# Comonadic Evaluation

## applications in Cellular Automata

Madalina Sas (*madalina.sas@pm.me*)

Advanced Haskell

April 2020

github.com/mearlboro/hascell

# Outline

Motivation: Cellular Automata

The numbering system: Wolfram Codes

Worldbuilding: List Zippers

Modelling repeated computation: Comonads

Future work

# Haskell Topics

- Bitwise operations: `Data.Bits` and `Data.Bits.Bitwise`
- Integer to/from bitstream representation
- Functors and Lists: the `Functor` class
- List Zippers
- Monads: `Control.Monad`
- Comonads: `Control.Comonad`

# Useful Prerequisites

- Expressions, Functions, and Types
- Lists and List Comprehensions
- Higher-order Functions
- Currying and Partial Application
- Laziness
- Algebraic Data Types
- Classes and Instances
- Knowledge of Functors and Monads is recommended but not essential

# Recommended Reading

- Paul Hudak, *The Haskell School of Expression*
- Bryan O'Sullivan, Don Stewart, John Goerzen, *Real World Haskell*
- Learn You a Haskell for Greater Good, *Zippers*
- Learn You a Haskell for Greater Good, *A fistful of Monads*
- Stephen Wolfram, *Statistical mechanics of cellular automata*
- Stephen Wolfram, *A new kind of science*

The libraries are documented on Hackage:

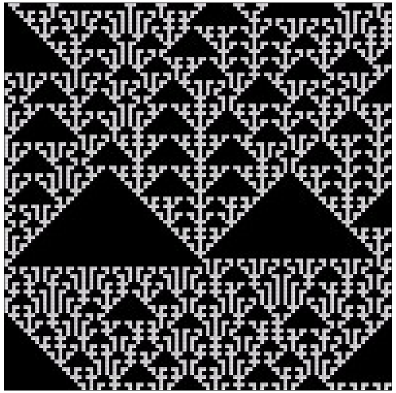
- `Data.Bits`
- `Data.Bits.Bitwise`
- `Control.Comonad`

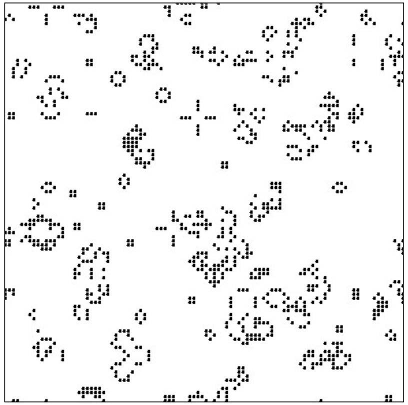
# Motivation

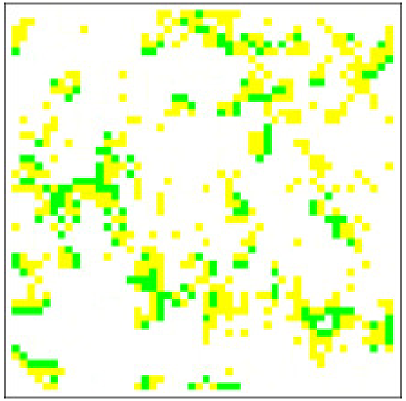
## Cellular Automata

# Cellular Automata (CA)

- A system of simple, spatially distributed, *identical* agents
- Follow rules of evolution over *discrete* time steps
- Usually interact based on their topology, i.e. the state of one cell is influenced by the state of *neighbouring* cells
- *Simple* models with *complex* dynamics e.g. chaotic behaviour
- Applications in encryption and computation theory due to their randomness or complexity







## Elementary

- 1-dimensional
- 2 states: *alive*, *dead*
- 4 ‘classes’ of behaviour

## Game of Life

- 2-dimensional
- 2 states: *alive*, *dead*
- Turing complete

## Excitable Medium

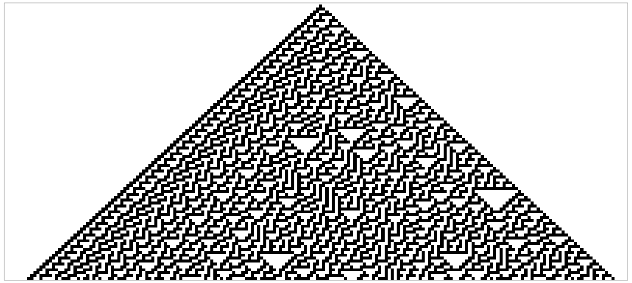
- 2-dimensional
- 3 states: *excitable*, *excited*, *refractory*
- Brains, hearts, forest fires

