The Monad.Reader Issue 20

by Tillmann Vogt ⟨tillk.vogt@googlemail.com⟩
and Jan Stolarek ⟨jan.stolarek@p.lodz.pl⟩
and Julian Porter ⟨julian.porter@porternet.org⟩

August 25, 2012

Edward Z. Yang, editor.

Contents

Editorial

by Edward Z. Yang ⟨ezyang@mit.edu⟩

It's not dead, it's resting!

Awakening from its slumber, The Monad Reader shambles back to life with three new articles for your perusal. The authors in this issue have my gratefulness for putting up with what ended up being a very long editing cycle, and you, fair reader, have my thanks for waiting patiently. Enjoy!

Enumeration of Tuples with Hyperplanes

by Tillmann Vogt ⟨tillk.vogt@googlemail.com⟩

Enumeration means assigning an integer value to a structure or, looking at the Haskell Prelude, defining an Enum instance. There are Enum instances for basic types like Bool or Char. But why are there no Enum instances for tuples? Given that the elements of a tuple are enumerable, the combination ought to be enumerable too. Unfortunately, it takes too long to e.g. enumerate all 7-tuples of Chars to find "Haskell". Another problem is that the classic method (imagine a counter ticking upwards) makes the digits increase with different speed and is therefore unusable for combinatorial problems where you want a quick variation of every digit. The solution that people most of the time chose in such a situation is a random function. The problem with a random function (apart from not being side effect free) is that you never have the confidence of having examined at least a defined part of the search space exhaustively. Using an enumeration as we present here, you can halt the processing e.g. in a proving system or a genetic algorithm, see how long it took to compute, add constraints, and continue with the assurance of not having missed a solution. In this article, we will describe a method how to enumerate tuples in a fair manner that could be used in such cases.

Cartesian product of small sets

The most straightforward way to enumerate tuples is to generate all combinations of elements of (finite) sets like this:

```
cartProd (set:sets) = let cp = cartProd sets
                      in [x:xs | x <- set, xs <- cp]
cartProd [] = [[]]
```

This is a Cartesian product in maths, which can be found in the library haskell-for-maths. The exact way how the sets are enumerated does not matter because usually every element is used.

To understand what this function does, let's look at an example:

```
cartProd [[0..3],[0..3],[0..3]],
```

evaluates to:

```
[[0,0,0],
 [0,0,1],
 [0,0,2],
 [0,0,3],
 [0,1,0],
 [0,1,1],
 [0,1,2], ...].
```

One can see that this is counting of numbers with base 4. This is fine if every combination has to be examined and the sets are small. If the sets become too big there is no way to go through all combinations. In that case, this enumeration is problematic for several reasons:

1. It is not able to enumerate an infinite set at one of its digits. For example,
   `cartProd [[0..3],[0..3],repeat 0]`
   will only produce lists of [0,0,0] because it is stuck in the third list that contains infinitely many 0s.
2. It does not distribute the changes fairly among the digits. The above enumeration is a lexicographic ordering that changes the last digit in the tuple all the time and the first rarely. In the next subsection we will see an enumeration (repeated diagonalization) that solves the first point but not this point.
3. Every set has to be of the same type, because it is a list. This limitation could be easily changed by transforming the lists into tuples and maybe do a lookup with the integer values in a table .

Enumerating $\mathbb{Q}$

There is another way to enumerate tuples that every computer scientist learns when showing that the rational numbers are enumerable. A rational number can be enumerated by giving an axis of natural numbers to each numerator and denominator. Then an enumeration is a path consisting of increasingly longer diagonals (Figure 1). For a way to avoid duplicates in this enumeration (like 2/2) see [1].

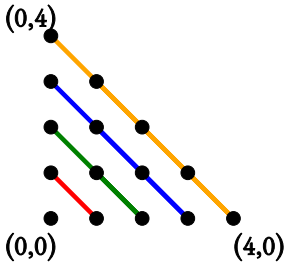


Figure 1: Enumeration of 2-tuples

The next question, typically raised in an exercise course, is how to enumerate 3-tuples. While everybody is thinking hard about a path through three dimensional space, somebody suggests to use the 2-tuples as a new axis and do the same diagonalization again. This gets the job done, because it allows infinite sets in the digits (here $\mathbb{N}$), but is unfair: there are many more 2-tuples than single values ($O(x^2)$ against $O(x)$, x being the size of each set at a digit). The result is an unfair distribution (see the left picture of Figure 2 that shows the path that is biased towards one axis). To make matters worse, doing this trick repeatedly results in distributions where one digit grows with $O(x^3)$ for 4-tuples, $O(x^4)$ for 5-tuples.

This unfairness can be avoided in 4-tuples, generally 2^n-tuples, and at least be made less severe by choosing tuples for both axes, but it still leaves the non-2^n-tuples with an unfair distribution.

Diagonals are Hyperplanes

If you look again at the diagonals of the enumeration of 2-tuples, you can observe that a diagonal (*e.g.* $[(0,4),(1,3),(2,2),(3,1),(4,0)]$) always consists of tuples whose sum of digits is constant: $[(x,y) \mid x+y=c], c \in \mathbb{N}$. The same idea can be applied to 3-tuples. The right picture of Figure 2 shows an enumeration path where the sum of the digits stay constant for an enumeration plane:

$$[(x,y,z) \mid x+y+z=c], c \in \mathbb{N}.$$

The idea can be generally stated as grouping tuples by the distance to the origin with a norm. Then these tuples can be enumerated by arranging them monotonically increasing. Instead of planes, one could imagine to use sphere surfaces, though it is probably much harder to calculate the n-th tuple in this case (and this is necessary as we will see later to make a proper Enum instance).

The enumeration with planes in the right picture of Figure 2 generates tuples with a fair distribution of changes in the digits. It is especially fair in the beginning.

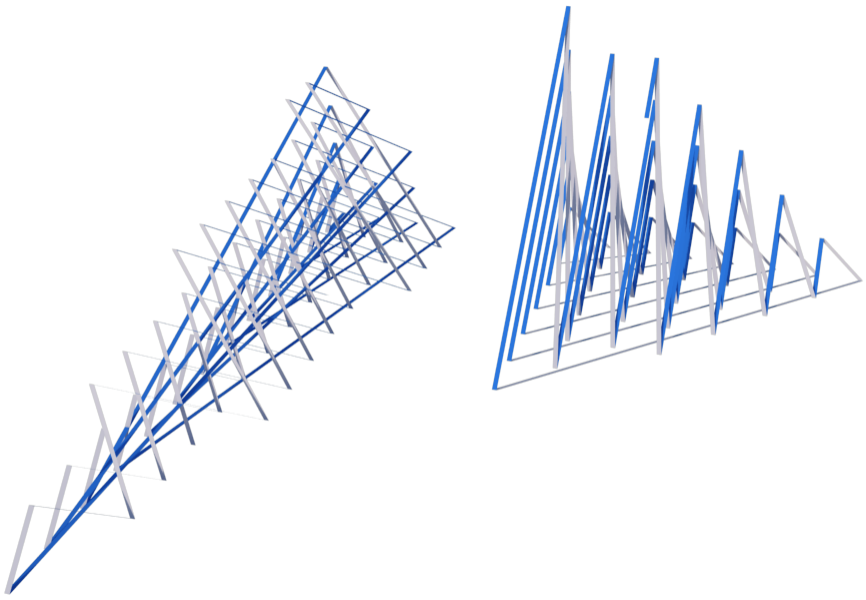


Figure 2: Enumeration with repeated diagonalization (left), with hyperplanes (right)

An enumeration of n-tuples switches every digit once in the first n+1 values. For 3 tuples it is for example:

```
[(0,0,0), (1,0,0), (0,1,0), (0,0,1), ...]
```

For higher values it becomes unfairer, though other solutions (like hyperspheres) may have much more complicated algorithms and formulas. While the 3D enumeration uses planes, in higher dimensions hyperplanes have to be used. In fact, diagonals in 2D are hyperplanes, since hyperplanes are a $(n-1)$-dimensional subset of a n-dimensional space that divides the space in two.

Enum Instances

The first experiments of this enumeration algorithm generated a list of tuples of natural numbers. Ideally, we would like to replace natural numbers with arbitrary types as long as they can be enumerated. That of course also includes tuples itself. Haskell has a very nice feature called typeclass that permit the nesting of arbitrary tuples and values whose type have an `Enum` instance. Take for example the `Enum` instance for 2-tuples:

```
instance (Enum a, Enum b, Eq a, Eq b, Bounded a, Bounded b)
                    => Enum (a, b) where
```

If `a` is enumerable and `b` is enumerable then the tuple `(a,b)` is also enumerable. Because the values are enumerated from the low boundary to the high boundary and comparisons have to be made, `a` and `b` are also in `Eq` and `Bounded`. The goal is now to be able to write

```
( enumFrom (0,(1,2),3) ) :: [(Word8,(Word8,Word8),Word8)]
```

This is an example for an arbitrary nesting, with arbitrary starting values, that evaluates to (you will see later why):

```
[(0,(1,2),3), (0,(2,1),4), (0,(3,0),5), ...]
```

A type that is enumerable can be made an instance of the `Enum` typeclass by defining at least two functions:

```
toEnum            :: Int -> a
```

which returns a value of type `a` to a number and

```
fromEnum          :: a -> Int
```

which returns a number to a value of type `a`. There exist instances for primitive types like Char, Bool and Int. The following functions are defined with `toEnum` and `fromEnum`, but can be overridden:

```
succ, pred        :: a -> a
enumFrom          :: a -> [a]             -- [n..]
enumFromThen      :: a -> a -> [a]        -- [n,n'..]
enumFromTo        :: a -> a -> [a]        -- [n..m]
enumFromThenTo    :: a -> a -> a -> [a]   -- [n,n'..m]
```

In our case, we will overwrite `succ` (the successor in the enumeration) and `pred` (the predecessor), because they can be calculated more quickly than the default implementation:

```
succ = toEnum . (`plusInt` oneInt)  . fromEnum
```

succ, pred

In order to understand the following code, it's worth keeping the image Figure 2 in your head. Changes happen when a 0 is reached in the tuple. Because enumeration of arbitrary types should be possible, 0 is replaced by minBound. Instead of looking at a tuple it is faster to look only at the important digits:

```
succ (a,b,c) =
 if c == minBound then
  if b == minBound then
   if a == minBound then (succ a, b     , c     ) -- (0,0,0)
                    else (pred a, succ b, c     ) -- (a,0,0)
                   else  (a     , pred b, succ c) -- (a,b,0)
                  else
 if b == minBound then -- switching to the next hyperplane
 if a == minBound then (toEnum (fc+1), minBound, minBound) -- (0,0,c)
                  else (pred a  , toEnum (fc+1), minBound) -- (a,0,c)
                  else (a       , pred b       , succ c )  -- diagonals
  where
    fc = fromEnum c
```

The last tuple generates diagonals and every `else` in the first half of this code generates a higher dimensional hyperplane. The line `(toEnum (fc+1), minBound, minBound)` makes a switch to a new hyperplane, that is one bigger in the sum of digits than the last hyperplane. Because the types in the tuple can be different from each other, `fromEnum c` and `toEnum (fc+1)` have to be used instead of `succ` and `pred`. The function `pred` for 3-tuples is implemented similarly:

```
  pred (x,y,z) =
   if z == minBound then
    if y == minBound then
     if x == minBound then
       error "Enum.pred{(x,y,z)}: tried to take ‘pred’ of minBound"
     else (minBound, minBound, toEnum (fx-1)) -- (fy,fz) was (0,0)
    else  (succ x  , minBound, toEnum (fy-1)) --     fz  was    0
   else   (x       , succ y  , pred z       )
    where
      fx = fromEnum x
      fy = fromEnum y
```

Here the line `(minBound, minBound, toEnum (fx-1))` makes a switch to a lower hyperplane.

