

# The MapReduce type of a Monad

Julian Porter  
julian.porter@porternet.org

## Abstract

MapReduce ([1]) is an increasingly popular paradigm for distributed computing. It consists of stages in which lists of key / data pairs are sorted on the key value and then each chunk is passed to a transformation function. The resulting key / data pairs are accumulated and passed to the next phase.

In this paper we build on [2], where we showed that MapReduce can be implemented as a kind of monad, with  $\gg=$  corresponding to the composition of processing steps, and demonstrated a simple application (word count). Here we show how this can be generalised from operating on lists to acting on data in *any* monad. We conclude by discussing the interpretation of this generalisation.

## 1 Notation

1. All code examples are written in Haskell.
2. We use stereotyped notation for types:
  - (a) Throughout we will refer to paired **data types** ( $x, y$  and  $z$ ) and **key types** ( $a, b$  and  $c$ ) so that the natural pairing is  $a \leftrightarrow x, b \leftrightarrow y, c \leftrightarrow z$ . We assume as a matter of course that all key types are instances of the type class *Eq*.
  - (b)  $t, u$  and  $v$  are general types with no further assumptions made.
  - (c)  $m$  is a monadic type.
3. When we speak of a *monad* we shall allow for a type  $m$  multiple index types, and a natural generalisation of the bind rule, e.g.

$$(\gg=) :: m\ t\ u \rightarrow (u \rightarrow m\ u\ v) \rightarrow m\ t\ v$$

4. When we speak of a **quasi-monad** we shall refer to a type on which it is possible to define the usual monadic operations, but not all of the monad laws (see [3]) need be satisfied.

## 2 The idea

### 2.1 MapReduce in a nutshell

We start with a brief summary of the MapReduce algorithm (for more details see [1]). MapReduce involves two concepts:

1. Functions that I will call **transformers**, which have type  $[x] \rightarrow [(b, y)]$ .
2. A framework consisting of repeated application of a **stage** process which takes a transformer and wraps it to produce a function of type  $[(x, a)] \rightarrow [(y, b)]$ .

The stage, which can be construed as a function  $wrap :: (Eq\ a) \Rightarrow ([x] \rightarrow [(y, b)]) \rightarrow [(x, a)] \rightarrow [(y, b)]$  as follows:

1. It constructs a list  $ks$  of the unique key values from the input list  $kds$ .

2. It splits  $kds$  into a collection of lists  $vs$ , one per key value in  $ks$ , each one consisting of the data values from those pairs in  $kds$  with the given key value.
3. It pushes each of the lists in  $vs$  separately through the transformer and concatenates the results.

In a **map** stage the key values in the input list are assigned randomly. In a **reduce** stage they are assigned deterministically by the preceding stage. Clearly both map and reduce are specialisations of one more general concept.

## 2.2 Generalising transformers

Given  $trans :: ([x] \rightarrow [(y, b)])$  then  $wrap\ trans :: (Eq\ a) \Rightarrow a \rightarrow ([ (a, x) ] \rightarrow [(b, y)])$ . We can generalise this to an arbitrary function  $a \rightarrow ([ (x, a) ] \rightarrow [(b, y)])$  which combines the transformer and the wrapper; call this a **generalised transformer**.

Now consider composing stages. Each stage maps key / data pairs to more key / data pairs: . Composing this with the function above gives:

$$([ (a, x) ] \rightarrow [(b, y)]) \rightarrow (b \rightarrow ([ (b, y) ] \rightarrow [(c, z)])) \rightarrow ([ (a, x) ] \rightarrow [(c, z)])$$

which is highly suggestive of a monadic bind where the monad wraps a function of type  $([ (a, x) ] \rightarrow [(b, y)])$ .

## 2.3 Motivation

The goal is for generalised MapReduce to be expressible using a monad  $MapReduce\ x\ a\ y\ b$  as:

$$\dots \xrightarrow{MR} map \xrightarrow{MR} reduce \xrightarrow{MR} \dots$$

where  $map$  and  $reduce$  are generalised transformers. We want to include MapReduce as a special case; so there should be a function

$$wrap' :: (Eq\ a) \Rightarrow ([x] \rightarrow [(b, y)]) \rightarrow (a \rightarrow MapReduce\ x\ a\ y\ b)$$

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

$$\dots \xrightarrow{MR} wrap'\ mapper \xrightarrow{MR} wrap'\ reducer \xrightarrow{MR} \dots$$

