Löb and möb: strange loops in Haskell

Every once in a while, people recognize strange loops in unexpected places. This is one of them, in the form of a Haskell function.

`loeb`

`loeb` is one of those functions in Haskell that are amazing, crazy, simple and complicated in equal parts.

1. The implementation is very simple to write.
2. The implementation is hard to understand.
3. It is easy to use.
4. It is explainable.

Implementation

Feeling smart? Let's change that, here's `loeb` :

```
loeb :: Functor f => f (f a -> a) -> f a
loeb x = go where go = fmap ($ go) x
```

That settles the first two points from above (the first one being accomplished using trial-and-error typechecker abuse).

What `loeb` does

Short version: `loeb` calculates a result in terms of itself, but with more crazy than what you felt when you first heard about recursion.

Long version: what is this thing useful for? It can be used to implement spreadsheet-like behaviour if the functor is `[]` :

Take a list of functions mapping lists to values, `fs :: [[a] -> a]` . These functions can each be applied to some other list `xs :: [a]` . For each function `f` in `fs` , applying it to `xs` results in a value `r` ; call this collection of values `rs` . This in code form reads

```
xs :: [a]
xs = [...]

fs :: [[a] -> a]
fs = [...]

rs :: [a]
rs = [ f xs | f <- fs ]  -- r = f xs
```

This computes `rs` out of the given lists `xs` and `fs` .

Now the crux: it is possible to not take `xs` as given, using `rs` instead. In other words, the `f` are applied to the list of results they produce.

```
fs :: [[a] -> a]
fs = [...]

rs :: [a]
rs = [ f rs | f <- fs ]
```

This of course relies heavily on laziness, as it computes `rs` in terms of itself. Instead of having `fs` as its own definition, let's supply it as a parameter to `rs` :

```
rs fs = [ f (rs fs) | f <- fs ]
```

and as it turns out, `rs = loeb` . Therefore, `loeb` takes a list of functions, and calculates the list of results they produce when applied to the list of results they produce. Strange? Check! Loop? You bet!

An example should make using it clearer:

```
fs = [ const 1
     , succ . (!! 0)
     , succ . (!! 1)
     , succ . (!! 2)
     ]
```

This describes a list where the list elements are defined in terms of the previous result value. `const 1` is the first element of the function list, and applied to the resulting list it is always `1` ; therefore the resulting list's first element is `1` . `succ . (!! 0)` applied to the resulting list can now be calculated: the indexing results in the previously calculated `1` , and `succ` makes it a `2` . The second result element will therefore be `2` . This pattern repeats itself, resulting in

```
loeb fs ==> [1,2,3,4]
```

The interesting part is that the order of the functions is not necessarily left-to-right. The list elements can be swapped around, as long as the circularity is still resolved (otherwise the function won't terminate):

```
fs = [ succ . (!! 1)
     , succ . (!! 3)
     , succ . (!! 0)
     , const 1
     ]

loeb fs ==> [3,2,4,1]
```

So this is like a spreadsheet, right? One cell's value is known, and the other cells refer to each other in some way. When the evaluation terminates, each cell has a defined value. In a sense this is like a generalization of a fixed point combinator.

Spreadsheets!

The lists mentioned above are a little like spreadsheets with only one line. But there are other functors closer to the real thing, arrays for example!

```
import Data.Array
import Data.List
import Control.Monad
import Text.Printf

loeb :: Functor f => f (f a -> a) -> f a
loeb x = go where go = fmap ($ go) x

-- Empty cell
e = val 0

-- Simple cell value
```

```haskell
val = const

-- VAT of a cell's contents (10 %)
vat ix = (* 0.1) . (! ix)

-- Sum of the values at a list of indices
sum' ixs = \arr -> foldl' (\acc ix -> acc + arr ! ix) 0 ixs

spreadsheet = listArray ((0,0), (4,4))
--      Prices | VAT        | Effective prices + total
      [ val 1,   vat (0,0),   sum' [(0,i) | i <- [0..1]],   e,   e
      , val 3,   vat (1,0),   sum' [(1,i) | i <- [0..1]],   e,   e
      , val 5,   vat (2,0),   sum' [(2,i) | i <- [0..1]],   e,   e
      , val 2,   vat (3,0),   sum' [(3,i) | i <- [0..1]],   e,   e
      ,     e,           e,   sum' [(i,2) | i <- [0..3]],   e,   e
      ]

printArr :: Array (Int, Int) Double -> IO ()
printArr arr =
      forM_ [0..4] $ \i -> do
            forM_ [0..4] $ \j ->
                  printf "%4.1f   " (arr ! (i,j))
            printf "\n"

main = printArr $ loeb spreadsheet
```

Run it! The output will be

```
 1.0    0.1    1.1    0.0    0.0
3.0    0.3    3.3    0.0    0.0
5.0    0.5    5.5    0.0    0.0
2.0    0.2    2.2    0.0    0.0
0.0    0.0   12.1    0.0    0.0
```

where in the first column you'll see the prices (declared using `val` above), the second column is the added tax to the price on its left, the third lists the effective price, and below the effective prices there's the total sum you have to pay in order to buy everything. Magic! :-)

moeb

`moeb` is the result of playing around with `loeb`'s definition: what if we abstract over the `fmap` too? First and foremost, it makes the type signature go crazy:

```haskell
-- [m]oeb = multi-loeb :-)
moeb :: (((a -> b) -> b) -> c -> a) -> c -> a
moeb f x = go where go = f ($ go) x
```

`loeb` can now be recovered as `moeb fmap`. But are there other parameters that are useful for `f`? Well,

```haskell
moeb id x = id ($ moeb id x) x
          = ($ moeb id x) x
          = x (moeb id x)

-- This has the same form as
-- fix f = f (fix f)
==> moeb id = fix
```

This shows how `moeb` is a generalization of `fix`.

There are other functions that can be used as parameter to `moeb` such as `traverse` and `foldMap`, but I don't know of any useful applications for them.