Avoiding some typewriting

The function `succ` could be defined for the other tuples in the same way, but this is a lot of typing for 15-tuples (the biggest tuple for which there exist typeclass instances in the base libraries of Haskell). There is a pattern in the tuple that is produced when a certain pattern of zero or non-zero values in an inbound tuple occurs. One can come up with functions that do the same as the upper `succ` function when seeing an n-tuple as a list. But a tuple can have all kinds of types and therefore cannot be encoded as ordinary

lists. What to use? A heterogeneous list? One can also use a trick that transforms a tuple into a finite list-like structure and back:

```
to5Tuple  ((((a,b),c),d),e)  = (a,b,c,d,e)
```

Then `succ` can be defined like this:

```
succ fz s ((x,y),z)
 | y /= minBound && z == minBound = ((x,pred y), succ z)
 | y == minBound && z == minBound = (succ fz False (x,y), z)
 | y /= minBound && z /= minBound = ((x,pred y), if s then succ z else toEnum (fz+1))
 | y == minBound && z /= minBound = (succ fz False (x,toEnum fz), minBound)
```

Here `y` and `z` should be imagined as single values while `x` can be the list-like structure. Unfortunately the upper code gives a type error in Haskell: "Occurs check: cannot construct the infinite type: t0 = (t0, a0)." This is understandable, since the compiler can't know that this nesting terminates without a termination analyzer. In the expression `((a,b),c)` the value `a` must not stand for another tuple *e.g.* `(((d,e),b),c)`; the compiler does not know that we are not constructing an infinite type. We can hack our way around this by defining `succ` for all tuples, like this:

```
succ5 :: ( Enum a, Enum b, Enum c, Eq a, Eq b, Eq c,
           Bounded a, Bounded b, Bounded c) =>
         Int -> Bool -> ((((a,b),c),d),e) -> ((((a,b),c),d),e)
succ5 fz s ((x,y),z)
| y /= minBound && z == minBound = ((x,pred y), succ z)
| y == minBound && z == minBound =  (succ4 fz False (x,y), z)
| y /= minBound && z /= minBound =((x,pred y), if s then succ z else toEnum (fz+1))
| y == minBound && z /= minBound =  (succ4 fz False (x,toEnum fz), minBound)
```

To define the other `succ` functions one only has to change the number after the two `succ`-functions in the body accordingly.

I haven't found an easy way to do the same with `pred`, because the last value in the tuple depends on values at various positions.

The size of enumeration hyperplanes

The sizes of the enumeration hyperplanes are needed in order to assign numbers to tuples (`toEnum`) or to find the place of a tuple in an enumeration (`fromEnum`). From the right picture of Figure 2, it can be seen that the 2D-hyperplanes consist of 1D-hyperplanes (diagonals). Generally an n-dimensional space is enumerated with $(n-1)$-dimensional hyperplanes. These hyperplanes are again enumerated by $(n-2)$-dimensional hyperplanes, and so on. The size of an n-dimensional enumeration hyperplane can be deduced by looking at the two and three dimensional case.

The size of our 2D-hyperplane (a plane of diagonals) is a well known problem that Gauß solved at the age of nine:

$$\begin{aligned} &1+ \\ &(1+1)+ \\ &(1+1+1)+... \\ &1+2+3+4+5...+n=\sum_{k=1}^{n} k = n(n+1)/2 \end{aligned} \tag{1}$$

A 3D-hyperplane consists of increasingly larger 2D-hyperplanes:

$$\begin{aligned} &1+ \\ &(1+2)+ \\ &(1+2+3)+... \\ &=\sum_{k=1}^{n} k(k+1)/2 \\ &=\sum_{k=1}^{n}(\frac{k}{2}+\frac{k^2}{2}) = \frac{n(n+1)}{2*2}+\frac{2n^3+3n^2+n}{6*2} \end{aligned} \tag{2}$$

In (2), we sum over the polynomial of the Gauß sum (1).
To calculate the sum of k^2's, the Bernoulli formula for sums of powers was used:

$$\boxed{\sum_{k=1}^{n-1} k^p = \frac{1}{p+1}\sum_{j=0}^{p}\binom{p+1}{j}\beta_j n^{p+1-j}} \tag{3}$$

Applying this pattern again leads to the size of a 4D-hyperplane:

$$\begin{aligned} &1+ \\ &(1+(1+2))+ \\ &(1+(1+2)+(1+2+3))+... = \sum_{k=1}^{n}(\frac{k(k+1)}{2*2}+\frac{2k^3+3k^2+k}{6*2}) \end{aligned} \tag{4}$$

Calculating it

The last section showed that the size of a hyperplane can be given with a polynomial. The common way to represent a polynomial is to only use its coefficients. Therefore a function is needed to evaluate a polynomial in n:

```
polynomial :: Int -> [Rational] -> Rational
polynomial n coeffs = foldr (+) 0 (zipWith nPowerP coeffs [1..])
  where nPowerP a_j p = a_j * (fromIntegral (n^p))
```

The coefficients of a polynomial in n that results from $\sum_{k=1}^{n-1} k^p$ are needed. At position n^{p+1-j} this is:

$$\text{coefficient}(j) = \frac{1}{p+1}\binom{p+1}{j}\beta_j. \tag{5}$$

The `combinat` library, which can be found on Hackage, provides some parts of the Bernoulli formula: the Bernoulli numbers β_j and the binomial coefficient. With this, a part of the Bernoulli formula can be implemented by:

```
sumOfPowers :: Int -> [Rational]
sumOfPowers p = reverse [bin j * ber j / fromIntegral p + 1 | j <- [0..p]]
  where bin j = fromIntegral (binomial (p+1) j)
        ber j | j == 1 = negate (bernoulli j) -- see wikipedia entry
              | otherwise = bernoulli j
```

Because of the $-j$ in n^{p+1-j}, a `reverse` is needed. We apply this formula to a polynomial with increasing degree: first k, then $\frac{k}{2} + \frac{k^2}{2}$, ...

The size of an n-dimensional hyperplane can be calculated by repeatedly applying the Bernoulli formula (`map sumOfPowers`) to the n^p after the coefficients of a polynomial and adding the coefficients (`merge`):

```
hyperplaneSize :: Int -> Int -> Int
hyperplaneSize dim n = round (genPolynom 1 [1])
  where genPolynom :: Int -> [Rational] -> Rational
        genPolynom d coeffs | d == dim  = polynomial n coeffs
                            | otherwise = genPolynom (d+1)
                              (merge coeffs (map sumOfPowers [1..(length coeffs)]))

merge coeffs ls = foldr myZip [] multiplied_ls
  where multiplied_ls = zipWith (\c l -> map (c*) l) coeffs ls
        myZip (l0:l0s) (l1:l1s) = (l0+l1) : (myZip l0s l1s)
        myZip a b = a ++ b
```

fromEnum

The function `fromEnum` takes a tuple and gives back an integer. If we want to calculate *e.g.* `fromEnum (True,'a',2)` we first apply `fromEnum` to every argument:

```
fromEnum (fromEnum True, fromEnum 'a', fromEnum 2)
```

The tuple now only consists of integers (here `fromEnum (1,97,2)`). Using `fromEnum` at every digit of the tuple, a general function `fe` can be constructed that transforms an arbitrary sized list of integers:

```
fromEnum (a,b,c) = fe [fromEnum a, fromEnum b, fromEnum c]
```

To understand how `fe` works, recall that a hyperplane consists of all tuples whose sum of digits is constant. For the upper 3-tuple there is one 2D-plane that is located at the sum of the three integers. All the hyperplanes below this one have to be summed up. To calculate a tuple's position inside the 1D-hyperplane (diagonal) of this plane, the plane is projected to a lower dimension by setting one coordinate to zero. Instead of planes in 3D, we now have diagonals in 2D. The sum of the tuple values of this smaller tuple give us again a hyperplane and the hyperplane-sizes below have to be added to the overall sum. This process ends with a position inside of a 1D-line.

The decision of which dimension to set to zero is a degree of freedom, but the choice must be made consistently in `toEnum`, `succ` and `pred`. There are two ways to do this in 2D, three ways in 3D, generally n ways in an n-dimensional space.

In the last section, we saw how to calculate the size of the nth enumeration hyperplane of a d-dimensional space. The function `fe` uses lists of these sizes and an increasingly larger sum of hyperplanes:

```
ssizes d = [ sum (take n sizes) | n <- [1..] ]
  where sizes = [ hyperplaneSize d i | i <- [0..] ]

summedSizes :: Int -> Int -> Int
summedSizes dim n = (ssizes dim) !! n
```

For the upper 3-tuple, we always set the first value to zero, and thus calculate:

```
(summedSizes 2       (a1+b1+c1) ) +     -- a1 = fromEnum a
(summedSizes 1          (b1+c1) ) +     -- b1 = fromEnum b
                            c1           -- c1 = fromEnum c
```

For all dimensions:

```
fe [x] = x
fe (x:xs) = ( summedSizes (length xs) (foldr (+) 0 (x:xs)) ) + (fe xs)
```

toEnum

To calculate `toEnum`, we have to invert the upper process. If `toEnum` gets the argument `n` and the tuple has dimension d, then we search for where n is located between the sums of hyperplanes by summarizing the smaller hyperplanes until `n` is reached and continue with the lower dimensions. But, instead of values for each tuple position, we get sums of subtuple values. To see how to extract values at each position from the sums, look again at the example of a 3-tuple: To get `a1`, it is enough to know the two sums `(a1+b1+c1)` and `(b1+c1)` and calculate the difference:

```
differences :: [Int] -> [Int]
differences [x] = [x]
differences (x:y:ys) = (x-y) : (differences (y:ys))
```

But first, we calculate each of these sums (like the one belonging to `a1+b1+c1`), put them in the list `planes` and also remember the `rest` for the next round:

```
hplanes :: Int -> ([Int],Int) -> ([Int],Int)
hplanes d (planes,rest) = ((fst hp):planes, snd hp)
  where hp = hyperplane d rest

hyperplane dim n = ( (length s) - 1, n - (if null s then 0 else last s) )
  where s = filterAndStop [ summedSizes (dim-1) i | i <- [0..] ]
        filterAndStop (x:xs) | x <= n     = x : (filterAndStop xs)
                             | otherwise = []
```

`te` uses the upper functions and works for an arbitrary dimensioned tuple:

```
te :: Int -> Int -> [Int]
te dim n = differences $ reverse $ fst $ foldr hplanes ([],n) [1..dim]
```

Reaching boundaries

Until now only tuples with large sets were enumerated where it takes a long time to reach boundaries. But if tuples with Bools are enumerated (or with only four elements like in figure 3), boundaries are quickly reached. Notice that in figure 3 the diagonals are enumerated from bottom right to top left and only half of the last line is subtracted. For non-quadratic rectangles it is also easier to subtract triangular shapes than to calculate the red area (especially in higher dimensions as we see later).

