The comonadic smell of spreadsheets

Alessandro Candolini

September 12, 2023

D2 fx =B2*C2

	A	B	C	D	E
1	Fruit	Weight (kg)	Price / kg	Total price	% price
2	Apples	1.5	1.75	2.625	0.4738267148
3	Bananas	0.7	1.29	0.903	0.1629963899
4	Tangerines	0.6	2.5	1.5	0.2707581227
5					
6					
7					
8		Total	5.54		
9					
10					

Figure 1: Example of a 2D spreadsheet

H8 fx

	A	B	C	D	E	F
1	Fruit	Weight (kg)	Price / kg	Total price	% price	
2	Apples	1.5	1.75	=B2*C2	=D2/$C$8	
3	Bananas	0.7	1.29	=B3*C3	=D3/$C$8	
4	Tangerines	0.6	2.5	=B4*C4	=D4/$C$8	
5						
6						
7						
8		Total	=SUM($C$2:$C			
9						
10						

Figure 2: Formulas

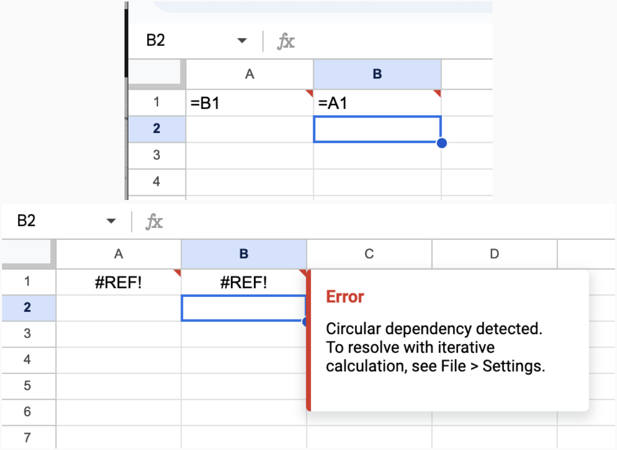

Figure 3: Circular dependencies

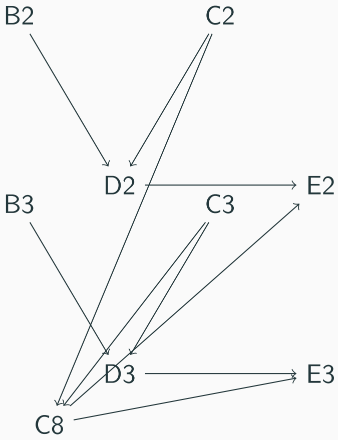

Figure 4: DAG

Associated to a spreadsheet is a DAG where

- nodes are cells
- there is a directed edge from a to b if and only if b has a formula that depends on a (i. e., b depends on a)

Spreadsheet *evaluation* is analogous to *dependency resolution*

A modern spreadsheet application provides many additional capabilities, not in scope in this talk, such as

- Parsing of cell formulas
- Built-in functions
- Graphics (e. g., histograms, piecharts, etc)
- Formatting, exporting, drag-and-drop, copy-pasting, etc
- Programmable functions (e. g., in VBA)
- Data connectors
- Collaborative online spreadsheets
- And many more

C3 fx 1.29

	A	B	C	D	E	F
1	Fruit	Weight (kg)	Price / kg	Total price	% price	
2	Apples	1.5	1.75	2.625	0.4738267148	
3	Bananas	0.7	1.29	0.903	0.1629963899	
4	Tangerines	0.6	2.5	1.5	0.2707581227	
5						
6						
7						
8		Total	5.54			
9						
10						

Figure 5: Recalculation and support graph

MVP / POC: recalculate the whole spreadsheet every time a cell changes

POST MVP: *incremental recalculation*

Traditional technique:

- Topological sorting of the support graph[1] (e. g., Kahn's algorithm)

[1] https://en.wikipedia.org/wiki/Topological_sorting

Production applications need to take into account more complicated challenges:[2]

- Performance tradeoffs (e. g., calculating the topological order also has an associated cost)
- Resource utilisation (e. g., compact representations)
- Volatile functions (e. g., `now()`, `rand()`)
- DAG caching
- Parallelisation algorithms, etc