# Elementary Cellular Automata (ECA)

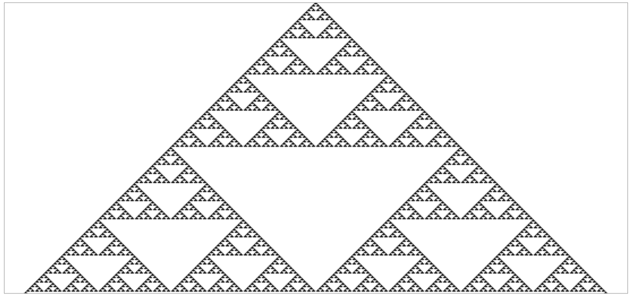
- Courtesy of Stephen Wolfram
- The simplest cellular automaton:
  - 1 dimension
  - 2 possible states: 0 and 1
  - each cell has 2 neighbours: evolution rules operate with 3 cells at a time

- Don't be fooled by its simplicity...

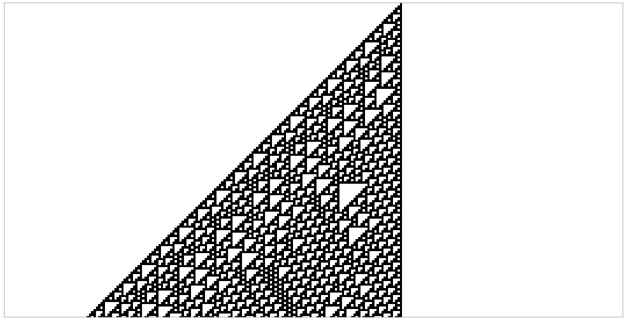
- Some ECA are so non-periodic and chaotic they can be used to generate random numbers for encryption: rule 22, 30, 86, 135

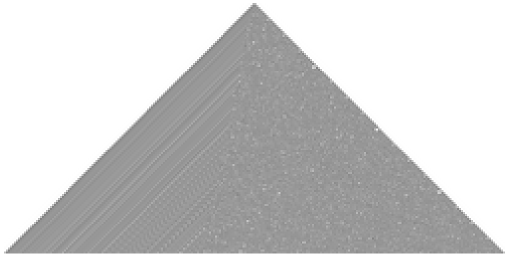


- Some are fractal: rule 90 starting from a single live cell is the Sierpinski triangle. Other examples are rule 129, 146, 150, 153

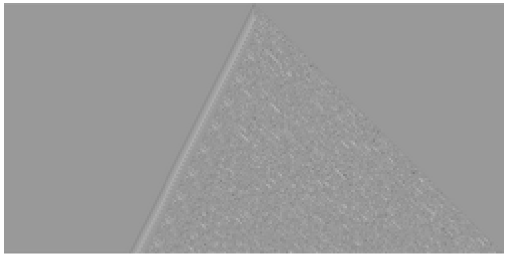


- Some live between order and chaos: rule 110, 124, 137 can be used to simulate any possible algorithm, like a Turing machine