## 3 A monadic approach

### 3.1 The MapReduce type

$MapReduce$  a quasi-monadic generalisation of the  $State$  monad:

$$\mathbf{newtype}\ MapReduce\ x\ a\ y\ b = MR\ \{ run :: ([ (x, a) ] \rightarrow [(y, b)]) \}$$

### 3.2 The monadic operations for *MapReduce*

**Definition 3.2.1.** *The monadic operations are:*

```

1 return :: b → MapReduce x a x b
2 return k = MR (map (first $ const k))
3 ( $\ggg^{MR}$ ) :: (Eq b) ⇒ MapReduce x a y b → (b → MapReduce y b z c) → MapReduce x a z c
4 ( $\ggg^{MR}$ ) f g = MR (λkds →
5   let
6     fs = run f kds
7     gs = map g $ nub $ fst ($) fs
8   in
9   concatMap (λx → run x fs) gs)

```

So *return* forces all the keys to the given value. In  $\ggg^{MR}$  the set of key values is extracted from the output of *f* in line 7 and used, via *g*, to generate transformers in line 9. Each of these functions is given the whole output of *f*; they determine which parts of the data to process.

### 3.3 Wrapping transformers

We required in section 2.3 that there should a way to wrap a transformer to produce a monadic function. The following function does just that:

```

wrap' :: (Eq a) ⇒ ([x] → [(b,y)]) → (a → MapReduce x a y b)
wrap' f = (λk → MR (g k))
  where
    g k kds = f $ snd ($) filter (λkd → k ≡ fst kd) kds

```

As can be seen it selects those key / data records in *kds* whose key is *k*, extracts their data values and then pushes the resulting list through the transformer *f*. This makes the *MapReduce* framework transparent to application writers.

### 3.4 Comments

If we think of *g* as a family of functions, then in  $f \ggg^{MR} g$  each of these functions is given the entire output by *f*. It is left to the functions themselves to determine which (if any) parts of the data to process. This enables varied strategies:

- Standard MapReduce, where each function filters the data based on its key value.
- A generalised filtering strategy with a more complex filtering predicate.
- General transformers that depend explicitly on the key as well as the data.
- A mixture of the above, varying from stage to stage.

The crucial factor for making this parallel is the *map* in line 9 of the listing in definition 3.2.1 which creates one instance of the generalised transformer per key. In practical implementations, *map* will be replaced with some parallel processing mechanism.

## 4 MapReduce as a monad transformer

Observe that  $[]$  is a monad. It turns out that *MapReduce* can be further generalised as a monad transformer.

### 4.1 The *MapReduceT* type

**Definition 4.1.1.** Let  $m$  be a monad; the *MapReduceT* type of  $m$  is the type:

$$\text{newtype } (\text{Monad } m) \Rightarrow \text{MapReduceT } m \ t \ u = \text{MR } \{ \text{run} :: m \ t \rightarrow m \ u \}$$

We can treat instances of *MapReduceT*  $m$  as an arrow:

**Definition 4.1.2.** Define the operations:

$$\begin{aligned} (\gg) &:: (\text{Monad } m) \Rightarrow \text{MapReduceT } m \ t \ u \rightarrow \text{MapReduceT } m \ u \ v \\ &\quad \rightarrow \text{MapReduceT } m \ t \ v \\ (\gg) f g &= \text{MR } (\lambda ss \rightarrow \text{run } g \$ \text{run } f ss) \\ (\lhd) &:: (\text{Monad } m) \Rightarrow \text{MapReduceT } m \ t \ u \rightarrow m \ t \rightarrow m \ u \\ (\lhd) f x &= \text{run } f x \end{aligned}$$

There is also a useful lifting operation:

**Definition 4.1.3.** Define the lifting operation:

$$\begin{aligned} \text{lift} &:: (\text{Monad } m) \Rightarrow m \ t \rightarrow \text{MapReduceT } m \ t \ t \\ \text{lift } x &= \text{MR } (\text{const } x) \end{aligned}$$

### 4.2 The *MapReduceT* monad transformer

To make *MapReduceT* a monad transformer we make *MapReduceT*  $m$  a quasi-monad with a *lift* operation.

