# Lazy streams with $O(1)$ access

Andrew Lelechenko
1@dxdy.ru

Barclays, London

London Haskell, 25.02.2020

# Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89...

$$F_0 = 0,$$
$$F_1 = 1,$$
$$F_{n+2} = F_{n+1} + F_n.$$

or

$$F_n = \begin{cases} n, & n < 2, \\ F_{n-1} + F_{n-2}, & \text{otherwise.} \end{cases}$$

```
fib :: Word -> Integer
fib n
  | n < 2      = toInteger n
  | otherwise = fib (n - 1) + fib (n - 2)
```

# Clever solution

$$F_n = \begin{cases} n, & n < 2, \\ F_{n-1} + F_{n-2}, & \text{otherwise.} \end{cases}$$

```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+)
                    fibs (tail fibs)

fib :: Word -> Integer
fib = genericIndex fibs
```

# How it works? fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

$$fibs = 0 : 1 : \bot$$
$$tail\ fibs = 1 : \bot$$

$$fibs = 0 : 1 : 1 : \bot$$
$$tail\ fibs = 1 : 1 : \bot$$

$$fibs = 0 : 1 : 1 : 2 : \bot$$
$$tail\ fibs = 1 : 1 : 2 : \bot$$

$$fibs = 0 : 1 : 1 : 2 : 3 : \bot$$
$$tail\ fibs = 1 : 1 : 2 : 3 : \bot$$

$$fibs = 0 : 1 : 1 : 2 : 3 : 5 : \bot$$
$$tail\ fibs = 1 : 1 : 2 : 3 : 5 : \bot$$

# Foobonacci numbers: 0, 1, 1, 3, 5, 11, 21, 43, 85, 171...

$$\mathcal{F}_n = \begin{cases} n, & n < 2, \\ \mathcal{F}_{n-1} + 2\mathcal{F}_{n-2}, & \text{otherwise.} \end{cases}$$

```
foos :: [Integer]
foos = 0 : 1 : zipWith (\a b -> a+2*b) foos (tail foos)

> take 10 foos
[0,1,2,5,12,29,70,169,408,985]

foos :: [Integer]
foos = 0 : 1 : zipWith (\a b -> 2*a+b) foos (tail foos)
```

# Barbonacci numbers: 0, 1, 2, 4, 11, 25, 59, 142, 335, 796...

$$B_n = \begin{cases} n, & n < 3, \\ B_{n-1} + 2B_{n-2} + 3B_{n-3}, & \text{otherwise.} \end{cases}$$

```
bars :: [Integer]
bars = 0 : 1 : 2 : zipWith3 (\a b c -> 3*a+2*b+c)
                            bars (tail bars) (drop 2 bars)

bar :: Word -> Integer
bar n
  | n < 3      = toInteger n
  | otherwise = bar (n-1) + 2 * bar (n-2) + 3 * bar (n-3)
```

# Catalan numbers: 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862...

Richard P. Stanley,
*Catalan numbers*, CUP, 2015

$C_0 = 1$

$C_1 = 1$ ()

$C_2 = 2$ (()) ()()

$C_3 = 5$ ((())) (())() ()(()) (()()) ()()()

$$C_0 = 1,$$

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-k-1}.$$

# Very clever and not so clever solutions

$$C_0 = 1,$$
$$C_n = \sum_{k=0}^{n-1} C_k C_{n-k-1}.$$

```
cat :: Word -> Integer
cat 0 = 1
cat n = sum $ map (\k -> cat k * cat (n-k-1)) [0..n-1]

cats :: [Integer]
cats = 1 : map (\xs -> sum $ zipWith (*) xs (reverse xs))
               (tail $ inits cats)

cats :: [Integer]
cats = 1 : map (\n -> sum $
  map (\k -> cats !! k * cats !! (n-k-1)) [0..n-1]) [1..]
```