[2] `https://learn.microsoft.com/en-us/office/client-developer/excel/excel-recalculation`

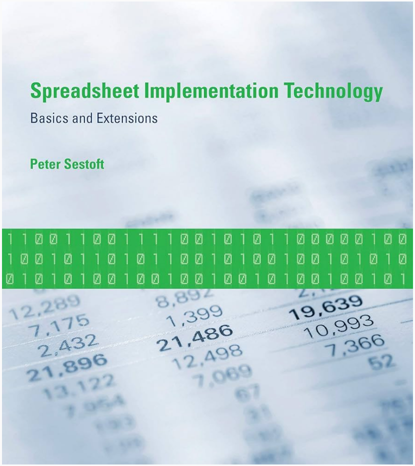

Figure 6: P. Sestoft, *Spreadsheet Implementation Technology*, MIT press (2014).[4]

[4] `https://direct.mit.edu/books/book/3071/Spreadsheet-Implementation-TechnologyBasics-and`

Observation

Spreadsheet-like evaluation can be expressed as a *fixed-point of higher dimensional comonads*

Agenda

1. A taste of comonads
2. Comonadic vibes meet spreadsheets
3. Down the rabbit hole

A taste of comonads

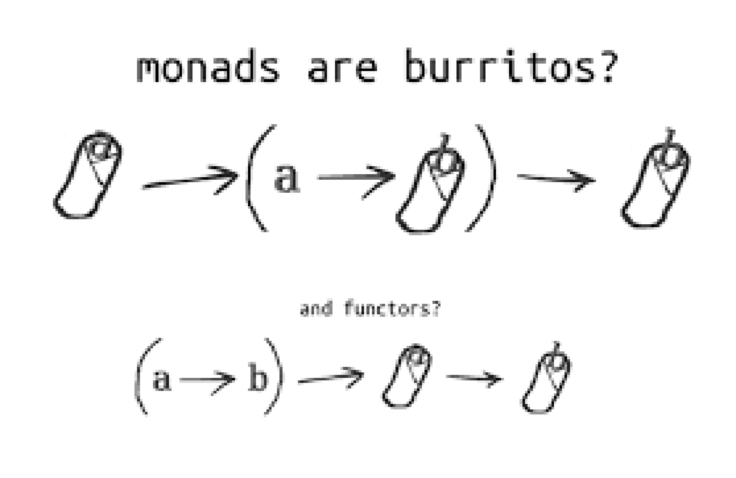

Figure 7: Monads are burritos

Definition

Comonads are co-burritos

Functor hierarchy:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

together with some laws.

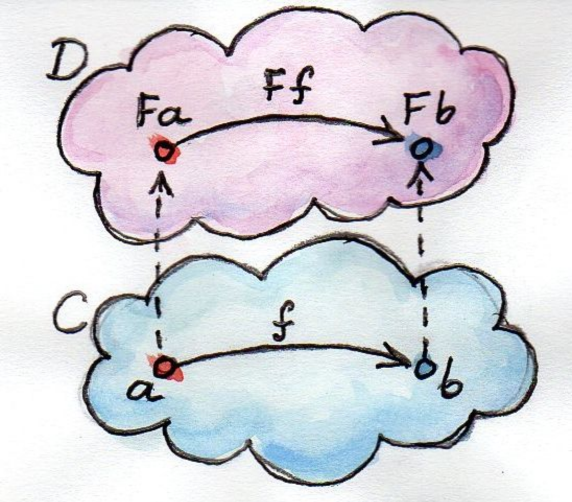

Figure 8: Functors: *Lifting* of 1-ary pure functions to the f-world

```haskell
f1 :: [Int] -> [String]

-- vs

f2 :: Int -> String

f3 :: [Int] -> [String]
f3 = fmap f2
```

```
class (Functor f) => Applicative f where
  pure :: Applicative f => a -> f a
  ap :: Applicative f => f (a ->b) -> f a -> f b
```

together with laws governing properties of `fmap`, `ap`, and `pure`.