**Definition 4.2.1.** If  $m$  is a monad then *MapReduceT*  $m$  is made into a quasi-monad by the operations

$$\begin{aligned} 1 \text{ return} &:: (\text{Monad } m) \Rightarrow t \rightarrow \text{MapReduceT } m \ t \ t \\ 2 \text{ return } x &= \text{lift } (\text{return } x) \\ 3 \text{ bind} &:: (\text{Monad } m) \Rightarrow \text{MapReduceT } m \ u \ u \rightarrow \text{MapReduceT } m \ t \ u \\ 4 &\quad \rightarrow (u \rightarrow \text{MapReduceT } m \ u \ v) \rightarrow \text{MapReduceT } m \ t \ v \\ 5 \text{ bind } p f g &= \text{MR } (\lambda xs \rightarrow ps \ xs \gg\! = \! \gg gs \ xs) \\ 6 &\quad \mathbf{where} \\ 7 &\quad ps \ xs = (f \gg\! = \! \gg p) \lhd xs \\ 8 &\quad gs \ xs \ x = (f \gg\! = \! \gg g \ x) \lhd xs \end{aligned}$$

where  $\gg\! = \! \gg$  is in  $m$ . If  $p :: \text{MapReduceT } m \ u \ u$  is idempotent ( $p \gg\! = \! \gg p \cong p$ ) define  $(\gg\! = \! \gg_p) = \text{bind } p$ .

The parameter  $p$  in  $\gg\! = \! \gg_p$  allows for filtering of the key / data pairs before the generation of transformer functions in line 7. It will turn out (see theorem 4.3.2) to be the equivalent of  $\text{nub} \$ \text{snd} \$$  in the definition of  $\overset{MR}{\gg\! = \! \gg}$ .

**Definition 4.2.2.** If  $m$  is a monad and *MapReduceT*  $m$  is its *MapReduce* quasi-monad, then make it a monad transformer using *lift* as defined in 4.1.3.

### 4.3 Equivalence to *MapReduce*

**Definition 4.3.1.** *Define*

$$\begin{aligned} \text{dedupe} &:: (Eq\ a) \Rightarrow \text{MapReduceT}\ []\ (a, x)\ (a, x) \\ \text{dedupe} &= \text{MR}\ (\text{nubBy}\ (\lambda kd\ kd' \rightarrow \text{fst}\ kd \equiv \text{fst}\ kd')) \end{aligned}$$

Clearly *dedupe* takes a list of key / data pairs and returns a list with one pair per unique key value (the data value is ill-defined but irrelevant).<sup>1</sup> Note also that *dedupe* is trivially idempotent.

If  $t = (a, x)$ ,  $u = (b, y)$   $v = (c, z)$ , observe that  $\text{MapReduce}\ x\ a\ y\ b \cong \text{MapReduceT}\ []\ t\ u$ , so we shall elide the distinction between the two types in the following.

**Theorem 4.3.2.** *If  $f :: \text{MapReduce}\ x\ a\ y\ b$  and  $g :: (b \rightarrow \text{MapReduce}\ y\ b\ z\ c)$  then*

$$(\ggg^{MR})f\ g \cong f \ggg_{\text{dedupe}} (g \circ \text{fst})$$

*Proof.* Rewrite definition 3.2.1 as:

$$\begin{aligned} (\ggg^{MR})f\ g &= \text{MR}\ (\lambda xs \rightarrow \\ &\text{let} \\ &\quad fs = f \frown xs \\ &\quad gs = \text{map}\ (g \circ \text{fst})\ \$\ \text{dedupe} \frown fs \\ &\text{in} \\ &\quad \text{concat}\ \$\ \text{map}\ (\lambda y \rightarrow y \frown fs)\ gs) \end{aligned}$$

Rewriting this some more we get:

$$\begin{aligned} (\ggg^{MR})f\ g &= \text{MR}\ (\lambda xs \rightarrow \\ &\text{let} \\ &\quad fs = f \frown xs \\ &\quad ks = \text{dedupe} \frown fs \\ &\text{in} \\ &\quad \text{concatMap}\ (\lambda k \rightarrow (g \circ \text{fst})\ k \frown fs)\ ks) \end{aligned}$$

But in the list monad  $xs \ggg f = \text{concatMap}\ f\ xs$  so we can rewrite this as:

$$\begin{aligned} (\ggg^{MR})f\ g &= \text{MR}\ (\lambda xs \rightarrow \\ &\text{let} \\ &\quad bs\ ys = f \ggg \text{dedupe} \frown ys \\ &\quad hs\ ys\ z = f \ggg (g \circ \text{fst})\ z \frown ys \\ &\text{in} \\ &\quad bs\ xs \ggg hs\ xs) \end{aligned}$$

where we have used the trivial identity  $n \frown (m \frown l) \cong (m \ggg n) \frown l$ . □

<sup>1</sup>To be really rigorous we could amend the definition of *dedupe* to read

$$\begin{aligned} \text{dedupe} &:: (Eq\ a) \Rightarrow \text{MapReduceT}\ []\ (a, \text{Maybe}\ x)\ (a, \text{Maybe}\ x) \\ \text{dedupe} &= \text{MR}\ (\lambda vks \rightarrow \text{second}\ (\text{const}\ \text{Nothing})\ \$)\ (\text{nubBy}\ (\lambda kd\ kd' \rightarrow \text{fst}\ kd \equiv \text{fst}\ kd'))\ vks) \end{aligned}$$

which makes the output of *dedupe* determinate at the price of hugely complicating the statement and proof of theorem 4.3.2.

