Evaluating cellular automata is comonadic

Dan Piponi

February 2024

This article originally appeared as a blog post. See there for the comments. It's possible some things I state are now out of date.

Paul Potts's post inspired me to say something about cellular automata too.

So here's the deal: whenever you see large datastructures pieced together from lots of small but similar computations there's a good chance that we're dealing with a comonad. In cellular automata we compute the value of each cell in the next generation by performing a local computation based on the neighbourhood of that cell. So cellular automata look like they might form a good candidate for comonadic evaluation.

I want to work on 'universes' that extend to infinity in both directions. And I want this universe to be constructed lazily on demand. One way of doing that is to represent a 'universe' as a centre point, a list of all elements to the left of that centre point and a list of all elements to the right. Here's a suitable type:

```
data U x = U [x] x [x]
```

For example `U [-1,-2..] 0 [1,2..]` can be thought of as representing all of the integers in sequence.

But this actually contains slightly more information than a list that extends to infinity both ways. The centre point forms a kind of focus of attention. We could shift that focus of attention left or right. For example

consider

```
U [-2,-3..] (-1) [0,1..]
```

This represents the same sequence of integers but the focus has been shifted left. So think of the type `U x` as being a doubly infinite sequence with a cursor. (In fact, this makes it a kind of zipper.)

We can formalise the notion of shifting left and right as follows:

```
right (U a b (c:cs)) = U (b:a) c cs
left  (U (a:as) b c) = U as a (b:c)
```

An object of type `U` is semantically like a `C` pointer into a const block of memory. You can increment it, decrement it and dereference it using the function I'll call `coreturn` below.

As `U` is a kind of list structure, it needs a map. In fact, we can define `fmap` for it:

```
instance Functor U where
   fmap f (U a b c) = U (map f a) (f b) (map f c)
```

Now the fun starts. First I'll bemoan the fact that `Comonads` aren't in the standard Haskell libraries (at least I don't think they are). So I have to define them myself:

```
class Functor w => Comonad w where
   (=>>)    :: w a -> (w a -> b) -> w b
   coreturn :: w a -> a
   cojoin     :: w a -> w (w a)
   x =>> f = fmap f (cojoin x)
```

`cojoin` is the dual to the usual `join` function. I've chosen to do things the category theoretical way and define `=>>` in terms of `cojoin`.

And here's why `U` forms a `Comonad`:

```
instance Comonad U where
   cojoin a = U (tail $ iterate left a)
                a
                (tail $ iterate right a)
   coreturn (U _ b _) = b
```

Look closely at `cojoin`. It turns a into a 'universe' of 'universes' where each element is a copy of a shifted left or right a number of times. This is where all the work is happening. The reason we want to do this is as follows: we want to write rules that work on the local neighbourhoods of our universe. We can think of a universe with the cursor pointing at a particular element as being an element with a neighbourhood on each side. For example, we can write a cellular automaton rule like this:

```
rule (U (a:_) b (c:_)) = not (a && b && not c || (a==b))
```

```
                  #
                  ##
                  # #
                  ####
                  #   #
                  ##  ##
                  # # # #
                  ########
                  #       #
                  ##      ##
                  # #     # #
                  ####    ####
                  #   #   #   #
                  ##  ##  ##  ##
                  # # # # # # # #
                  ################
                  #               #
                  ##              ##
                  # #             # #
                  ####            ####
```

In order to apply this everywhere in the universe we need to apply the rule to each possible shift of the universe. And that's what `cojoin` does, it constructs a universe of all possible shifts of a. Compare with what I said here. So believe it or not, we've already written the code to evaluate cellular automata. `u =>> rule` applies the rule to `u`. The rest is just boring `IO`:

```
shift i u = (iterate (if i<0 then left else right) u) !! abs i
```

```
toList i j u = take (j-i) $ half $ shift i u where
   half (U _ b c) = [b] ++ c

test = let u = U (repeat False) True (repeat False)
       in putStr $
          unlines $
          take 20 $
          map (map (\x -> if x then '#' else ' ') . toList (-20) 20) $
          iterate (=>> rule) u

main = test
```

Lazy infinite structures, comonads, zippers. I think I'm just beginning to get the hang of this functional programming lark! Over Xmas I might feel ready to try writing a piece of code longer than a dozen or so lines.

Anyway, I must end with a credit. I probably wouldn't have come up with this if I hadn't Comonadic functional attribute evaluation by Uustalu and Vene.