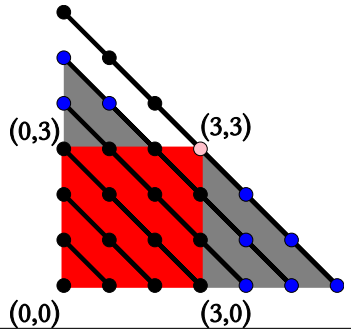


Figure 3: Reaching boundaries in two dimensions

Figure 4 shows boundaries in 3D where the enumeration planes in 3D form a tetrahedron starting at the origin. From this base tetrahedron, we must subtract three

tetrahedra translated by the size of the boundary in the respective dimension (also try to imagine a book shape instead of a cube in the picture). The intersections of these three tetrahedra make it necessary to add again three smaller tetrahedra that start at two added boundaries. In the former sections, we could see a pattern from the 2D to the 3D case and deduce a general pattern. There seems to be no obvious pattern in this case, as we don't know how hypertetrahedra intersect in higher dimensions.

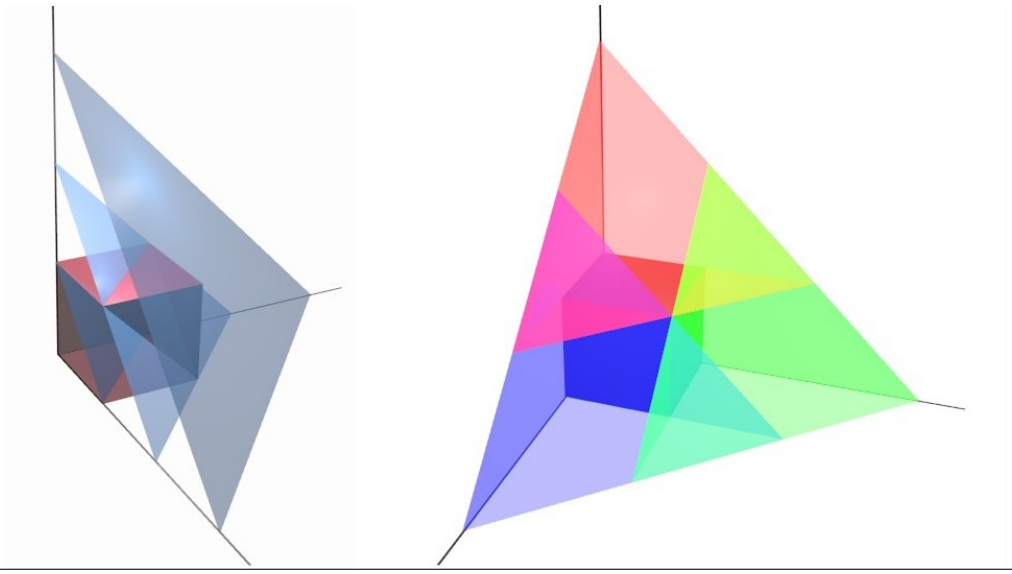


Figure 4: Left: Enumeration planes that intersect a bounded space. Right: The tetrahedra that have to be subtracted (red, green, blue) and added again (violet, yellow, blue-green)

Intersections of hypertetrahedra

To get an idea of how many intersections there are we generate a 5D-hypertetrahedron (`all5s` is an enumeration from (0,0,0,0,0)):

```
set5 :: Word8 -> [(Word8,Word8,Word8,Word8,Word8)]
set5 s = [ (a,b,c,d,e) | (a,b,c,d,e) <- take 5000 all5s, a+b+c+d+e <= s ]
set510 = set5 10
```

Then we translate this hypertetrahedron by the same length in each dimensions respectively and look at all combinations of intersections. The library haskell-for-maths gives us a function that generates all subsets of size k:

```
combinationsOf 0 _ = [[]]
combinationsOf _ [] = []
```

```
combinationsOf k (x:xs) =
  map (x:) (combinationsOf (k-1) xs) ++ combinationsOf k xs
```

The following function returns sets of indices to hypertetrahedra where the index specifies in which dimension they were translated.

```
intersectionSets = concat $ map (\k -> combinationsOf k [0..4] ) [2..5]
```

has the value

```
[[0,1],[0,2],[0,3],[0,4],[1,2],[1,3],[1,4],[2,3],[2,4],[3,4],
[0,1,2],[0,1,3],[0,1,4],[0,2,3],[0,2,4],[0,3,4],[1,2,3],[1,2,4],
[1,3,4],[2,3,4],[0,1,2,3],[0,1,2,4],[0,1,3,4],[0,2,3,4],
[1,2,3,4],[0,1,2,3,4]].
```

If we do an `intersectionTest` with

```
intersectionTest =
  filter (\set -> (length (intersection set)) > 1) intersectionSets
intersection set = intersectAll (map translated set)
intersectAll set = foldr intersect (head set) (tail set)
translated dim = map (add5 (tr5 dim 5)) set510
  where add5 (a,b,c,d,e) (f,g,h,i,j) = (a+f,b+g,c+h,d+i,e+j)
```

we get $\binom{5}{2} = 10$ intersections:

```
[[0,1],[0,2],[0,3],[0,4],[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
```

In higher dimensions it seems that all intersections of 2 hypertetrahedra exist, but three or more hypertetrahedra never intersect at more than one point.

fromEnum with boundaries

In the bounded case, `fromEnum` is calculated nearly in the same way as in the previous sections. Looking again at Figure 4, there is a top hyperplane below which the other hyperplanes are summed up. They form the base hypertetrahedron from which `d` tetrahedra (as many as the tuple has dimensions) have to be subtracted (`sum (map intersection comb)`) and because they also intersect between each other hypertetrahedra have to be added again in an alternating fashion (`addOrSub`):

```
feb :: [(Int,Int)] -> Bool -> Int
feb [(x,b)] _ = x
feb xs start  = sum (map intersection comb)
              + recursion
  where
    recursion | start = feb xs False
```

```
          | d > 0 = feb (tail xs) False
          | otherwise = 0
    ys = map fst xs
    bs = map snd xs
    l = (length ys) - 1
    d | start = l
      | otherwise = l-1
    s  = foldr (+) 0 ys
    s2 = foldr (+) 0 (tail ys)
    comb :: [[Int]]
    comb = concatMap (\x -> combinationsOf x [0..l]) range
    range :: [Int]
    range | start     = [0..2]
          | otherwise = [0..l]
    intersection ts = addOrSub * tbs
      where
        tbs | start     = tbs' (s -sumBs)
            | elem 0 ts = tbs' (s3-sumSpecial)
            | otherwise = tbs' (s2-sumBs)
        tbs' delta = if delta > 0 then summedSizes d delta else 0
        sumBs | null ts = 0
              | otherwise = sum (map (+1) (map (bs!!) ts))
        sumSpecial = sum (map tetra ts)
        tetra 0 = 0 -- because the special tetrahedron starts at 0
        tetra i = (bs!!i)+1
        -- size of special tetrahedron (begins a zero)
        specialTetra = s - (head bs)
        s3 | s2 <= specialTetra = s2
           | otherwise          = specialTetra
        addOrSub | even (length ts) = 1
                 | otherwise        = -1
```

In the upper code we have to introduce the bool argument `start` to cope with a strange effect. Although our experiment showed that there are only intersections of at most two hypertetrahedra, this is only true in the beginning. Once we project the tuple to one dimension below, there are more intersections. Consider this example: in four dimensions we have to subtract four hypertetrahedra. These hypertetrahedra are the original 4D enumeration, translated in each of the four dimensions respectively, *e.g.*

```
(1,0,0,0)
(0,1,0,0)
(0,0,1,0)
(0,0,0,1)
```

This 4D enumeration consists of repeated 3D enumerations. If we are only interested in the last 3D object in 4D space (the hyperplane that belongs to the sum of all digits), we get four 3D objects in 4D space. What do they look like? We can project them one dimension lower by setting one dimension to zero. Three 3D objects are still translated, but one is set to the origin. So the 3D objects are in each of the four corners of a normal 3D-tetrahedron. That is one tetrahedron more than in right part of Figure 4. Does this mean that if we start with a 10-dimensional tuple, the plane in 2D has a very complicated shape? Fortunately, with every dimension we go, the lower hypertetrahedron that is at the origin is parallel to the enumeration and does not influence the intersections at the top hyperplane where we go next. Instead of only at most two hypertetrahedra intersecting, all hyperplanes intersect. Try to imagine this in 3D with four tetrahedra in four corners and arbitrary sizes.

toEnum, succ and pred with boundaries

The only thing that is left is to implement the function `toEnum`, adjusted to the boundary case in the same way as we showed for `fromEnum`. The functions `succ` and `pred` currently just run over the boundary by switching to an Int representation and searching for the next tuple inside of the boundaries. There may be some speed improvements that have not been considered, but our focus was on a correct implementation.

Conclusion

The idea to enumerate tuples this way originated from my diploma thesis, where I used an automata to generate tuples. When I started to make `Enum`-instances for tuples, it became much more involved—interesting enough to write an article about it. So far, I could not find a reference mentioning the enumeration of tuples with hyperplanes (*e.g.* [3]), but the field of mathematics and informatics is vast and the problem may have been called a different name or considered not worth spending time on.

The problem of the long calculation time still remains. Going back to the example in the introduction (reaching "Haskell" in an enumeration of 7-tuples) one could use syllables instead of characters. For this there have to be list-like data types with Enum instances. This should allow to set priorities which could be given by the user or found by a fitness function. Then it should also be possible to fix values inside of a tuple and do all kinds of filtering. The application I have in mind are user interfaces where one obviously has to make a lot of decisions. Another interesting challenge would be to implement this algorithm in a typical imperative language.

I want to thank Edward Z. Yang for his patience and the helpful editing.

Appendix

The images were generated with the library `collada-output` and blender for the 3D images and `diagrams` for the 2D-diagrams. Here is the code for one diagram:

```
main = B.writeFile "test.svg" $ toLazyByteString $
          renderDia SVG (SVGOptions (Dims 100 100)) (enum2 ||| enum3)

-- 2D enumeration

enum2 =  ((stroke ( mconcat $ translatedPoints))
             # fc black # fillRule EvenOdd )
      <> enumLines
      <> ( translate (-1,-1) $ text' "(0,0)")
      <> ( translate (4 ,-1) $ text' "(4,0)")
      <> ( translate (-1, 4.25) $ text' "(0,4)")

translatedPoints = map (\v -> translate v (circle 0.15) ) points

points = map (\(x,y) -> (fromIntegral x, fromIntegral y))
                              $ take 15 ( all2s :: [(Word8,Word8)] )

enumLines = mconcat $
            colorize (map enumLine (pointList points []) # lw 0.1)
 where enumLine l = fromVertices (map P l)
       pointList []         l = [l]
       pointList ((0,y):ps) l = ((0,y):l) : ( pointList ps [] )
       pointList ((x,y):ps) l = pointList ps (l ++ [(x,y)])

colorize = zipWith (\c d -> d # lc c) [yellow,red,green,blue,orange]

text' t = stroke (textSVG t 0.8) # fc black # fillRule EvenOdd
```

References

[1] David Lester Jeremy Gibbons and Richard Bird. Functional pearl: Enumerating the rationals. **Journal of Functional Programming**, Volume 16 Issue 3:pages 281–291 (May 2006).

[2] `http://hackage.haskell.org/package/tuple-gen`.

[3] Donald Knuth. **The Art of Computer Programming: Fascicle 2: Generating All Tuples and Permutations: 4**. Addison-Wesley Longman, Amsterdam (2005).