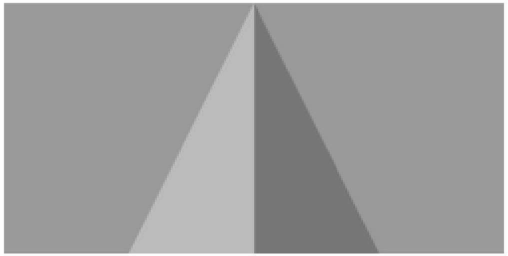




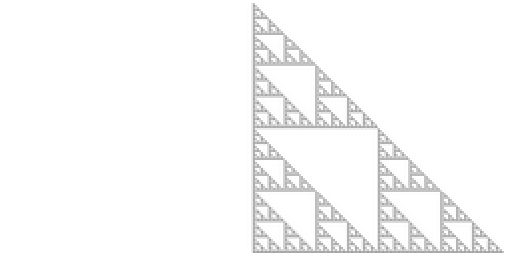
Rule 30



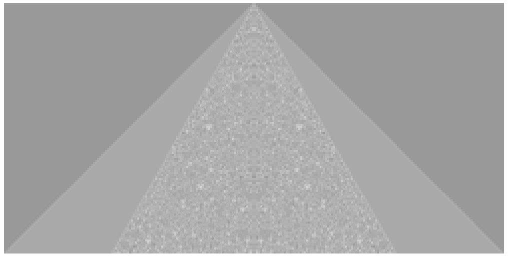
Rule 45



Rule 57



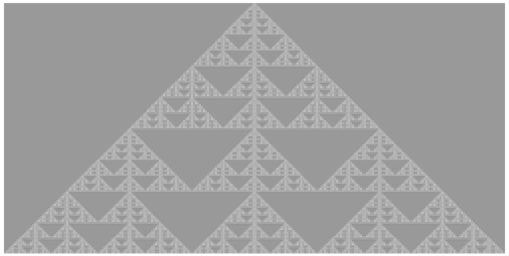
Rule 60



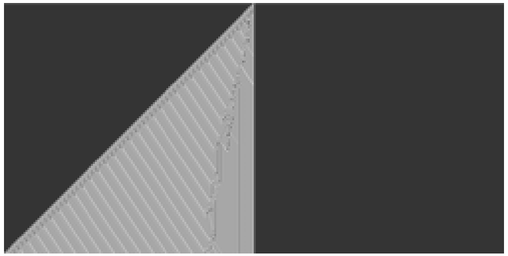
Rule 73



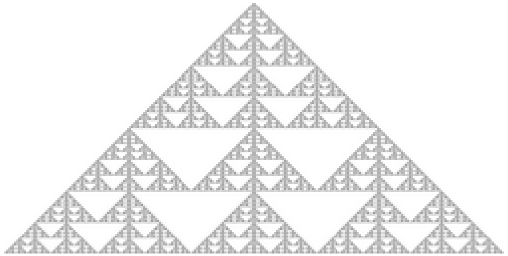
Rule 90



Rule 105



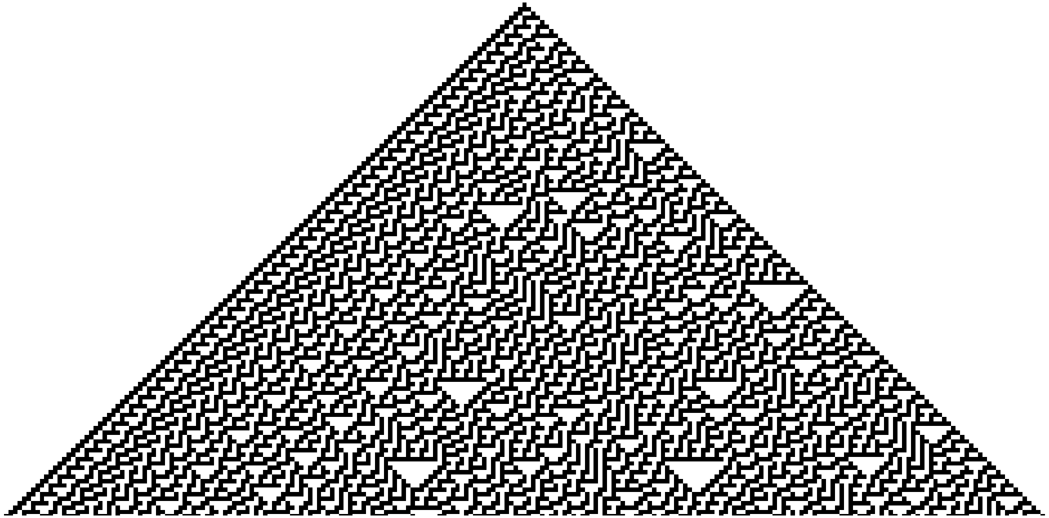
Rule 137



Rule 150

* the plots above have been generated using the code presented in this lecture

# Created or Discovered?



Rule 30

*Conus Textile* shell

# The Numbering System

## Wolfram Codes

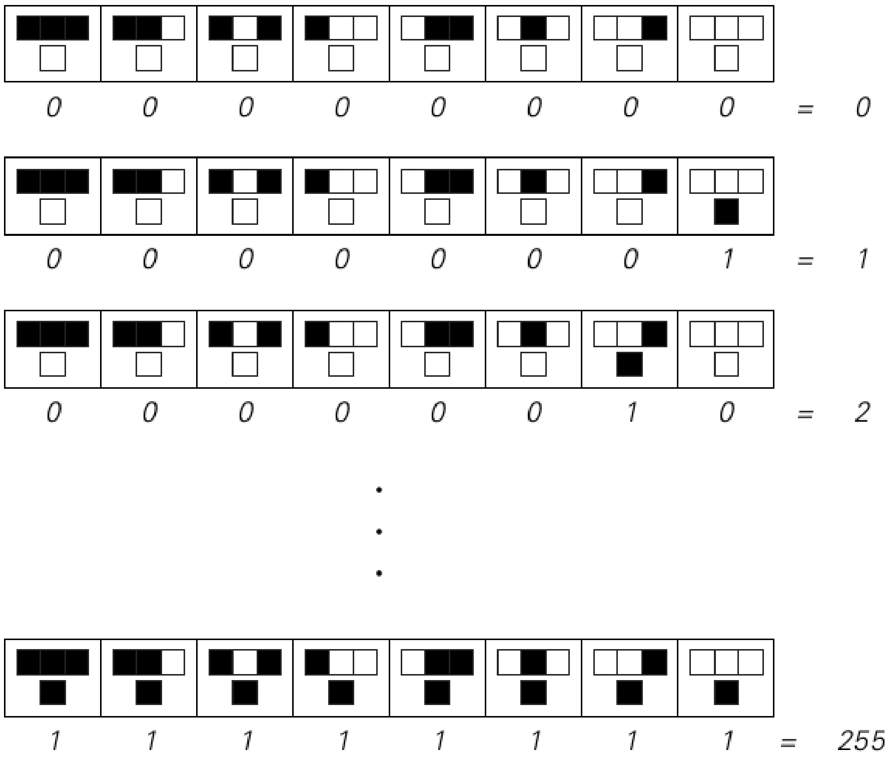
# Wolfram Codes

- A system of generating all possible CA rules for this configuration
- For each cell $n$ in generation $G$, its value is computed based on the values itself and its neighbours had in the previous generation

$$\texttt{val}(n,\ G) = \texttt{f}(\texttt{val}(n\text{-}1,\ G\text{-}1),\ \texttt{val}(n,\ G\text{-}1),\ \texttt{val}(n+1,\ G\text{-}1))$$

- $2^{2^3}=2^8=256$ possible functions $\texttt{f}\colon \{0,1\} \times \{0,1\} \times \{0,1\} \rightarrow \{0,1\}$
- The corresponding Wolfram Code is the 8-bit number with the binary expansion that represents `f`

The sequence of 256 possible cellular automaton rules of the kind shown above. As indicated, the rules can conveniently be numbered from 0 to 255. The number assigned is such that when written in base 2, it gives a sequence of 0's and 1's that correspond to the sequence of new colors chosen for each of the eight possible cases covered by the rule.



# Implementation: List Comprehension

- First try: list comprehension

```
wolframRule :: Int -> [Int]
wolframRule r = [ (r `div` 2^i) `mod` 2 | i <- [0..7] ]
