rightarrow k \equiv \mathit{snd}\ s)\ ss\end{aligned}$$

Note that the transformer need know nothing about the monad. It is simply a function that maps value/key pair to value/key pairs. All the business of selecting value/key pairs for the attention of the transformer is done by *wrapMR*.

Limitations

Bind sends the entire dataset to every thread of execution. This may be desirable in cases where generalised MapReduce is being used, but for standard MapReduce with large datasets on many processors, this may be an issue. Possible approaches involve use of shared memory / disk to store the datasets, or a more carefully engineered implementation of *wrapMR*.

Code

Library

This is an implementation of generalised MapReduce that is multithreaded on multi-processor system with a single OS image. The library consists of three modules, a hashing function for data, a parallel generalisation of *map* and the MapReduce monad. Clearly more complex approaches to parallelisation could be used, for example coordinating processes running on a cluster of processing nodes.

Application

This is an implementation of the classic first MapReduce program: a word-count routine. Note that the MapReduce part of the application (lines 14–17) requires only two lines of active code. Also, the transformers are pure data processing; they are completely unaware of the generalised MapReduce framework. Finally, observe the use of *distributeMR* to spread the initial data across multiple mappers. Clearly we have established an extremely elegant architecture in which transformations can be plugged into a parallel processing chain with no difficulty.

Performance

The application has been tested on a heavily loaded dual core MacBook Pro on datasets of up to 1,000,000 words. Specifically, the application was tested varying

```
1  {-# LANGUAGE TypeSynonymInstances #-}
2  module Hashable (Hashable, hash) where
3  import qualified Crypto.Hash.MD5 as H
4  import Data.Char (ord)
5  import Data.ByteString (ByteString, pack, unpack)
6    -- any type that can convert to ByteString is hashable
7  class Hashable s where
8    conv :: s → ByteString
9    -- the hash function
10 hash :: (Hashable s) ⇒ s → Int
11 hash s = sum $ map fromIntegral (unpack h)
12   where
13   h = H.hash $ conv s
14   -- make String Hashable
15 instance Hashable String where
16   conv s = pack $ map (fromIntegral ∘ ord) s
```

Listing 1: The Hash Function Module

```
1  module ParallelMap (map) where
2  import Control.Parallel.Strategies (parMap, rdeepseq)
3  import Control.DeepSeq (NFData)
4  import Prelude hiding (map)
5  map :: (NFData b) ⇒ (a → b) → [ a ] → [ b ]
6  map = parMap rdeepseq
```

Listing 2: The Parallel map Module

```
1  module MapReduce (MapReduce, return, (>>=), runMapReduce, distributeMR, wrapMR)
2  import Data.List (nub)
3  import Control.Applicative ((<$>))
4  import Control.Monad (liftM)
5  import Control.DeepSeq (NFData)
6  import IO
7  import Prelude hiding (return, (>>=))
8  import Hashable (Hashable, hash)
9  import qualified ParallelMap as P
10   -- the generalised Monad
11 class Monad' m where
12   return :: a -> m s x s a
13   (>>=) :: (Eq b, NFData s'', NFData c) => m s a s' b -> (b -> m s' b s'' c) -> m s a s'' c
14   -- the generalised MapReduce data type
15 newtype MapReduce s a s' b = MR {runMR :: ([(s, a)] -> [(s', b)])}
16   -- the generalised MapReduce type as an instance of Monad'
17 instance Monad' MapReduce where
18   return = retMR
19   (>>=) = bindMR
20 retMR :: a -> MapReduce s x s a
21 retMR k = MR (\ss -> [(s, k) | s <- fst <$> ss])
22 bindMR :: (Eq b, NFData s'', NFData c) => MapReduce s a s' b
23    -> (b -> MapReduce s' b s'' c) -> MapReduce s a s'' c
24 bindMR f g = MR (\s ->
25   let
26     fs = runMR f s
27     gs = map g $ nub $ snd <$> fs
28   in
29   concat $ P.map (\g' -> runMR g' fs) gs)
30   -- execute a MapReduce Monad' given specified initial state
31 runMapReduce :: MapReduce s () s' b -> [s] -> [(s', b)]
32 runMapReduce m ss = (runMR m) [(s, ()) | s <- ss]
33   -- distribute a data set across a range of key values
34 distributeMR :: (Hashable s) => MapReduce s () s Int
35 distributeMR = MR (\ss -> [(s, hash s) | s <- fst <$> ss])
36   -- wrap a transformer in a monadic function
37 wrapMR :: (Eq a) => ([s] -> [(s', b)]) -> (a -> MapReduce s a s' b)
38 wrapMR f = (\k -> MR (g k))
39   where
40   g k ss = f $ fst <$> filter (\s -> k == snd s) ss
```