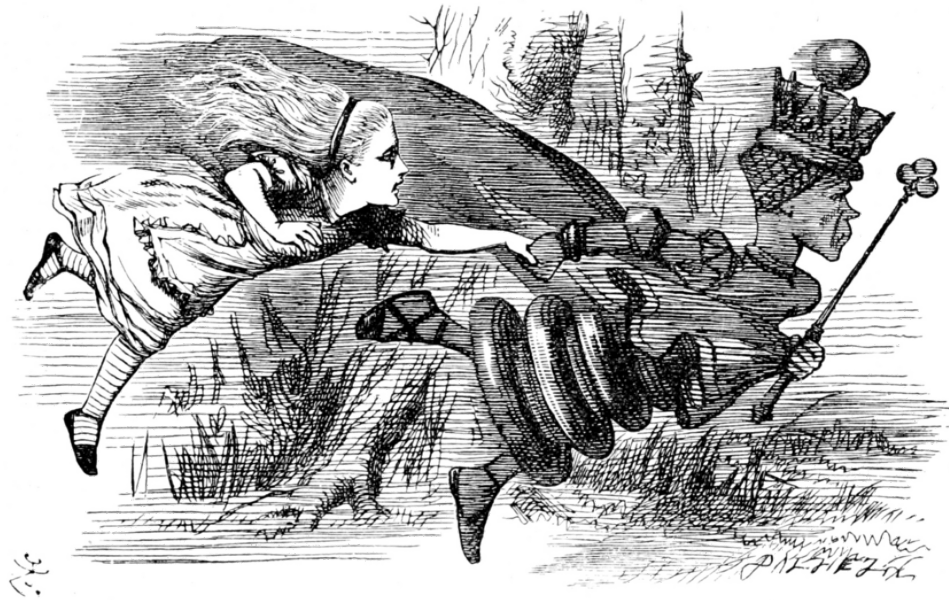
# Dynamic programming

## Problem

*Alice is travelling over a one-dimensional chess board, starting from the origin. The Black Queen can move Alice by 1, 4, 9, 16... cells, but only ahead. Each cell n contains $f(n)$ cakes. For a given N find the maximum number of cakes $V(N)$, which Alice can collect on her way to $N^{th}$ cell.*

Through the looking-glass a number of cakes may be negative!

$$V(0) = f(0)$$
$$V(n) = f(n) + \max_{1 \le k \le \sqrt{n}} V(n - k^2)$$



# Vectors to the rescue

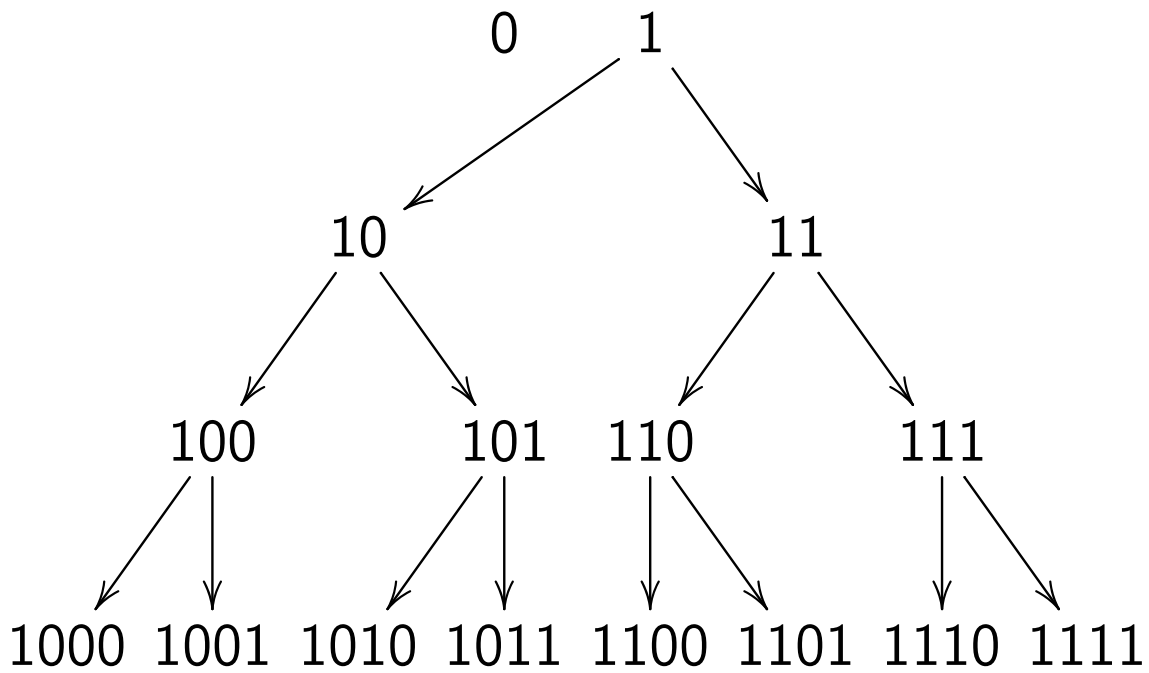
```
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fibs :: [Integer]
fibs = map (\n -> if n < 2
                  then toInteger n
                  else fibs !! (n-1) + fibs !! (n-2))
           [0..]

import qualified Data.Vector as V
fibs :: V.Vector Integer
fibs = V.generate 100 $ \n ->
  if n < 2
  then toInteger n
  else fibs V.! (n-1) + fibs V.! (n-2)
```