Understanding Basic Haskell Error Messages

by Jan Stolarek ⟨jan.stolarek@p.lodz.pl⟩

Haskell is a language that differs greatly from the mainstream languages of today. An emphasis on pure functions, a strong typing system, and a lack of loops and other conventional features make it harder to learn for programmers familiar only with imperative programming. One particular problem I faced during my initial contact with Haskell was unclear error messages. Later, seeing some discussions on #haskell, I noticed that I wasn't the only one. Correcting a program without understanding error messages is not an easy task. In this tutorial, I aim to remedy this problem by explaining how to understand Haskell's error messages. I will present code snippets that generate errors, followed by explanations and solutions. I used GHC 7.4.1 and Haskell Platform 2012.2.0.0 [1] for demonstration. I assume reader's working knowledge of GHCi. I also assume that reader knows how data types are constructed, what type classes are and how they are used. Knowledge of monads and language extensions is not required.

Compilation errors

Simple mistakes

I'll begin by discussing some simple mistakes that you are likely to make because you're not yet used to the Haskell syntax. Some of these errors will be similar to what you know from other languages—other will be Haskell specific.

Let's motivate our exploration of Haskell errors with a short case study. Standard `Prelude` provides some basic list operating functions like `head`, `tail`, `init` and `last`. These are **partial functions**. They work correctly only for a subset of all possible inputs. These four functions will explode in your face when you apply them to an empty list:

```
ghci> head []
*** Exception: Prelude.head: empty list
```

You end up with an exception that immediately halts your program. However, it is possible to create a safe version of `head` function which will work for all possible inputs without throwing an exception. Such functions are called **total functions**.

To reach our goal we will use the `Maybe` data type. The function will return `Nothing` when given an empty list; otherwise, it will return the result of applying `head` to `xs` wrapped in the `Just` constructor. Here's our completely bugged first attempt:

```
safeHead [a] -> Maybe a
safeHead [] = Nothing
safeHead xs = Maybe head xs
```

The first line is intended to be a type annotation, while the remaining two lines are the actual code. Loading this sample into `ghci` produces a **parse error**:

```
ghci> :l tmr.hs
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:1:14: parse error on input `->'
Failed, modules loaded: none.
```

Parse errors indicate that the program violates Haskell syntax. The error message starts in the third line (not counting the blank one). It begins with name of the file and exact location of the error expressed as line and column numbers separated with colons. In this case the error is in the first line, which means our intended type annotation. The compiler complains about `->` which was not expected to appear at this location. The problem is that the name of a function should be separated from type annotation with `::`. If there's no `::`, the compiler assumes we are defining a function body, treats `[a]` as a pattern binding and expects that it is followed either by `=` or `|` (a guard). Of course there are lots of other ways to cause parse error, but they all are dealt with in the same way: use your favourite Haskell book or tutorial to check what syntax Haskell expects for a particular expression. Let's fix this particular mistake by adding the missing `::` symbol:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead xs = Maybe head xs
```

and see what happens:

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:3:15: Not in scope: data constructor ‘Maybe’
Failed, modules loaded: none.
```

Now that the type annotation is correct, it turns out that there's an error in third line. **Not in scope** means that some variable, function or, as implied in this case, a data constructor, is not known to the compiler. You certainly know this kind of error from different programming languages. Let's look at the definition of `Maybe` data type to understand what is wrong:

```
ghci> :i Maybe
data Maybe a = Nothing | Just a -- Defined in ‘Data.Maybe’
```

In this definition, `Maybe` is a **type constructor**, while `Nothing` and `Just` are **value constructors**, also referred to as **data constructors**. This means that when you want to create a value of a given type you must use either `Nothing` or `Just`. In our code we have mistakenly used `Maybe`. There is no data constructor called `Maybe`: there is a type constructor called `Maybe`. Let's replace that `Maybe` with `Just`, which was our original intention:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead xs = Just head xs
```

The previous error is gone, only to produce a new one, shown in Listing 1. This time, the first two lines of the error message are quite explanatory, if you know

---

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:3:15:
    The function ‘Just’ is applied to two arguments,
    but its type ‘a0 -> Maybe a0’ has only one
    In the expression: Just head xs
    In an equation for ‘safeHead’: safeHead xs = Just head xs
Failed, modules loaded: none.
```

Listing 1: Applying data constructor to too many arguments.

that every data constructor is in fact a function. The definition of `Maybe` data type shows that the `Just` value constructor takes one parameter, while in our code we have mistakenly passed two parameters: `head` and `xs`. From a formal point of view this is a type system error. These will be discussed in more detail in the next section, but we treat this one here because it is very easy to make if you forget that function application is left-associative and that functions are curried by default. This means that `Just head xs` is the same as `((Just head) xs)`. We can use either parentheses or function composition and the `$` operator to override the default associativity. I've elaborated on the second approach on my blog [2] so I will not go into explaining it here; we'll just use the parentheses:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead xs = Just (head xs)
```

Surprise, surprise! The code finally compiles:

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )
Ok, modules loaded: Main.
```

Our function `safeHead` now works as intended:

```
ghci> safeHead []
Nothing
ghci> safeHead [1,2,3]
Just 1
```

You could implement safe versions of other unsafe functions in the same way, but there's no need to: there already is a library called `safe` [3], which provides different safe versions of originally unsafe functions.

Let's recall the not-in-scope error that we saw earlier. It was caused by using a function which didn't exist. Another common situation in which this error arises is when you use an existing function, but fail to import it. Let's look at a simple piece of code in which a different name is given to an already existing function:

```
module Main where
  sortWrapper xs = sort xs
```

Loading this code also produces the `not in scope` error:

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:2:22:
    Not in scope: ‘sort’
    Perhaps you meant ‘sqrt’ (imported from Prelude)
Failed, modules loaded: none.
```

GHCi doesn't know `sort` function, but it knows `sqrt` from standard `Prelude` and suggests that we might have made a typo. The `sort` function we want to use is not in the standard `Prelude` so it must be explicitly imported. A problem arises if you know that a certain function exists, but you don't know in which package it is located. For such situations use `hoogle` [4]. If you have `hoogle` installed localy you can look up package name like this:

```
[jan.stolarek@GLaDOS : ~] hoogle --count=1 sort
Data.List sort :: Ord a => [a] -> [a]
```

Module name is located before the function name. In this case it is `Data.List`. Now we can import module like this:

```
module Main where
  import Data.List (sort)
  sortWrapper xs = sort xs
```

or even more explicitly:

```
module Main where
  import qualified Data.List as Lists (sort)
  sortWrapper xs = Lists.sort xs
```

So if you get a not-in-scope error, but you know that the function really exists, the missing import is most likely the culprit.

Another common mistake made by beginners is attempting to invoke functions directly from a module. A typical example might look like this

```
module Main where
fact :: Int -> Int
fact 0 = 1
fact n = n * fact ( n - 1 )

print (fact 5)
```

This code gives a correct definition of factorial function and then tries to print the result of invoking this function. This, however, is incorrect and results in a following error:

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:6:1: Parse error: naked expression at top level
Failed, modules loaded: none
```

In Haskell everything in the module must be enclosed within function definitions. It is forbidden to call functions directly in a way shown in the above example. There are two possible solutions: first is removing the call to `print (fact 5)`, loading module into GHCi and invoking `fact 5` from interactive prompt. Alternatively, if you want your program to run as a separate executable, enclose the call to `print (fact 5)` within the `main` function, which is an entry point to every Haskell program:

```
module Main where
fact :: Int -> Int
fact 0 = 1
fact n = n * fact ( n - 1 )

main = print (fact 5)
```

This code can be compiled and executed as a standalone executable:

```
[jan.stolarek@GLaDOS : ~] ghc --make tmr.hs
[1 of 1] Compiling Main             ( tmr.hs, tmr.o )
Linking tmr ...
[jan.stolarek@GLaDOS : ~] ./tmr
120
```

A naked expression error can also be easily caused by accidentally typing `Import` instead of `import`:

```
module Main where
    Import Data.List
    sortWrapper xs = sort xs
```

This results in an error, because compiler treats `Import` as an application of data constructor to the parameter `Data.List`. Remember: capitalization matters in Haskell!

Type system errors

The complexity of Haskell's strong static type system can cause problems for beginners. Seemingly identical errors can result in very different error messages, and dealing with this can be a challenge. In this section I will discuss some common type system errors.

```
ghci> True && 1

 <interactive>:23:9:
    No instance for (Num Bool)
      arising from the literal ‘1’
    Possible fix: add an instance declaration for (Num Bool)
    In the second argument of ‘(&&)’, namely ‘1’
    In the expression: True && 1
    In an equation for ‘it’: it = True && 1
```

Listing 2: Non-exhaustive pattern in a guard.

I'll begin exploring errors related to type system with a very simple example shown in Listing 2. It demonstrates what can happen when you pass parameters of the wrong type to a function. The statement `No instance for (Num Bool) arising from the literal ‘1’` is the key to understanding the error message. The last three lines of the error message provide information on exact expression that caused the error. In our case that's the second argument of `(&&)` operator[1]. Even without knowing what exactly happened, it seems clear that there's a problem with `1` literal.

To understand what is going on, you need to know that numbers in Haskell are polymorphic. This means that when you write an integer literal in Haskell—just as we've written `1` in our example—it can be interpreted as different type depending on the context it is used in. Here's an example:

```
ghci> 1 :: Int
1
ghci> 1 :: Double
1.0
```

In the first case, the literal `1` is interpreted as an `Int`; in the second one, it is interpreted as a `Double`. Both uses are correct and don't cause a type error. This

[1]The `it` value, mentioned in the last line, is equal to the value of last expression evaluated in GHCi.

```
ghci> :i Bool
data Bool = False | True -- Defined in GHC.Bool
instance Bounded Bool -- Defined in GHC.Enum
instance Enum Bool -- Defined in GHC.Enum
instance Eq Bool -- Defined in GHC.Classes
instance Ord Bool -- Defined in GHC.Classes
instance Read Bool -- Defined in GHC.Read
instance Show Bool -- Defined in GHC.Show
```

Listing 3: Information about `Bool` data type.

works as if the `fromInteger` function defined in the `Num` class was called implicitly on the numeric literal. This means that `True && 1` and `True && fromInteger 1` are equivalent expressions. Let's check the type signature of `fromInteger`:

```
ghci> :t fromInteger
fromInteger :: Num a => Integer -> a
```

This means that `fromInteger` takes an `Integer` and returns a value of **any** type `a`, with a restriction that the type `a` belongs to the `Num` type class. What type exactly should be returned? That depends on the context in which `fromInteger` was applied. For example, if the return value is required to be `Int` then the implementation of `fromInteger` defined in the `Num Int` instance declaration is called. That specific implementation returns a value of type `Int`. This mechanism allows integer literal to become an instance of a type belonging to `Num` type class.