## 4.4 Discussion

### 4.4.1 Relation to the *State monad*

There is an obvious similarity between *MapReduceT* and the state monad, in that members of *MapReduceT m* are functions  $m\ a \rightarrow m\ b$  while members of *State s* are functions  $s \rightarrow (s, a)$ . In fact this similarity runs very deep, and it turns out that the state monad is a special case of MapReduce.

**Definition 4.4.1.** *Define*

**data**  $Hom\ a\ b = H\ \{run :: (a \rightarrow b)\}$

and make *Hom* a monad by taking

$return\ x = H\ (const\ x)$   
 $f \ggg^H g = H\ (\lambda x \rightarrow g'\ (f'\ x)\ x)$

where  $f' \equiv run\ f$  and  $g'\ x\ y \equiv run\ (g\ x)\ y$ .

Clearly we can make this type into a category by defining

$(\ggg) :: Hom\ a\ b \rightarrow Hom\ b\ c \rightarrow Hom\ a\ c$   
 $(\ggg)\ f\ g = H\ \$\ (run\ g) \circ (run\ f)$   
 $id :: Hom\ a\ a$   
 $id = H\ id$

**Lemma 4.4.2.** *There is a natural map  $hom :: Hom\ a\ b \rightarrow MapReduceT\ Hom\ s\ a\ b$  such that*

$(hom\ q) \ggg_p (hom\ r) \equiv hom\ \$\ (q \ggg p) \ggg^H (q \ggg r)$

*Proof.* The map is given by

$hom :: Hom\ a\ b \rightarrow MapReduceT\ Hom\ s\ a\ b$   
 $hom\ (H\ f) = MR\ (\lambda h \rightarrow f \circ h)$

The result now follows trivially from the definitions of  $\ggg^H$  and  $\ggg_p$ . □

Now say  $f :: State\ s\ a$ ,  $g :: a \rightarrow State\ s\ b$  and define  $m :: s \rightarrow (s, a)$ ,  $n :: a \rightarrow s \rightarrow (s, b)$  by  $f = state\ m$ ,  $g\ x = state\ (n\ x)$ . Then in the state monad

$f \ggg^S g \equiv state\ \$\ (\lambda s \rightarrow n\ (snd\ \$\ m\ s)\ (fst\ \$\ m\ s))$   
 $\equiv state\ \$\ run\ \$\ q \ggg^H r$

where

$q = H\ (snd \circ m) :: Hom\ s\ a$   
 $r\ x\ y = H\ (n'\ x\ \$\ m\ y) :: a \rightarrow Hom\ s\ (s, b)$

where  $n'\ x\ y \equiv n\ x\ \$\ fst\ y$ . Now pick a distinguished value  $e :: s$  (if  $s$  is a monoid we can take  $e = empty$ , but the value is irrelevant). Define

$$\begin{aligned}
p &:: \text{Hom } (s, a) \text{ } (s, a) \\
p \ x &= H \ (e, \text{snd } x) \\
q' &:: \text{Hom } (s, a) \text{ } (s, a) \\
q' &= H \ \$ \ m \ \circ \ \text{fst} \\
r' &:: (s, a) \rightarrow \text{Hom } (s, a) \text{ } (s, b) \\
r' \ x \ y &= H \ n' \ (\text{snd } x) \ y
\end{aligned}$$

then

$$\begin{aligned}
(q' \ggg p) \ggg^H (q' \ggg r') &\equiv H \ n' \ (\text{snd} \ \circ \ m \ \circ \ \text{fst}) \ (m \ \circ \ \text{fst}) \\
&\equiv (H \ \text{fst}) \ \ggg \ (q \ggg^H r)
\end{aligned}$$