# Between lists and vectors

# Lazy binary tree




```
data Tree a = Tree (Tree a) a (Tree a)
```

# Lazy binary tree

```
data Tree a = Tree (Tree a) a (Tree a)

tabulate :: (Word -> a) -> (a, Tree a)
tabulate f = (f 0, go 1) where
  go n = Tree (go (2 * n)) (f n) (go (2 * n + 1))

index :: (a, Tree a) -> Word -> a
index (z, _) 0 = z
index (_, tree) n = go tree n where
  go (Tree _ m _) 1 = m
  go (Tree l _ r) k =
    go (if even n then l else r) (k `quot` 2)

fib :: Word -> Integer
fib n = if n<2 then toInteger n else fib'(n-1) + fib'(n-2)


fib' :: Word -> Integer
fib' n = index (tabulate fib) n
```

# Fixed-point combinator

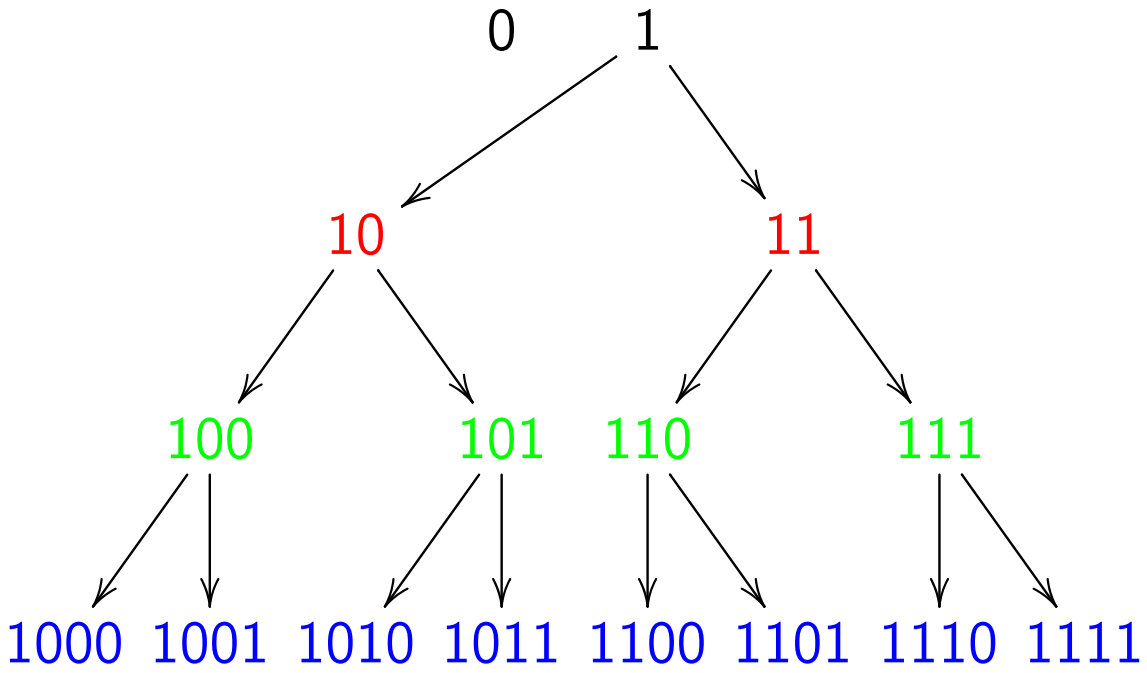
```
fix :: (a -> a) -> a
fix f = f (fix f)

fibF :: (Word -> Integer) -> Word -> Integer
fibF f n = if n<2 then toInteger n else f (n-1) + f (n-2)

tabulate :: (Word -> a) -> (a, Tree a)
tabulate f = (f 0, go 1) where
  go n = Tree (go (2*n)) (f n) (go (2*n+1))

tabulateFix :: ((Word -> a) -> Word -> a) -> (a, Tree a)
tabulateFix f = res where
  res = (f 0, go 1)
  go n = Tree (go (2*n)) (f (index res) n) (go (2*n+1))
```