With this knowledge, we can go back to our example in which integer literal `1` was used as a parameter to `(&&)` function. This function takes two `Bool` parameters and returns a single `Bool`:

```
ghci> :t (&&)
(&&) :: Bool -> Bool -> Bool
```

which means that in order for literal `1` to be a valid parameter to `(&&)`, the type returned by `fromInteger` should be `Bool`. There is one problem though. The `Bool` type is not an instance of `Num` type class, as shown in Listing 3. However, it should be, since the `fromInteger` function imposes a constraint that its return value belongs to `Num` type class. This is exactly what the error message said and that is the reason why a type error occurs. The next line of the error message suggests a solution: `Possible fix: add an instance declaration for (Num Bool)`. Indeed, if we made `Bool` type an instance of `Num` type class the problem would be gone. In some cases, this may be the solution. Deriving `Bool` to be an

instance of `Num` is certainly a good exercise that you can try out. In many other cases however this error means you wanted something different than you actually wrote. Let's assume here that we wanted `1` to denote logical truth. Fixing that is easy:

```
ghci> True && True
True
```

The constraints imposed by the type classes propagate in the type system. Here's a simple example that demonstrates this: let's say we want to write a function that tells if the two parameters passed to it are equal:

```
isEq :: a -> a -> Bool
isEq x y = x == y
```

Our `isEq` function expects two parameters that are of the same type and returns a `Bool`. Our code looks perfectly reasonable, but loading that code into GHCi results with an error shown in Listing 4.

```
ghci> :l tmr.hs
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:2:14:
    No instance for (Eq a)
      arising from a use of ‘==’
    In the expression: x == y
    In an equation for ‘isEq’: isEq x y = x == y
Failed, modules loaded: none.
```

Listing 4: Error caused by a lack of type class constraint.

The first two lines of this message are the most important: they say that type `a` should be an instance of `Eq` type class and that this requirement is imposed by the use of `==` function. Let's inspect the type of `(==)`:

```
ghci> :t (==)
(==) :: Eq a => a -> a -> Bool
```

Indeed the `(==)` function expects that its arguments are instances of `Eq` type class. That constraint was propagated to `isEq` function, which uses `(==)`. We must therefore add a type class constraint to parameters of `isEq`:

```
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:2:17:
  Could not deduce (b ~ a)
  from the context (Eq a, Eq b)
    bound by the type signature for
             isEq :: (Eq a, Eq b) => a -> b -> Bool
    at tmr.hs:2:1-17
    ‘b’ is a rigid type variable bound by
      the type signature for isEq :: (Eq a, Eq b) => a -> b -> Bool
      at tmr.hs:2:1
    ‘a’ is a rigid type variable bound by
      the type signature for isEq :: (Eq a, Eq b) => a -> b -> Bool
      at tmr.hs:2:1
  In the second argument of ‘(==)’, namely ‘y’
  In the expression: x == y
  In an equation for ‘isEq’: isEq x y = x == y
Failed, modules loaded: none.
```

Listing 5: Comparing two different instances of `Eq` type class.

```
isEq :: Eq a => a -> a -> Bool
isEq x y = x == y
```

This fixes the problem. Let's now see what happens if we try to compare parameters of two different types, both belonging to `Eq` type class:

```
isEq :: (Eq a, Eq b) => a -> b -> Bool
isEq x y = x == y
```

The error message is shown in Listing 5. The `(b ~ a)` notation indicates equality of the types `a` and `b`. The compiler uses this notation to say `Could not deduce (b ~ a)`, which means that `a` and `b` should be identical, but the type definition we provided does not guarantee it. The requirement of types `a` and `b` being of the same type is a direct consequence of a type signature of `(==)` which, as we recall, requires its parameters to be of the same type. The term `rigid type variable` indicates that our types `a` and `b` have been directly specified by the type annotation [5] and the compiler is not free to unify them[2]. We have already seen a correct version of this code, but we can also make it work in a different way:

[2]Unification of two types means that they are assumed to be the same type.

```
{-# LANGUAGE TypeFamilies #-}

isEq :: (Eq a, Eq b, a ~ b) => a -> b -> Bool
isEq x y = x == y
```

Enabling `TypeFamilies` language extension and adding `a ~ b` type constraint informs the compiler that `a` and `b` can be unified to be the same type. The above code is for demonstration purposes: it makes no real sense to write the type declaration in this way, since `a` and `b` will be unified into one type. It can be written in a more straightforward way. Loading this into GHCi and checking type of `isEq` will show that it's actually `isEq :: Eq b => b -> b -> Bool`. The two distinct types introduced deliberately have disappeared because the compiler was allowed to unify them.

When you begin working with type classes, you may find it difficult what type class constraints to impose on your functions. Here's where the Haskell's type inference is useful: write down your function without the type declaration, load it into GHCi and use Haskell's type inference to determine function's type for you. This can be done using the `:t` command on a function. Suppose the `isEq` function was written without type declaration. Here's what Haskell infers about the `isEq`'s type:

```
ghci> :t isEq
isEq :: Eq a => a -> a -> Bool
```

This is a correct type signature and you can simply copy it to your code.

We've seen what will happen when type signature does not contain appropriate class restrictions. Let's now see what happens when a function's type signature is inconsistent with function's implementation. Assume we want to write a function that returns a first letter from a `String`. We could want to have a type signature like this:

```
getFirstLetter :: String -> String
```

This means that `getFirstLetter` function takes a value of type `String` and returns a value of type `String`. For example, if we pass `"Some string"` as a parameter then `getFirstLetter` will return value `"S"`. Since `String` in Haskell is a synonym for `[Char]` (list of `Char`s) we could use `head` function to take the first element of a `String`. Our `getFirstLetter` function would then look like this:

```
getFirstLetter :: String -> String
getFirstLetter = head
```

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:2:18:
    Couldn't match expected type 'String' with actual type 'Char'
    Expected type: String -> String
      Actual type: [Char] -> Char
    In the expression: head
    In an equation for 'getFirstLetter': getFirstLetter = head
Failed, modules loaded: none
```

Listing 6: Expected and actual type mismatch error.

The `head` function has type `[a] -> a`, which means it takes a list of some type `a` and returns a single element of type `a`, not a list. However, the type signature for `getFirstLetter` requires that the return argument be a list of `Char`s. Therefore, the provided signature is inconsistent with the actual return type of the function. Haskell will notice that when you try to load the code into GHCi and report an error as shown in Listing 6. It says that type annotation in the source code defines the return type to be `[Char]`, but the actual type inferred by the compiler is `Char`. There are two possible fixes for this problem. First, if we are really fine with getting a single `Char` instead of a `String`, we can change the type signature to `String -> Char` (or `[Char] -> Char`, since it's the same due to type synonyms). On the other hand, if we really expect to get a `String` from our function, we need to change the implementation. In this case the result of `head` should be wrapped in a list, which can be done like this:

```
getFirstLetter :: String -> String
getFirstLetter xs = [head xs]
```

or using a point-free style:

```
getFirstLetter :: String -> String
getFirstLetter = (: []) . head
```

The type mismatch error is probably the one you'll be seeing most often. There is really no way of anticipating all possible situations where it might occur. The above case was easy, because the implementation of a function was correct and the only problem was fixing the type annotation. Most often however you'll end up in situations where you've written some code and the type-checker complains that this code is type-incorrect. Here's an example of how this may look like:

```
ghci> :r
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:4:26:
    Couldn't match expected type 'Int' with actual type 'Char'
    In the first argument of '(*)', namely 'acc'
    In the first argument of '(+)', namely 'acc * 10'
    In the expression: acc * 10 + digitToInt x
Failed, modules loaded: none.
```

Listing 7: Expected and actual type mismatch error.

```
import Data.Char

asInt :: String -> Int
asInt = foldl (\x acc -> acc * 10 + digitToInt x) 0
```

This code takes a `String` representation of a number and turns it into an `Int`. Well... almost: if you try to compile it you'll get an error about mismatching types, as shown in Listing 7. The message says that `acc` (accumulator) is of the wrong type—`Char` instead of `Int`. In such cases it might not be immediately obvious what exactly is wrong. The only way to deal with such errors is to inspect the incorrect expressions. In this example the code isn't really complicated so this will be easy. The `asInt` function uses only `foldl` and one anonymous lambda function. Let's begin by checking the type signature of `foldl`:

```
ghci> :t foldl
foldl :: (a -> b -> a) -> a -> [b] -> a
```

This type signature says that `foldl` takes three parameters. These parameters are a function of type `a -> b -> a`, an accumulator of type `a` and a list containing elements of type `b`. By looking at the type variables in this signature—that is, `a` and `b`—you can see that the first parameter to folding function, the accumulator and the return value of `foldl` are of the same type `a`. We passed `0` as the accumulator and expect the return value of `asInt` function to be `Int`. Therefore type `a` is inferred to be `Int`. The compiler complains, however, that variable `acc` used as parameter to `(*)` is of type `Char`, not `Int`. The parameters to the lambda function are `x` and `acc`. According to type signature of `foldl`, `x` should be of type `Int`, because it is of the same type as the accumulator. On the other hand `acc` is of the same type as elements of the list passed as third parameter to `foldl`. In this

```
ghci> :m + Data.List
ghci> 'A' `isPrefixOf` "An error"

 <interactive>:8:1:
    Couldn't match expected type `[a0]' with actual type `Char'
    In the first argument of `isPrefixOf', namely 'A'
    In the expression: 'A' `isPrefixOf` "An error"
    In an equation for `it': it = 'A' `isPrefixOf` "An error"
```

Listing 8: Passing argument of wrong type to a function.

example, the list is of type `[Char]` so `acc`, being a single element, is of type `Char`. Well, this should be the other way around: `acc` should be of type `Int`; `x` should be of type `Char`. We conclude this error was caused by writing the parameters of lambda in incorrect order. The correct code therefore is:

```
import Data.Char

asInt :: String -> Int
asInt = foldl (\acc x -> acc * 10 + digitToInt x) 0
```

In some cases you will get an error message that doesn't specify concrete type. Consider example shown in Listing 8. The `isPrefixOf` function expects its both parameters to be lists of the same type:

```
ghci> :t isPrefixOf
isPrefixOf :: Eq a => [a] -> [a] -> Bool
```

As soon as compiler realizes that the first parameter is not a list, it complains. It doesn't even reach the second parameter to analyze it and infer that the first list should contain `Char`s. That is the reason why it uses a list `[a0]`, where `a0` represents any type. Listing 9 shows the result of swapping the two parameters. This time, the compiler is able to infer the exact type of the elements required in the second list. By the time compiler reaches the second parameter, it has already analyzed the first one and it knows that that parameter was a list of `Char`s.

Another common error related to types is type ambiguity. I'll demonstrate it using the `read` function, which is used to convert `String` representation of some value to that value. The `read` function is defined in the `Read` type class and has type signature `read :: Read a => String -> a`. All data types within standard Prelude, except function types and `IO` types, are instances of `Read`. Every type

```
ghci> "An error" `isPrefixOf` 'A'

 <interactive>:9:25:
    Couldn't match expected type `[Char]' with actual type `Char'
    In the second argument of `isPrefixOf', namely 'A'
    In the expression: "An error" `isPrefixOf` 'A'
    In an equation for `it': it = "An error" `isPrefixOf` 'A'
```

Listing 9: Passing argument of a wrong type to a function.

```
ghci> read "5.0"

 <interactive>:11:1:
   Ambiguous type variable `a0' in the constraint:
     (Read a0) arising from a use of `read'
   Probable fix: add a type signature that fixes these
     type variable(s)
   In the expression: read "5.0"
   In an equation for `it': it = read "5.0"
```

Listing 10: Type ambiguity.

that is an instance of `Read` type class provides its own definition of `read` function. This means that the compiler must know what resulting type to expect in order to call a correct implementation of the `read` function[3]. This is polymorphism, which was already discussed when we talked about `fromIntegral` function. Listing 10 shows that the error that occurs when the polymorphic type variable cannot be inferred to a concrete type. In this case, the compiler doesn't know which version of `read` function should be called. It suggests that we can solve the problem by adding a type signature. Let's try that:

```
ghci> read "5.0" :: Double
5.0
```

It is important to provide correct type signature:

```
ghci> read "5.0" :: Int
*** Exception: Prelude.read: no parse
```

[3]Even if there was only one instance of `Read` class, the compiler wouldn't know that and you would get type ambiguity error.