```

- What about datatypes?
  - an `Int` is much bigger than 8-bit word we need (Tip: `Data.Word`)
  - the result is a list of 0 and 1

```
wolframRule :: Word8 -> [Bool]
wolframRule r = [(r `div` 2^i) `mod` 2 != 0 | i <- [0..7]]
```

# Implementation: Binary expansions

- How do we *elegantly* turn a 1-bit `Int` into `Bool`? The answer is `Data.Bits`. Given a number (expressed as an array of bits) and an integer $n$, `testBit` returns the value of the $n$th least significant bit

```
testBit :: Bits a => a -> Int -> Bool
```

- There exists a `Bits` instance of `Word8`, which allows us to use `Word8` directly with `testBit`. The `:i` command will show you all instances of a datatype

```
λ Data.Word> :i Word8
instance Bits Word8 -- Defined in ‘GHC.Word’
```

```haskell
wolframRule :: Word8 -> [Bool]
wolframRule r = [ testBit r i | i <- [0..7] ]
```

- The list comprehension is better expressed as a `map`
- We already know how many bits the Wolfram Code `r` has from its data type, so the magic number `7` is redundant

```haskell
wolframRule r = map (testBit r) [0..finiteBitSize r-1]
```

- `finiteBitSize :: FiniteBits b => b -> Int` returns the number of bits required to represent its input argument

# Worldbuilding

## List Zippers

# 1-Dimensional Universe

- An infinite line made of discrete ‘points’ or cells
- We only care about a *finite subset* of our universe, so we can be *lazy*
- We could use an *infinite list*, but then we’d have to *traverse* it
- For every computational step, *focus* is on 3 cells only. All computations are *local*
- Interested in the idea of *local context*, rather than global context; *relative* positioning rather than absolute positioning