Listing 3: The MapReduce Module

```
1  module Main where
2  import MapReduce
3  import IO
4  import System.Environment (getArgs)
5  import Prelude hiding (return, (>>=))
6  import qualified Prelude as P
7  main :: IO ()
8  main = do
9     args ← getArgs
10    state ← getLines (head args)
11    let res = mapReduce state
12    putStrLn $ show res
13    -- perform MapReduce
14 mapReduce :: [String] → [(String, Int)]
15 mapReduce state = runMapReduce mr state
16    where
17    mr = distributeMR >>= wrapMR mapper >>= wrapMR reducer
18    -- transformers
19 mapper :: [String] → [(String, String)]
20 mapper [] = []
21 mapper (x : xs) = parse x ++ mapper xs
22    where
23    parse x = map (λw → (w, w)) $ words x
24 reducer :: [String] → [(String, Int)]
25 reducer [] = []
26 reducer xs = [(head xs, length xs)]
27    -- get input
28 getLines :: FilePath → IO [String]
29 getLines file = do
30    h ← openFile file ReadMode
31    text ← hGetContents h
32    P.return (lines text)
```

Listing 4: Word count MapReduce application

the following parameters:

1. Number of words: 10000, 100000, 250000, 500000, 1000000
2. Size of vocabulary: 500, 1000, 2000
3. Number of cores used: 1, 2

As can be seen from figure 1, on a single core, performance scaled approximately linearly with corpus size at a rate of approximately 100,000 words per second. On two cores performance was noticeably worse, and was distinctly non-linear, also depending on the vocabulary size.

I suspect the poor performance on two cores was due to the data passing issue discussed above, with bus and memory conflicts coming to the fore. Note, however that *this code has no optimisations* having been written for clarity rather than performance. Therefore it is expected that considerably better results could be obtained with very little effort.

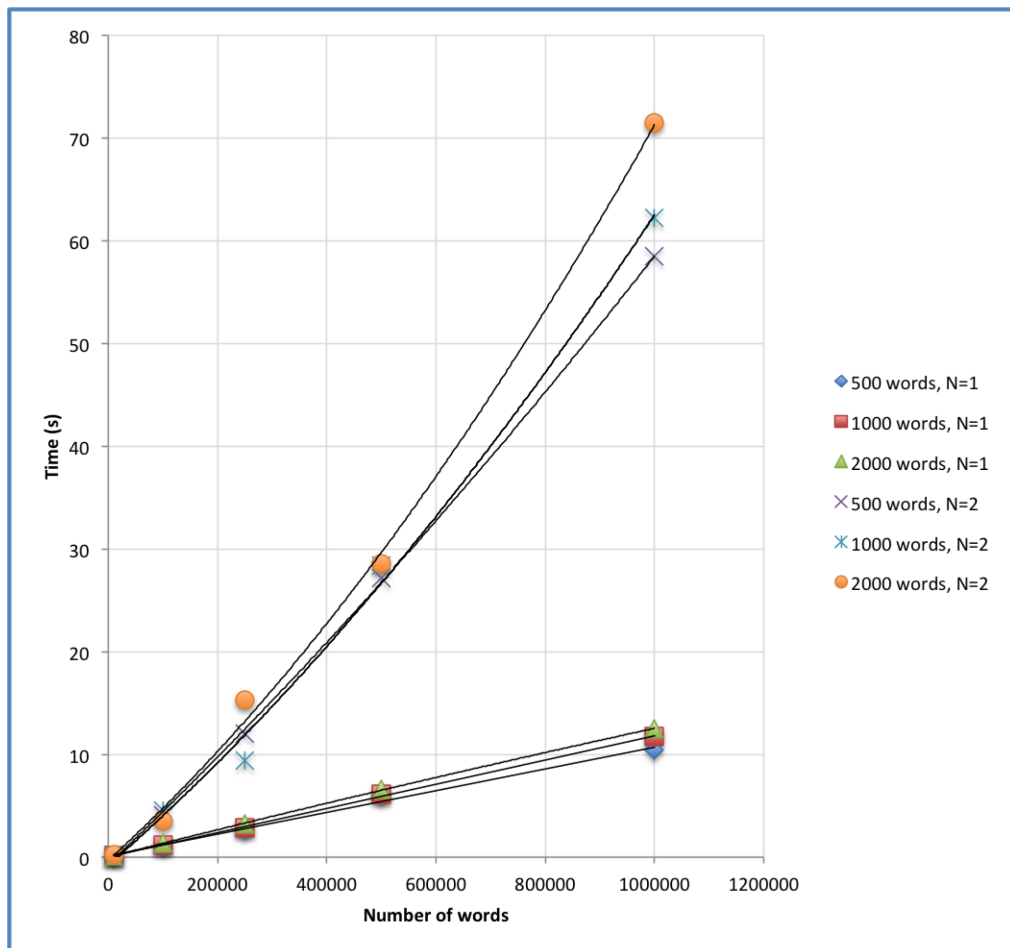

Figure 1: Performance of the application