```
ghci> id -1

 <interactive>:16:4:
    No instance for (Num (a0 -> a0))
      arising from a use of '-'
    Possible fix: add an instance declaration for (Num (a0 -> a0))
    In the expression: id - 1
    In an equation for 'it': it = id - 1
```

Listing 11: Incorrect usage of unary negation.

This code produces an exception because implementation of `read` for `Int` instances of `Read` doesn't expect any decimal signs.

The type system can come into play in very unexpected moments. Let's play a little bit with `id`, the identity function, which returns any parameter passed to it:

```
ghci> id 1
1
ghci> id 0
0
ghci> id (-1)
-1
```

Notice that the negative parameter was wrapped in parentheses. Had we neglected this, it would result in a `No instance for (Num (a0 -> a0))` error, as shown in Listing 11. We've already seen this error before when we talked about polymorphism of integer literals. This time the compiler expects the `a0 -> a0` type to be an instance of `Num` type class. The `a0 -> a0` denotes a function that takes an value of type `a0` and returns a value of the same type `a0`. What seems strange is the fact that the compiler expects literal `1` to be of type `a0 -> a0`. The problem here is that the `(-)` sign is treated as an infix binary operator denoting subtraction, not as unary negation operator as we expected [4]. The strange error message blaming `1` literal is caused by the fact that `(-)` operator expects its parameters to be of the same type:

```
ghci> :t (-)
(-) :: Num a => a -> a -> a
```

[4] Section 3.4 of Haskell 2010 Language Report [6] gives more detail on that matter.

```
ghci> id - "a string"

 <interactive>:11:6:
   Couldn't match expected type ‘a0 -> a0’ with actual type ‘[Char]’
   In the second argument of ‘(-)’, namely ‘"a string"’
   In the expression: id - "a string"
   In an equation for ‘it’: it = id - "a string"
```

Listing 12: Error caused by incorrect parameters that should be of the same type.

The type of first parameter was inferred to be `a0 -> a0`. Therefore the second parameter is also expected to be of type `a0 -> a0`, but this type is not an instance of `Num` type class. Just as before, this error results from the implicit use of `fromIntegral` function. In this example, however, there's one thing that you might be wondering. From the type signature of `(-)`, we can see that its parameters should belong to `Num` type class. The question is this: how can we be certain that this error is raised by the constraint in the `fromIntegral` function and not by the constraint in the `(-)` function itself? There's an easy way to verify this. Let's replace the second argument of `(-)` with a value of type `String`. We use `String`, because string literals don't implicitly call any function that would impose additional type constraints. The error that results in this case, shown in Listing 12, says that compiler expects the second argument to be the same type as the first one, which is a restriction resulting from the type signature of `(-)`. There is no complaint about `Num` type class, which allows to infer that at this point `Num` type class constraint imposed by `(-)` hasn't been checked yet. Let's verify this conclusion by supplying `(-)` with two arguments of the same type that is not an instance of `Num`. The result is shown in Listing 13. This time the compiler successfully verified that both parameters of `(-)` are of the same type `a0 -> a0`, and it could go further to check if type class constraints are satisfied. However, the `a0 -> a0` type is not an instance of `Num` type class, hence the type class constraint is violated and `No instance for` error arises.

Some runtime errors

We finally managed to get past the compilation errors. It was a lot of work, probably more than in other programming languages. That's another characteristic feature of Haskell: the strong type system moves much of the program debugging up front, into the compilation phase. You probably already heard that once a Haskell program compiles it usually does what the programmer intended. That's

```
ghci> id - id

 <interactive>:20:4:
    No instance for (Num (a0 -> a0))
      arising from a use of '-'
    Possible fix: add an instance declaration for (Num (a0 -> a0))
    In the expression: id - id
    In an equation for 'it': it = id - id
```

Listing 13: Another `No instance for (Num (Int -> Int))` error.

mostly true, but this doesn't mean that there are no runtime errors in Haskell. This section will discuss some of them.

Runtime errors in Haskell most often take the form of runtime exceptions. At the very beginning of this paper, I showed you that some functions don't work for all possible inputs and can raise an exception:

```
ghci> head []
*** Exception: Prelude.head: empty list
```

In languages like Java, exceptions are your friend. They provide a stack trace that allows to investigate the cause of an error. It is not that easy in Haskell. Runtime exceptions don't give you any stack trace, as this is not easily implemented in a language with lazy evaluation. You are usually left only with a short error message and line number.

One of the most commonly made errors resulting in runtime exceptions is non-exhaustive patterns in constructs like function definitions or guards. Let's recall `safeHead` function that we've written in the first section:

```
safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHEad xs = Just (head xs)
```

This function contains a typo: the function name in the third line is misspelled as `safeHEad` (notice the capital `E`). This code compiles perfectly, but will result in an error if we try to call `safeHead` function for non-empty list:

```
ghci> safeHead []
Nothing
ghci> safeHead [1,2,3]
*** Exception: tmr.hs:2:1-21: Non-exhaustive patterns in function
safeHead
```

The argument `[1,2,3]` passed to the function `safeHead` couldn't be matched against any pattern in the function definition. That's what **non-exhaustive pattern** error means. This is due to the typo I made in the third line. For Haskell compiler everything is perfectly fine. It treats the code as definition of two different functions: `safeHead`, matching only empty lists, and `safeHEad`, matching both empty and non-empty lists. Note that applying `safeHEad` to empty list will result in a runtime exception.

We were unexpectedly hit by the non-exhaustive pattern problem during runtime because GHCi has most warnings disabled by default. You can enable most of the warnings by passing `-Wall` command line switch when running `ghci` command[5]. Now GHCi will warn us during the compilation about non-exhaustive patterns in `safeHead` function and lack of type signature for accidentally defined `safeHEad` function. See Listing 14.

```
ghci> :l tmr.hs
[1 of 1] Compiling Main             ( tmr.hs, interpreted )

tmr.hs:2:1:
  Warning: Pattern match(es) are non-exhaustive
       In an equation for ‘safeHead’: Patterns not matched: _ : _

tmr.hs:3:1:
  Warning: Top-level binding with no type signature:
       safeHEad :: forall a. [a] -> Maybe a
Ok, modules loaded: Main.
```

Listing 14: Compilation warnings.

The non-exhaustive pattern error can also occur in an incorrect guard. To illustrate this, let's create our own signum function:

```
mySignum :: Int -> Int
mySignum x
    | x > 0  = 1
    | x == 0 = 0
```

You see the error, don't you? Listing 15 show what happens when we call `mySignum` function for negative arguments. This error is easily corrected by adding the third guard:

[5] See [7] for more details.

```
ghci> mySignum 1
1
ghci> mySignum 0
0
ghci> mySignum (-1)
*** Exception: tmr.hs:(14,1)-(16,16): Non-exhaustive patterns in
function mySignum
```

Listing 15: Non-exhaustive pattern in a guard.