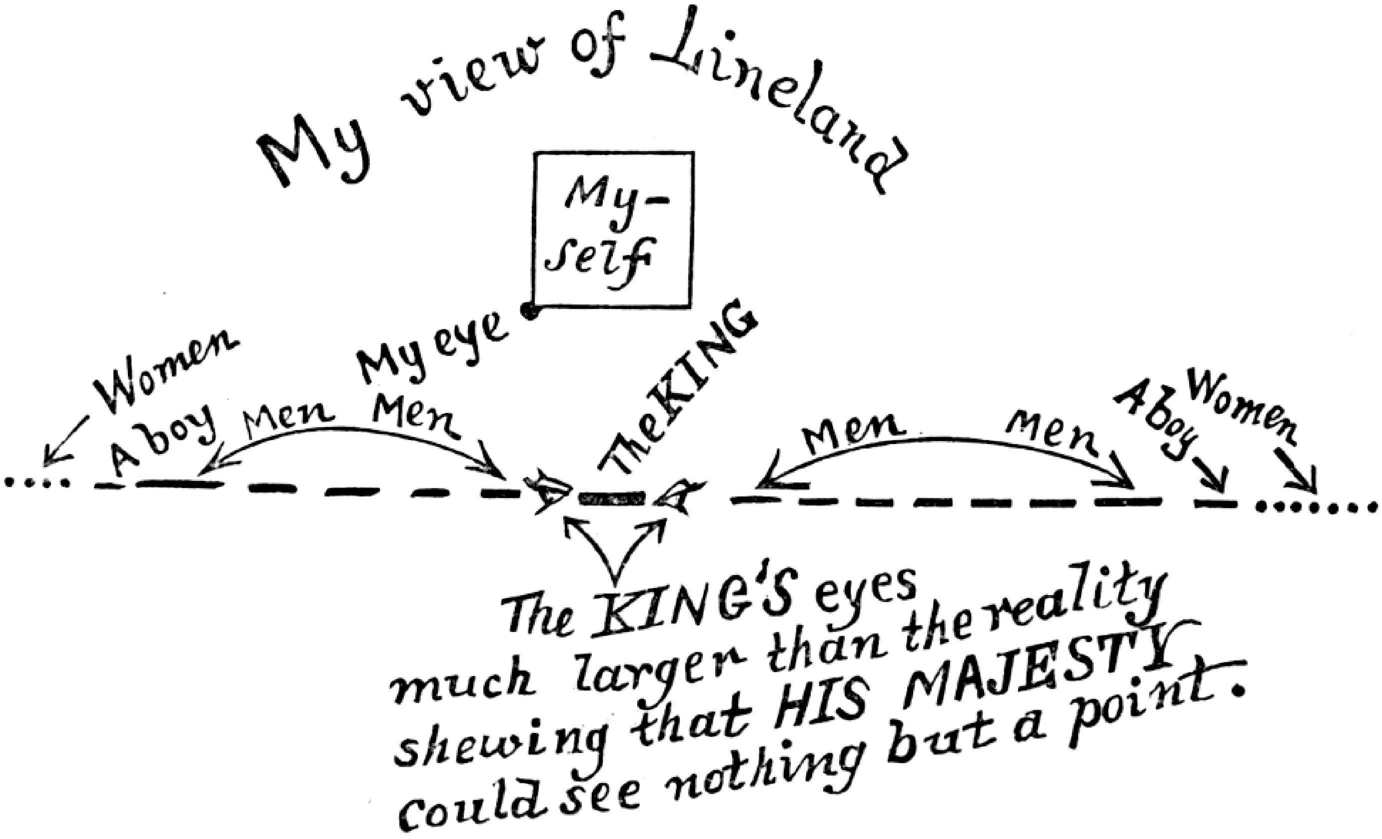



illustration © *Flatland,* Edwin C. Abbot

# Local Computations

- Can we write the following global computation as a local computation?

$$\texttt{val}(n,\ G) = \texttt{f}_{\texttt{r}}(\texttt{val}(n\text{-}1,\ G\text{-}1),\ \texttt{val}(n,\ G\text{-}1),\ \texttt{val}(n+1,\ G\text{-}1))$$