Applicatives provide lifting of N-ary pure functions in the f-context

```haskell
class (Applicative f) => Monad f where
  return :: Monad f => a -> f a
  join :: Monad f => f (f a) -> f a

-- flatMap
bind :: Monad f => m a -> (a -> m b) -> m b
```

In short:

- Functor $\sim$ `map`
- Applicative $\sim$ `mapN` (or equivalent)
- Monad $\sim$ `flatMap` (or equivalent)

(and the caveat that the implementation must satisfies certain "reasonable" laws that rule out "unexpected" behaviours)

This seems very abstract but from a software engineer perspective lifting provides

- Separation of concerns
- Reusability
- Composability

`flatMap` ensures no “callback hell” of nested effects

```
f ( f ( f ( ... (f a)  ) )) = f a
```

Duality:

```
class Functor f => Comonad f where
    coreturn :: f a -> a
    cojoin :: f a -> f ( f a )
```

(and, as usual, there are important laws)

```haskell
class Functor w => Comonad w where
  extract :: f a -> a
  duplicate :: w a -> w ( w a )

-- coKleisli composition / extract
coFlatMap :: Comonad w => (w a -> b) -> w a -> w b
coFlatMap f = fmap f . duplicate
```

(and, as usual, there are important laws)

- Monadic values are typically *produced* in effectful computations

  ```
  a -> m b
  ```

- Comonadic values are typically *consumed* in context-sensitive computations (e. g., “queries”)

  ```
  w a -> b
  ```

`Maybe` and `[]` are not comonads (why?)

```
class Copointed f  where
   extract :: f a -> a
```

```haskell
instance Comonad NonEmpty where
  extract = head
  duplicate = tails1
```

```haskell
{-# LANGUAGE OverloadedLists #-}
-- return all suffices including itself

tails [1,2,3] `shouldBe`
       [[1,2,3], [2, 3], [3]]
```

```haskell
{-# LANGUAGE OverloadedLists #-}
coFlatMap f [1,2,3] `shouldBe`
       [f [1,2,3], f [2, 3], f [3]]

-- in this case
-- coFlatMap f = fmap f . tails
```

Simple moving average of a time series $[\phi_1, \dots, \phi_N]$ with window size k at the time t is

$$SMA_k(t) = \frac{1}{k} \sum_{j=t-k+1}^{t} \phi_j \tag{1}$$

```haskell
{-# LANGUAGE OverloadedLists #-}

sma :: Window -> NonEmpty Int -> [Double]

sma 3 [1,2,3,4,5,6,7,8,9,10] `shouldBe`
   [2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0]
```

because

- $(1+2+3)/3 = 6/3 = 2$
- $(2+3+4)/3 = 9/3 = 3$
- $(3+4+5)/3 = 12/3 = 4$

```haskell
smaLocal :: Window -> NonEmpty Double
   -> Maybe Double
smaLocal k s = (/ (fromIntegral k)) <$>
        sumK k s

sumFirstK :: Num a =>
   Window -> NonEmpty a -> Maybe a
sumFirstK (Window k) _ | k <= 0 = Nothing
sumFirstK (Window 1) (MyNonEmpty a _) = Just a
sumFirstK k (MyNonEmpty a as) =
   case nonEmpty as of
     Just as2 -> (+a) <$> sumFirstK (k-1) as
     Nothing -> Nothing
```

```
sma :: Window -> MyNonEmpty Double
         -> MyNonEmpty (Maybe Double)
sma size = coFlatMap (smaLocal size)
```

Folkloristic understanding of some comonads usages

- the function passed to `coFlatMap` represents a *local* computation (can explore the "neighbourhood" of the focus point, but produces a single value)
- `cojoin` produces a (lazy) view of the structure from all perspectives
- `coFlatMap` applies the local computation to all perspectives

Similar examples:

- Numerical derivation
- Reconciliation in a list of events
- etc

Interesting usages:

- Cellular automaton (e. g., Conway's game of life)
- Store, Moore machine and comonadic UIs