```
mySignum :: Int -> Int
mySignum x
    | x > 0     = 1
    | x == 0    = 0
    | otherwise = -1
```

The `otherwise` function is defined to always return `True`, so the third guard will always evaluate if the previous guards didn't. The compiler can also give a warning about non-exhaustive guards in a same way it gives warning about non-exhaustive patterns in function definition. Remember that order of guards matters, so `otherwise` must always be the last guard.

Summary

This completes our overview of basic Haskell error messages. By now, you should know how to read error messages and how to correct the problems in your code that caused them. The above list is by no means an exhaustive one: there are a lot of different errors you will encounter when you start using some more advanced features of Haskell. Perhaps one day you'll even write a program that will cause `My brain just exploded` error. Until then, happy error solving!

Acknowledgements

I thank the great Haskell community at #haskell IRC channel. They helped me to understand error messages when I was beginning my adventure with Haskell. Many thanks go to Edward Z. Yang for his feedback. I also thank Marek Zdankiewicz for reviewing draft version of this paper.

References

[1] http://hackage.haskell.org/platform/.

[2] http://ics.p.lodz.pl/~stolarek/blog/2012/03/function-composition-and-dollar-operator-in-ha

[3] http://hackage.haskell.org/package/safe.

[4] http://www.haskell.org/hoogle/.

[5] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In **ICFP**, pages 50–61 (2006).

[6] http://www.haskell.org/onlinereport/haskell2010/.

[7] http://www.haskell.org/ghc/docs/latest/html/users_guide/options-sanity.html.

The MapReduce type of a Monad

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

MapReduce is a popular paradigm for distributed computing that is particularly important as a framework for massively parallel computation. Many existing algorithms have been recast in terms of MapReduce, and new algorithms are being discovered in the paradigm. The MapReduce framework imposes a fixed structure on algorithms: they consist of one or more stages, which are executed in series. Each stage takes as input a list of key/value pairs and produces as output another such list. In each stage, the input key/value pairs are sorted on the key value and divided into chunks, one per key value. Each chunk is passed to a transformation function. The resulting key/value pairs are gathered together and output as a list which can be passed to the next stage.

In this paper, we build on our earlier paper [1], 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 seen as the result of applying a general transformation, which is a kind of monad transformer, to the List monad. Finally, we show that the familiar Haskell State monad can be seen as equivalent to the MapReduce type associated to the reader monad, of functions $s \to a$ for fixed s. This raises the question of how many other applications, apart from the obvious one of MapReduce itself, this transformer might have.

All of the ideas described in this paper have been implemented, and we have successfully demonstrated a MapReduce application using the transformed List monad to represent MapReduce. A DARCS repository containing the code may be found at [2].

The idea

MapReduce in a nutshell

We start with a brief summary of the MapReduce algorithm. MapReduce takes a list of key/value pairs and processes them using a cluster of many processing nodes and one control node. A MapReduce algorithm consists of a number of repetitions of a basic processing step, which has two parts:

1. **Map**. The control node distributes the list of values randomly between the processing nodes. Each processing node applies a function called a *mapper* to its values and produces a new list of key/value pairs, which it returns to the control node.
2. **Reduce**. This has three sub-parts:
   a) The control node gathers the outputs from each of the processing nodes, concatenates them and sorts the resulting list by the key. It divides the sorted list into chunks, one per key value and distributes the chunks among processing nodes, each chunk to one node.
   b) Each processing node takes the chunk of key/value pairs it has been passed and uses them as input to a function called a *reducer*. This produces a list of values.
   c) The processing nodes return their output lists to the control node, which concatenates them to form a single list of values.

The control node then concatenates these output lists and proceeds to use them as input to the Map part of the next stage of processing.

Observe that we if we modify this by making Reduce produce not key/value pairs with random keys, then the random distribution of values among processing nodes in Map can be done with the distribution algorithm used in Reduce. Therefore. we can treat Map and Reduce as being two instances of the same basic processing stage.

Generalising MapReduce

Looking at this abstractly, what we have is a function

$$f :: a \to [(x, a)] \to [(y, b)]$$

Here and elsewhere, x, y and z are value types and a, b and c are key types. The first argument is the key value that selects a chunk of data and the second argument is the data, consisting of key/value pairs. The function therefore corresponds to selecting a key, extracting the corresponding values and applying a mapper or reducer to them.

The mapper and reducer transform lists of values into lists of key/value pairs

$$[x] \to [(y, b)]$$

where x, y and b are as above, so a stage takes a function (mapper or reducer) $[x] \to [(y, b)]$ and turns it into a function $a \to [(x, a) \to (y, b)]$, and so looks like

$$wrap :: ([x] \to [(y, b)]) \to a \to [(x, a)] \to [(y, b)]$$

Looking at the algorithm as a whole, a MapReduce process is a function

$$[(x, a)] \to [(y, b)]$$

where x, y, a, and b are as above. Applying a mapper or reducer to the output of a MapReduce process gives another MapReduce process, so the act of adding a new stage to the end of a chain of processing can be described as

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

where z is a value type and c is a key type, where the existing process is the first argument, the additional stage is the second, and the resulting combined process is the output. This looks like the signature of a monadic bind where the underlying type is a function $([(a, x)] \to [(b, y)])$. Therefore, MapReduce can be expressed as

$$... \overset{MR}{\gg\!=} wrap\ map \overset{MR}{\gg\!=} wrap\ reduce \overset{MR}{\gg\!=} ...$$

where $\overset{MR}{\gg\!=}$ is a suitable bind function.

A quasi-monadic approach to MapReduce

Quasi-monads

A standard monad is a parameterised type $m\ u$ with operations

$$return :: u \to m\ u$$
$$(\gg\!=) \ :: m\ u \to (u \to m\ v) \to m\ v$$

We need to generalise this. First, we need monads with two parameters. There are several ways of doing this; for our purposes, we take types $m\ t\ u$ with operations

$$return :: t \to m\ t\ t$$
$$(\gg\!=) \ :: m\ t\ u \to (u \to m\ u\ v) \to m\ u\ v$$

We need one further generalisation. The structure we will define obeys natural generalisations of the first two monad laws, but the third breaks down. The third monad law is

$$(x \gg\!= f) \gg\!= g \equiv x \gg\!= (\lambda y \to f\ y \gg\!= g)$$

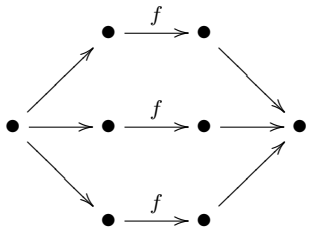
To see why it must break down, let us se how we could interpret this in terms of the model set out above, where x is a MapReduce process and f and g correspond to individual stages.

The left hand side has obvious meaning: x is a process to which we append first the stage f and then the stage g. On the right hand side we have $\lambda y \to f\ y \overset{MR}{\gg\!=} g$ which is the concatenation of two stages, but we cannot append g to f until we know what the output of f is, because a critical part of appending a stage to a process is dividing the process' output into chunks based on their key. So for MapReduce, $\lambda y \to f\ y \overset{MR}{\gg\!=} g$ is meaningless: the only meaningful order in which to apply stages is one at a time starting from the left:

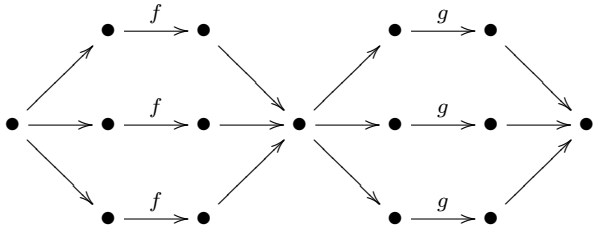
$$(((x \overset{MR}{\gg\!=} f) \overset{MR}{\gg\!=} g) \overset{MR}{\gg\!=} h) \overset{MR}{\gg\!=} ...$$

That is to say, stages cannot be composed together without an input being specified. Thus, we cannot expect MapReduce to obey the third monad law, because of the sorting and partitioning on keys that is a fundamental part of the algorithm.

Looking at this another way, we can, of course, combine two stages, but the resulting entity is not itself a stage. A single stage sorts on the key, passes the resulting chunks through a function, then recombines the result, represented by this data-flow pattern:



So combining two stages gives the pattern:



which is clearly only equivalent to a single stage when f and g are extremely simple. This is not surprising: otherwise MapReduce could be collapsed down to a single stage, and would not be a very interesting processing paradigm. What this means, then, is that if it were possible to compute

$$\lambda y \to f\ y \overset{MR}{\gg\!=} g$$

then it would not be anything that could be composed with x via $\overset{MR}{\gg\!=}$, so

$$x \overset{MR}{\gg\!=} (\lambda y \to f\ y \overset{MR}{\gg\!=} g)$$

is meaningless. Sorting on keys therefore makes it impossible that MapReduce could obey the third monad law.

Therefore, we coin the term **quasi-monad** to indicate an object that has monadic operations and obeys natural generalisations of the first two monad laws, but does not obey the third law. We have just seen that any complex stage-based distributed algorithm must fail to obey the third law, and so is at most a quasi-monad. In fact, MapReduce is a two-parameter quasi-monad:

$$\textbf{newtype}\ \mathit{MapReduce}\ x\ a\ y\ b = \mathit{MR}\ \{\mathit{run} :: ([(x, a)] \to [(y, b)])\}$$

We shall assume that all key types belong to the class *Eq*, which is necessary to enable sorting and partitioning on keys.) This should remind us of the *State* monad with, as it turns out, good reason. The monadic operations are:

```
1 return :: b → MapReduce x a x b
2 return k = MR (map (first $ const k))
3 (>>=) :: MapReduce x a y b → (b → MapReduce y b z c) → MapReduce x a z c
4 (>>=) f g = MR (λkvs →
5   let
6     fs = run f kvs
7     gs = map g $ nub $ fst <$> fs
8   in
9   concatMap (λx → run x fs) gs)
```

So *return* forces all keys to a given quantity. To see how $\overset{MR}{\gg\!=}$ works, rewrite lines 6 and 7 as

$$\begin{aligned} fs &= \mathit{run}\ f\ \mathit{kvs} \\ ks &= \mathit{nub}\ (\mathit{fst} \mathbin{\langle\$\rangle} fs) \\ gs &= \mathit{map}\ g\ gs \end{aligned}$$

So *fs* is the output of the mapper/reducer, *ks* is the *set* of keys from the output, and *gs* is a set of functions, one per key in the set. So we have one function per key in the output of f. Then in *concatMap*, we apply each function to the full data set (leaving it to each function to decide what subset of the data to process) and combine the result.

They do this with the help of the function *wrap*, which, we recall, takes a key and a function and then applies the function to those records matching the key. We can now write *wrap* as

$$
\begin{aligned}
&\mathit{wrap}' :: ([x] \to [(b, y)]) \to (a \to \mathit{MapReduce}\ x\ a\ y\ b) \\
&\mathit{wrap}'\ f = (\lambda k \to \mathit{MR}\ (g\ k)) \\
&\quad \textbf{where} \\
&\quad g\ k\ \mathit{kds} = f\ \$\ \mathit{snd}\ \langle\$\rangle\ \mathit{filter}\ (\lambda \mathit{kd} \to k \equiv \mathit{fst}\ \mathit{kd})\ \mathit{kds}
\end{aligned}
$$

As can be seen, it selects the key/value records in *kds* whose key is k, extracts their values and then pushes the resulting list through f.

MapReduce as a monad transformer

The *MapReduceT* type

[] is a monad with $\gg\!=\ \equiv$ *concatMap*, so *MapReduce* can be thought of as a transformation of the list monad. It turns out that there is a very natural way of describing a general transformer that will apply to any monad, with *MapReduce* the result of applying it to [].

So, let m be a monad; then the *MapReduceT* **type of** m is

$$\textbf{newtype}\ (\mathit{Monad}\ m) \Rightarrow \mathit{MapReduceT}\ m\ t\ u = \mathit{MR}\ \{\mathit{run} :: m\ t \to m\ u\}$$

From now on, m will be a monad type, while t, u and v are arbitrary data types parameterising m. There is also a natural way to lift monads into the transformer:

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

In addition, we define the operations:

$$
\begin{aligned}
&(\succ) ::\ \ \mathit{MapReduceT}\ m\ t\ u \to \mathit{MapReduceT}\ m\ u\ v \to \mathit{MapReduceT}\ m\ t\ v \\
&(\succ)\ f\ g\ \ = \mathit{MR}\ (\lambda \mathit{ss} \to \mathit{run}\ g\ \$\ \mathit{run}\ f\ \mathit{ss}) \\
&(-\!\prec) :: \mathit{MapReduceT}\ m\ t\ u \to m\ t \to m\ u \\
&(-\!\prec)\ f\ x = \mathit{run}\ f\ x
\end{aligned}
$$

These are arrow-like but *MapReduceT* m is not itself an arrow. Note that $\succ$ is just function composition in disguise, and $-\!\prec$ is just function application in disguise.

Now, if *MapReduceT* is to be a monad transformer, then *MapReduceT* m must be a quasi-monad with a *lift* operation. The quasi-monadic operations for *MapReduceT* m are:

```
1 return :: (Monad m) ⇒ t → MapReduceT m t t
2 return x = lift (return′ x)
3 bind :: (Monad m) ⇒ MapReduceT m u u → MapReduceT m t u
4                   → (u → MapReduceT m u v) → MapReduceT m t v
5 bind p f g = MR (λxs → ps xs ≫= gs xs)
6                 where
7                 ps xs    = (f ➢ p) ≺ xs
8                 gs xs x = (f ➢ g x) ≺ xs
```

where $\gg\!=$ is bind in m and *return′* is return in m. If $p :: MapReduceT\ m\ u\ u$ define

$$(\gg\!=_p) = bind\ p$$

The parameter p seems mysterious, and it is not clear why we do not just take $p = MR\ id$. The reason is that, in order to generalise MapReduce, we need a way of filtering the output of f before passing it to g. So p is the equivalent of the $nub \circ snd \langle\$\rangle$ in $\overset{MR}{\gg\!=}$.

It can be shown that that $MapReduceT\ m$ obeys natural modifications of the first and second monad laws, but the third law breaks down entirely, as expected. Subject to the reasonable assumption that $run\ p\ \$\ return'\ x \equiv return'\ x$, which is certainly true in all the cases we care about, we have:

$$f \gg\!=_p return \equiv f \gg\!= p$$
$$return\ x \gg\!=_p f \equiv return\ x ➢ f\ (return'\ x)$$

which are natural generalisations of the standard monadic laws. Any attempt at proving an analogue to the third law quickly becomes intractable, however, because the crucial filter function p mixes with the monadic functions in complex and confusing ways.

Equivalence to *MapReduce*

Before we can show that *MapReduce* is equivalent to $MapReduceT\ [\,]$ we need to choose a filter function, that is, a suitable value of p. Define