But now lemma 4.4.2 lets us write

$$\begin{aligned}
p &:: \text{MapReduceT } (\text{Hom } s) \ (s, a) \ (s, a) \\
q' &:: \text{MapReduceT } (\text{Hom } s) \ (s, a) \ (s, a) \\
r' &:: (s, a) \rightarrow \text{MapReduceT } (\text{Hom } s) \ (s, a) \ (s, b)
\end{aligned}$$

and gives

$$\begin{aligned}
q' \ggg_p r' &\equiv \text{hom } \$ \ (q' \ggg p) \ \ggg^H \ (q' \ggg r') \\
&\equiv \text{hom } \$ \ (H \ \text{fst}) \ \ggg \ \text{bind}
\end{aligned}$$

where we showed above that  $f \ggg^S g \equiv \text{state } \$ \ \text{run } \text{bind}$ . Therefore we have proved:

**Proposition 4.4.3.** *State  $s$  is isomorphic to  $\text{MapReduceT } (\text{Hom } s)$ .*

□

#### 4.4.2 General properties of $\text{MapReduceT } m$

What is the significance of the general monad  $\text{MapReduceT } m$ ? Recall from definition 4.2.1:

$$\begin{aligned}
\text{bind } p \ f \ g &= MR \ (\lambda \ x \ s \rightarrow \text{bs } \ x \ s \ \ggg \ g \ s \ x \ s) \\
&\mathbf{where} \\
\text{bs } \ x \ s &= f \ \ggg \ p \ \lrcorner \ x \ s \\
g \ s \ x \ s \ x &= f \ \ggg \ g \ x \ \lrcorner \ x \ s
\end{aligned}$$

We consider the two functions that sit on either side of the  $\ggg$ . If we take  $f = MR \ f'$  and  $p = MR \ p'$  then  $(\lambda \ x \ s \rightarrow f \ \ggg \ p \ \lrcorner \ x \ s)$  becomes

$$\begin{aligned}
\text{left} &:: m \ a \rightarrow m \ b \\
\text{left } \ x \ s &= p' \ \$ \ f' \ x \ s
\end{aligned}$$

so we could say that it pushes data through  $f' :: m \ a \rightarrow m \ b$  and then projects the result onto the image of  $p' :: m \ b \rightarrow m \ b$ . Noting that  $p'$  is idempotent iff  $p$  is, this is a genuine projection. We therefore write  $p'$  as  $\text{Proj}_p$ .

On the right hand side, defining  $g' :: b \rightarrow m \ b \rightarrow m \ c$  by  $MR \ (g' \ x) = g \ x$ , then  $(\lambda \ x \ s \ y \rightarrow f \ \ggg \ g \ y \ \lrcorner \ x \ s)$  becomes

$$\begin{aligned}
\text{right} &:: b \rightarrow m \ a \rightarrow m \ c \\
\text{right } \ y \ x \ s &= g' \ y \ \$ \ f' \ x \ s
\end{aligned}$$

Putting these together,  $f \ggg_p g$  can be written as

$$MR (\lambda xs \rightarrow (\text{Proj}_p \$f' xs) \ggg (\lambda y \rightarrow g' y \$f' xs))$$

We can rewrite this as

$$f \ggg_p g \cong f \ggg MR (\text{Proj}_p \ggg g)$$

where

$$\begin{aligned} (\ggg) &:: \text{MapReduceT } m \ t \ u \rightarrow (u \rightarrow \text{MapReduceT } m \ t \ v) \rightarrow \text{MapReduceT } m \ t \ v \\ (\ggg) f g &= MR (\lambda xs \rightarrow (f \text{--} xs) \ggg (\lambda y \rightarrow g y \text{--} xs)) \end{aligned}$$

So we modify  $g'$  by binding it to the projection onto  $p$  in the underlying monad and then we push data through  $f$  and on into *both* sides of the bind. Alternately, the projected output of  $f$  selects which of the functions parameterised by  $g$  its raw output is pushed through, the precise nature of the selection depending on the monad  $m$ .

This is a little opaque, but it gives a very clear understanding, if any more is needed, of why this formulation is the natural generalisation of MapReduce.

## References

- [1] R Lämmel. “Google’s MapReduce Programming Model—Revisited”. In: *Science of Computer Programming* 70(1) (2008), pp. 1–30.
- [2] J Porter. “MapReduce as a Monad”. In: *The Monad Reader* 18 (2011), pp. 5–16. URL: <http://themonadreader.files.wordpress.com/2011/07/issue18.pdf>.
- [3] P Wadler. “Monads and Composable Continuations”. In: *LISP and Symbolic Computation* 7 (1993), pp. 39–56.