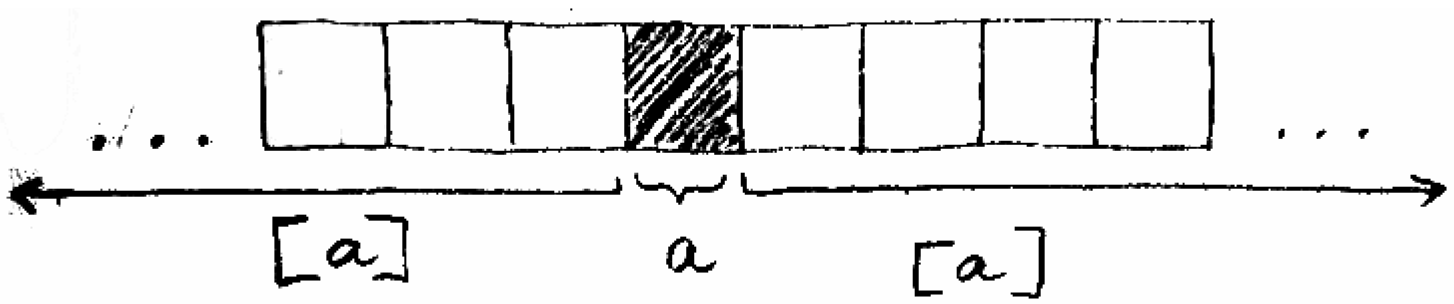
- Considering a *focus cell*, `c`, and generation *G:*

$$\texttt{c}_G = \texttt{f}_{\texttt{r}}(\texttt{left}(\mathrm{c}_{G\text{-}1}),\ \mathrm{c}_{G\text{-}1},\ \texttt{right}(\mathrm{c}_{G\text{-}1}))$$

# Zippers

- A *zipper* is an idiom that uses the idea of *context* to the means of manipulating locations in a data structure
- Idea: a *list* zipper would have a focus on a certain element and have two sub-lists, one to its left, one to its right

```
data W a = W [a] a [a]
```

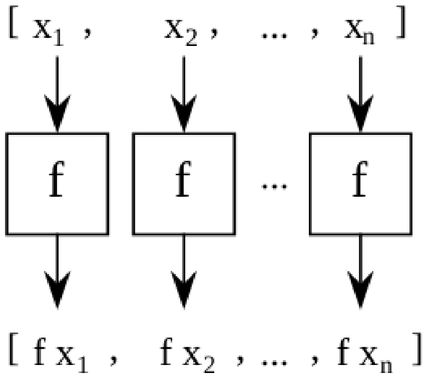


# Navigating Zippers

- Need to locally navigate the data structure
- Jump left or right, get back the data structure with the *focus element* shifted in the respective direction

```
data W a = W [a] a [a]
left, right :: W a -> W a
left  (W (l:ls) x    rs ) = W    ls  l (x:rs)
right (W    ls  x (r:rs)) = W (x:ls) r rs
```

# Functors




- Remember functors?

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

- Functors represent types that can be mapped over
- Must preserve identity and composition

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

# List Zippers are Functors

- Lists are functors:

```
instance Functor [] where
  fmap = map
```

- Since list zippers are lists with a focus element, functions can be mapped over the list zipper `W` using `fmap`, so they are functors too

```
instance Functor W where
  fmap f (W ls x rs) = W (fmap f ls) (f x) (fmap f rs)
```

- `fmap` is needed to apply our evolution rules over each cell

# Working with Context

- Need a way to extract the focus element from the zipper

```
extract :: W a -> a
extract (W _ x _) = x
```

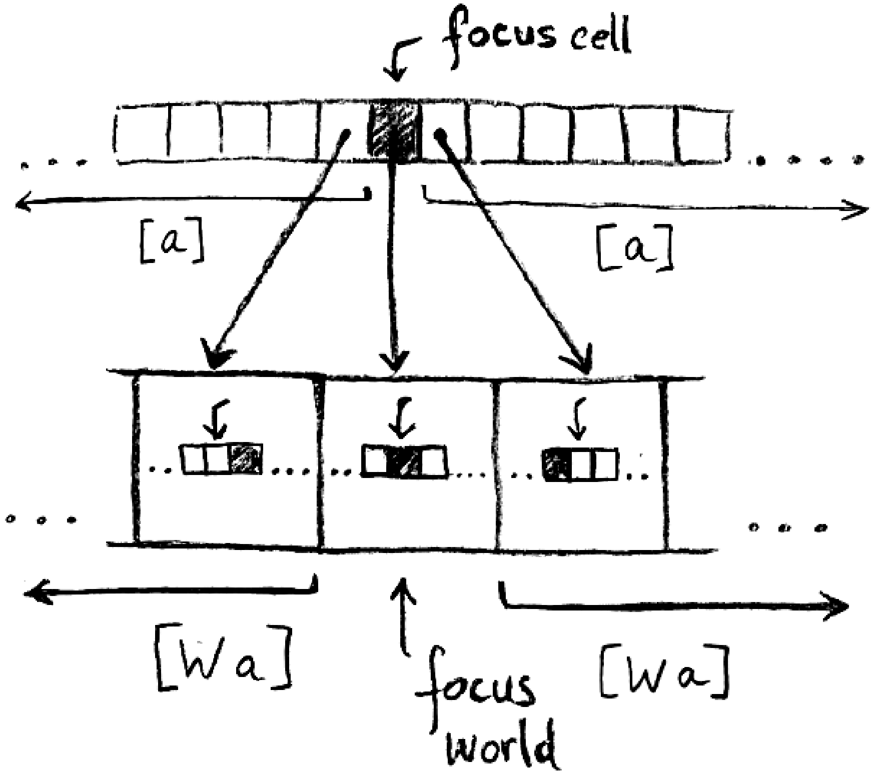
- Evolution rules have the same type: take a zipper with the current generation of cells, return the next state of the specific cell that is the focus element
- After applying a rule, the focus cell is taken *out of context*. Need to put it back without losing the information about the other cells.