```
dedupe :: (Eq a) ⇒ MapReduceT [] (a, x) (a, x)
dedupe = MR (nubBy (λkd kd′ → fst kd ≡ fst kd′))
```

Clearly *dedupe* takes a list of key/value pairs and returns a list with one pair per unique key (whose value is ill-defined but irrelevant). If we wanted to be really rigorous we could amend the definition to read

$$dedupe' :: (Eq\ a) \Rightarrow MapReduceT\ [\,]\ (a, Maybe\ x)\ (a, Maybe\ x)$$
$$dedupe' = MR\ (\lambda vks \rightarrow second\ (const\ Nothing)\ \Diamond\$\Diamond\ (nubBy\ (\lambda kd\ kd' \rightarrow fst\ kd \equiv fst\ kd'))$$

which makes the output determinate at the price of hugely complicating the proof of equivalence and adding no meaningful value. Therefore we stick with *dedupe*.

Let x, y and z be value types and a, b and c be key types, and set $t = (a, x)$, $u = (b, y)$ $v = (c, z)$. Therefore, trivially, at the level of type definitions

$$MapReduce\ x\ a\ y\ b \equiv MapReduceT\ [\,]\ t\ u$$

We now show that this equality extends to the quasi-monadic structures.

Say we are given $f :: MapReduce\ x\ a\ y\ b$ and $g :: (b \rightarrow MapReduce\ y\ b\ z\ c)$. We rewrite the definition of $\overset{MR}{\gg\!=}$ as:

$$\begin{aligned}
&f \overset{MR}{\gg\!=} g = MR\ (\lambda xs \rightarrow \\
&\quad \textbf{let} \\
&\quad\quad fs = f \prec\!\!- xs \\
&\quad\quad gs = map\ (g \circ fst)\ \$\ dedupe \prec\!\!- fs \\
&\quad \textbf{in} \\
&\quad concat\ \$\ map\ (\lambda y \rightarrow y \prec\!\!- fs)\ gs)
\end{aligned}$$

Rewriting this some more we get:

$$\begin{aligned}
&f \overset{MR}{\gg\!=} g = MR\ (\lambda xs \rightarrow \\
&\quad \textbf{let} \\
&\quad\quad fs = f \prec\!\!- xs \\
&\quad\quad ks = dedupe \prec\!\!- fs \\
&\quad \textbf{in} \\
&\quad concatMap\ (\lambda k \rightarrow (g \circ fst)\ k \prec\!\!- fs)\ ks)
\end{aligned}$$

But in the list monad $xs \gg\!= f = concatMap\ f\ xs$, so we can write this as:

$$\begin{aligned}
&f \overset{MR}{\gg\!=} g = MR\ (\lambda xs \rightarrow \\
&\quad \textbf{let} \\
&\quad\quad bs\ ys\ \ \ = f \succ\!\!- dedupe \prec\!\!- ys \\
&\quad\quad hs\ ys\ z = f \succ\!\!- (g \circ fst)\ z \prec\!\!- ys \\
&\quad \textbf{in} \\
&\quad bs\ xs \gg\!= hs\ xs)
\end{aligned}$$

where we have used the trivial identity $n \prec\!\!- (m \prec\!\!- l) \cong (m \succ\!\!- n) \prec\!\!- l$. Therefore

Proposition 1. The *MapReduce* monad is equivalent to $MapReduceT\ [\,]$ with

$$f \overset{MR}{\gg\!=} g \cong f \gg\!=_{dedupe} (g \circ fst)$$

Proof. The identity was proved above; the *return* functions are trivially identical. □

MapReduceT and the State Monad

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

Define the **Reader Monad**:

$$\textbf{data}\ Hom\ a\ b = H\ \{run :: (a \to b)\}$$

and make *Hom s* a monad by taking

$$\begin{aligned} return\ x &= H\ (const\ x) \\ f \overset{H}{\gg\!=} g\ \ &= H\ (\lambda x \to g'\ (f'\ x)\ x) \end{aligned}$$

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

$$\begin{aligned} &(\succ) :: Hom\ a\ b \to Hom\ b\ c \to Hom\ a\ c \\ &(\succ)\ f\ g = H\ \$\ (run\ g) \circ (run\ f) \\ &id :: Hom\ a\ a \\ &id = H\ id \end{aligned}$$

Now consider *MapReduceT* $(Hom\ s)\ b\ c$, which consists of functions $Hom\ s\ b \to Hom\ s\ c$.

Lemma 2. *There is a natural map*

$$hom :: Hom\ b\ c \to MapReduceT\ (Hom\ s)\ b\ c$$

Such that

$$(hom\ q)\ \gg\!=_{hom\ p}\ (hom\ r) \equiv hom\ \$\ (q \succ p) \overset{H}{\gg\!=} (q \succ r)$$

Proof. Define

$$\begin{aligned} hom\ f &= MR\ \$\ \lambda h \to H\ (run\ f) \circ (run\ h) \\ &= MR\ \$\ \lambda h \to h \succ f \end{aligned}$$

We have the following, easily proved, identities:

$$\begin{aligned} &hom\ \$\ f \succ g\ \equiv hom\ f \succ hom\ g \\ &run \circ hom\ f \equiv \lambda x \to x \succ f \end{aligned}$$

from which we can deduce that

$$
\begin{aligned}
(hom\ q)\ \gg\!=_{hom\ p}\ (hom\ r) &\equiv MR\ \$\ \lambda x \to (run \circ hom\ q \succ run \circ hom\ p) \prec x \\
&\quad \overset{H}{\gg\!=} \lambda y \to (run \circ hom\ q \succ run \circ hom\ (r\ y)) \prec x \\
&\equiv MR\ \$\ \lambda x \to run\ \$\ hom\ (q \succ p) \prec x \\
&\quad \overset{H}{\gg\!=} \lambda y \to run\ \$\ hom\ (q \succ (r\ y)) \prec x \\
&\equiv MR\ \$\ \lambda x \to (q \succ p \overset{H}{\gg\!=} \lambda y \to q \succ (r\ y)) \prec x \\
&\equiv MR\ \$\ \lambda x \to x \succ \$(q \succ p) \overset{H}{\gg\!=} \lambda y \to (q \succ r\ y) \\
&\equiv hom\ \$\ (q \succ p) \overset{H}{\gg\!=} (q \succ r)
\end{aligned}
$$

□

We can now prove:

Proposition 3. Given $f :: State\ s\ a$, $g :: a \to State\ s\ b$, and

$$p = H\ \$\ \lambda x \to (e, snd\ x) :: Hom\ (s, a)\ (s, a)$$

Then there exist $q :: Hom\ (s, a)\ (s, a)$ depending only on f and $r :: (s, a) \to Hom\ (s, a)\ (s, b)$ depending only on g such that

$$(hom\ q)\ \gg\!=_{hom\ p}\ (hom\ r) \equiv hom\ \$\ (H\ fst) \succ H\ (run\ \$\ f \overset{S}{\gg\!=} g)$$

Proof. Define

$$
\begin{aligned}
&m :: s \to (s, a) \\
&n ::\ a \to s \to (s, b)
\end{aligned}
$$

by

$$
\begin{aligned}
&f = state\ m \\
&g = \lambda x \to state\ (n\ x)
\end{aligned}
$$

In the state monad, the bind function $\overset{S}{\gg\!=}$ obeys

$$
\begin{aligned}
f \overset{S}{\gg\!=} g &\equiv state\ \$\ (\lambda s \to n\ (snd\ \$\ m\ s)\ (fst\ \$\ m\ s)) \\
&\equiv state\ \$\ run\ \$\ u \overset{H}{\gg\!=} v
\end{aligned}
$$

where

$$
\begin{aligned}
u &= H\ \$\ snd \circ m &&:: Hom\ s\ a \\
v\ x\ y &= n'\ x\ (m\ y) &&:: a \to Hom\ s\ (s, b) \\
n'\ x\ y &= H\ \$\ n\ x\ (fst\ y) &&:: a \to Hom\ (s, a)\ (s, b)
\end{aligned}
$$

Now pick an arbitrary value $e :: s$ (if s is a monoid we can take $e = mempty$, but the value is irrelevant). Define

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

so

$$\begin{aligned}
q\ x &= H\ \$\ (run\ f) \circ fst\\
r\ x\ y &= H\ \$\ run\ (g\ (snd\ x))\ (fst\ y)
\end{aligned}$$

and these functions are thus the natural extension of f and g to functions where all variables are of type (s, a). Now

$$\begin{aligned}
(q \rightarrowtail p) \overset{H}{\gg=} (q \rightarrowtail r) &\equiv \lambda x \to H\ \$\ (e, snd \circ m \circ fst\ x) \overset{H}{\gg=} \lambda x \to n'\ (snd\ x)\ (m \circ fst)\\
&\equiv \lambda x \to n'\ (snd \circ m \circ fst\ x)\ (m \circ fst\ x)\\
&\equiv (H\ fst) \rightarrowtail (u \overset{H}{\gg=} v)
\end{aligned}$$

And, putting this together with the identity proved above, we get

$$\begin{aligned}
(hom\ q)\ \gg=_{hom\ p}\ (hom\ r) &\equiv hom\ \$\ (q \rightarrowtail p) \overset{H}{\gg=} (q \rightarrowtail r)\\
&\equiv hom\ \$\ (H\ fst) \rightarrowtail (u \overset{H}{\gg=} v)\\
&\equiv hom\ \$\ (H\ fst) \rightarrowtail H\ (run\ \$\ f \overset{S}{\gg=} g)
\end{aligned}$$

□

This is a very complex expression, and one may object that the $\gg=_{hom\ p}$ join gives rise to a map $(s, a) \to (s, b)$ while the $\overset{S}{\gg=}$ join is a map $s \to (s, b)$. So let us unpick the type-structure of the two sides. On the left-hand side

$$\begin{aligned}
&hom\ q :: MapReduceT\ (Hom\ s)\ (s,a)\ (s,a)\\
&hom\ r :: (s,a) \to MapReduceT\ (Hom\ s)\ (s,a)\ (s,b)
\end{aligned}$$

so

$$(hom\ q)\ \gg=_{hom\ p}\ (hom\ r) :: MapReduceT\ (Hom\ s)\ (s,a)\ (s,b)$$

On the right-hand side

$$\begin{aligned}
&f :: State\ s\ a\\
&g :: a \to State\ s\ b
\end{aligned}$$

so

$$f \overset{S}{\gg\!=} g :: State\ s\ b \Rightarrow run\ \$\ f \overset{S}{\gg\!=} g :: s \to (s, b)$$

Now $fst :: (s, a) \to s$, which means that

$$H\ fst \succ H\ (run\ \$\ f \overset{S}{\gg\!=} g) :: Hom\ (s, a)\ (s, b)$$

and so

$$hom\ \$\ H\ fst \succ H\ (run\ \$\ f \overset{S}{\gg\!=} g) :: MapReduceT\ (Hom\ s)\ (s, a)\ (s, b)$$

and so the two sides do have the same type.

Conclusion

Implementation

We have implemented *MapReduce* and *MapReduceT* as simple libraries and implemented a simple test application using the standard word-count MapReduce algorithm (see [2]). It turns out that swapping out *MapReduce* and replacing it with *MapReduceT* [] has no effect on the output. Practical results therefore bear out the theoretical equivalence proved above.

Future directions

The obvious next step is to examine and analyse *MapReduceT* of other well known monads, e.g. *Maybe*. Given that the richness of MapReduce arises from *MapReduceT* applied to the list monad, this is obviously of some interest.

It is also worth examining the role of the choice of filtering function p. Taking $p \equiv dedupe$ on the list monad gave us classical MapReduce, but the condition that $p \circ return'\ x \equiv return'\ x$ simply states that p is the identity on singletons, which is not a particularly strong condition. Therefore there is the possibility of there being MapReduce-like algorithms for other choices of p.

Finally, the concept of the pseudo-monad is surely an interesting generalisation in and of itself.

References

[1] Julan Porter. Mapreduce as a monad. **The Monad Reader**, 18:pages 5–16 (2011). `http://themonadreader.files.wordpress.com/2011/07/issue18.pdf`.

[2] Distributed mapreduce project darcs repository. `http://code.haskell.org/MapReduce`.