Comonadic vibes meet spreadsheets

```haskell
data Stream a = Stream a (Stream a)
  deriving (Eq, Show,Functor)

iterate :: (a -> a) -> a
  -> Stream a
iterate f seed =
  Stream seed (iterate f (f seed))

instance Comonad Stream where
   extract (Stream a _) = a
   duplicate s'@(Stream _ as) =
     Stream s' (duplicate as)
```

```
iterate :: (a -> a) -> a
  -> Stream a
iterate f seed =
  Stream seed (iterate f (f seed))
```

```haskell
data Sheet1 = Sheet1 (Stream a) a (Stream a)
   deriving (Eq,Show, Functor)
```

```haskell
focus :: Sheet1 a -> a
focus (Sheet1 _ a _ ) = a

moveL :: Sheet1 a -> Sheet1 a
moveL (Sheet1 (Stream a as) focus s)
   = Sheet1 as a (Stream focus s)

moveR :: Sheet1 a -> Sheet1 a
moveR (Sheet1 s focus (Stream a as))
   = Sheet1 (Stream focus s) a as
```

```haskell
instance Comonad Sheet1 where
  extract = focus
  duplicate s =
    Sheet1 (allLeft s) s (allRight s) where
      allLeft = iterate moveL
      allRight = iterate moveR
```

Duplicate is “diagonalisation” (as with other comonads derived from zippers)

```
naturals :: Sheet1 Integer
naturals = sheet 0 (repeat (cell left + 1))
```

Down the rabbit hole

README.md

Iterated Numbers

A one-dimensional sheet which is zero left of the origin and lists the natural numbers right of the origin:

```
naturals :: Sheet1 Integer
naturals = evaluate $ sheet 0 (repeat (cell left + 1))
```

When we print this out...

```
> take (rightBy 10) naturals
[1,2,3,4,5,6,7,8,9,10,11]
```

Pascal's Triangle

An infinite spreadsheet listing the rows of Pascal's triangle as upwards-rightwards diagonals:

```
pascal :: Sheet2 Integer
pascal = evaluate . sheet 0 $
   repeat 1 <:> repeat (1 <:> pascalRow)
   where pascalRow = repeat $ cell above + cell left
```

Notice the fact that I'm using the `(+)` function to add *functions* (namely, `cell above` and `cell left`). This is thanks to some clever overloading from `Data.Numeric.Function`.

This looks like:

```
> take (belowBy 9 & rightBy 9) pascal
[[1,  1,  1,   1,   1,    1,    1,     1,     1,     1],
 [1,  2,  3,   4,   5,    6,    7,     8,     9,    10],
 [1,  3,  6,  10,  15,   21,   28,    36,    45,    55],
 [1,  4, 10,  20,  35,   56,   84,   120,   165,   220],
 [1,  5, 15,  35,  70,  126,  210,   330,   495,   715],
 [1,  6, 21,  56, 126,  252,  462,   792,  1287,  2002],
 [1,  7, 28,  84, 210,  462,  924,  1716,  3003,  5005],
 [1,  8, 36, 120, 330,  792, 1716,  3432,  6435, 11440],
 [1,  9, 45, 165, 495, 1287, 3003,  6435, 12870, 24310],
 [1, 10, 55, 220, 715, 2002, 5005, 11440, 24310, 48620]]
```

About comonads:

- `https://bartoszmilewski.com/2017/01/02/comonads/`
- `https://blog.higher-order.com/blog/2015/10/04/scala-comonad-tutorial-part-2/`
- `https://://reasonablypolymorphic.com/blog/cofree-comonads/`
- `https://reasonablypolymorphic.com/blog/comonadic-physics/index.html`

About comonadic UIs:

- https://functorial.com/the-future-is-comonadic/main.pdf
- https://arthurxavierx.github.io/ComonadsForUIs.pdf

From *Löb's Theorem* in modal logic[5]

$$\Box\left(\Box\phi \to \phi\right) \to \Box\phi \tag{2}$$

to spreadsheet evaluation:

- Original post from Piponi (2006): `http://blog.sigfpe.com/2006/11/from-l-theorem-to-spreadsheet.html`
- Functional pearl paper from Kenneth Foner (2015): `https://dl.acm.org/doi/10.1145/2887747.2804310`
- ComonadSheet hackage package (unmaintained): `https://hackage.haskell.org/package/ComonadSheet`

[5] In this context, $\Box\phi$ is a formal provability predicate which reads "ϕ is provable"

Questions?