- For each cell: look-behind at a zipper and compute a new value
- For each generation: look-behind at a zipper of zippers, by changing the focus element to every cell in the zipper, and compute a new zipper
- Idea: a function to wrap the context into *another context*
- The aim is to obtain the `id` function when composing the two functions

```
extract :: W a -> a
wrap    :: W a -> W (W a)
id      :: W a -> W a
```

```
extract . wrap = id
wrap . extract = id
```

- `wrap` creates a *zipper of zippers*:
  - The *focus element* is the original zipper, with its focus element set
  - The left and right lists are made of copies of the original zipper by *repeatedly shifting the focus element left and right*




```
wrap :: W a -> W (W a)
wrap w = W (tail (iterate left w)) w (tail (iterate right w))
```

```
extract ::  W a -> a
wrap    ::  W a -> W (W a)
```

- Using these two functions, we can now `apply` a function `rule` to the zipper and get back also a zipper

```
rule    ::  W a -> a
apply   :: (W a -> a) -> W a -> W a
```

- Take a rule and a zipper that represents the current generation, get a zipper that represents the next generation:

```
apply rule w = fmap rule (wrap w)
```

# Adapting Rules

- Any rule can be applied on a 8-bit number using its Wolfram Code `r`

```
wolframRule r = map (testBit r) [0..finiteBitSize r-1]
```

- To apply it to a zipper `w`, construct the 8-bit number represented by the focus cell and its neighbours

```
wolframRule r w = testBit r (2^0 * lc + 2^1 * cc + 2^2 * rc)
  where
    cc = fromEnum (extract w)
    lc = fromEnum (extract (left  w))
    rc = fromEnum (extract (right w))
```

- Need a function like the ‘opposite’ of `testBit` that returns an integer given its binary expansion. Found in `Data.Bits.Bitwise`

```
λ Data.Bits Data.Bits.Bitwise> :t fromListBE
fromListBE :: Bits b => b -> Int
```

- `extract` from the zipper in which the current cell is in focus and the zippers in which its two neighbours are in focus: `left w, w, right w`
- The result of `extract` is a list of `Bool` to pass to `fromListBE`

```
wolframRule :: Word8 -> W Bool -> Bool
wolframRule r w
  = testBit r (fromListBE (map extract [left w, w, right w]))
```

- `wolframRule` can now be used with `apply` to create the next generation

```
generation :: Word8 -> W a -> W a
generation r w = apply (wolframRule r) w
```

- Can repeat the computation as many times we want, and every time it returns a zipper. Take the first `g` computations and get a list of zippers that represent all generations [0, 1, ... g-1]

```
experiment :: Word8 -> W a -> Int -> [W a]
experiment r w g
  = take g (iterate (generation r w))
```

# Infinite Laziness

- Our one-dimensional world is lazily generated. An initial world, with a single living cell in the middle, can be (lazily) defined as follows:

```
wolframWorld :: W a
wolframWorld = (repeat False) True (repeat False)
```

- `experiment` produces a list of zippers, but we must truncate them before attempting to print

```
truncateD :: Int -> W a -> W a
truncateD d (W ls x rs) = W (take d ls) x (take d rs)
```

# Modelling Repeated Computation

## Comonads

# Monads

- Remember monads?

```
class Monad m where
  return  ::  a -> m a
  (>>=)   ::  m a -> (a -> m b) -> m b
```

- A monad encapsulates a value (or values) `a` inside a context `m`
- The only way to access the value inside is through a continuation, that is, by *binding* it to an operation that accepts a value and produces an encapsulated value

# Monads are Functors too

- All monads are *functors*. To construct a monad from a functor:

```
class Functor m => Monad m where
  join   :: m (m a) -> m a
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

- Now bind `(>>=)` can be defined in terms of `fmap` and `join`:

```
ma >>= f = join (fmap f ma)
```

# Comonads

- Compare the list zipper with the monad:

```
extract::  W a -> a                  return :: a -> m a
wrap   ::  W a -> W (W a)            join   :: m (m a) -> m a
apply  :: (W a -> a) -> W a -> W a   (>>=)  :: m a -> (a -> m b) -> m b
```

- The list zipper above is the opposite (or *categorical dual)* of a monad, and is called a *comonad*
- The comonad puts forward the value it contains, and requires a continuation to access the rest of its context, by *extending* it with an operation that takes an encapsulated value and produces a value

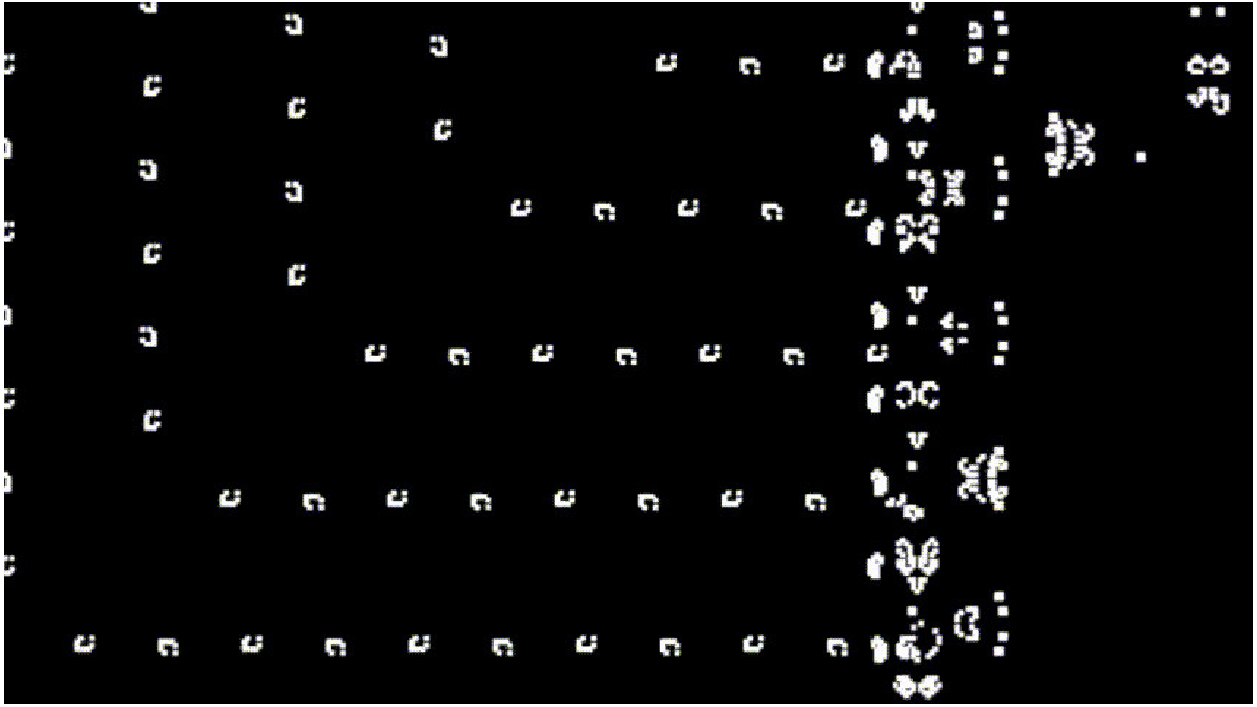
# Closing Thoughts

- The comonad lives in the `Control.Comonad` package
- Its 'official' function names are `extract`, `duplicate` (for `wrap`) and `extend` (for `apply`)
- Its full definition derives a `Functor`
- There are many possible instances of a comonad, which are more efficient than infinite lists

# Possible Improvements?

- Some cellular automata live on toroidal worlds, which are not supported by a stream-like infinite list zipper
- Lists need to be traversed in order to save the results of an experiment, but lists are very inefficient to index – $O(n)$
- Since the computation is always local, it could be done in parallel
- What would the automaton look like if it were started from a more random initial configuration?
- How would a list zipper extend to 2 dimensions? Can we use it to implement Game of Life?
- Can we create a comonad for any number of dimensions?

# WHAT IS LIFE?



John Conway's *Game of Life*

In memoriam of John Horton Conway, FRS

26 December 1937 – 11 April 2020