# Chimera = Vector (Vector a)




$$[[0],[1],[10,11],[100,101,110,111],$$
$$[1000,1001,1010,1011,1100,1101,1110,1111]]$$

# Magic and its exposure

```
newtype Chimera a =
  Chimera (Vector (Vector a))
```

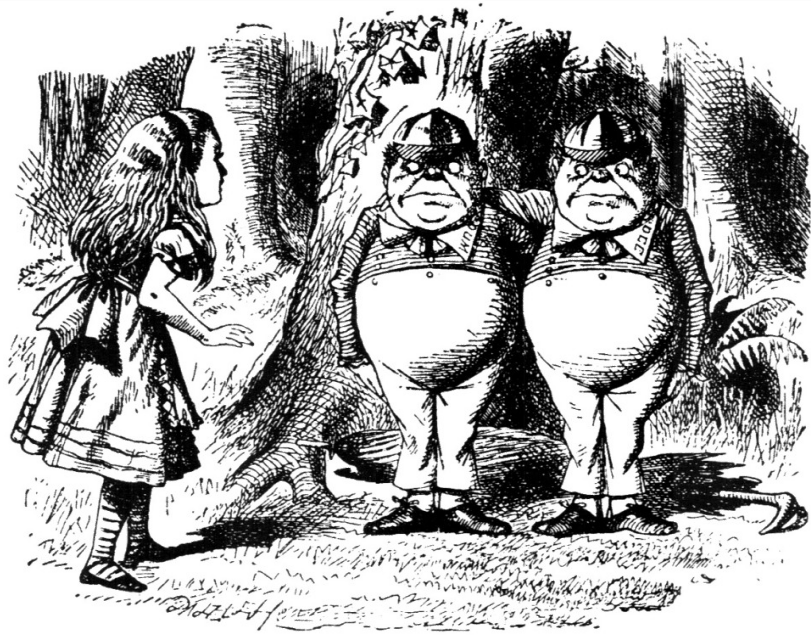
A boxed outer `Vector` points to 65 inner `Vector`s of growing sizes

$$1, 1, 2, 2^2, 2^3, \ldots, 2^{62}, 2^{63}.$$

| $f(0)$ | $f(1)$ | $f(2)$ | $f(3)$ |
|---|---|---|---|
| | | $f(4)$ | $f(5)$ | $f(6)$ | $f(7)$ |

Inner `Vector`s are allocated on-demand, resembling dynamic arrays in imperative languages, but caught in amber.

# Bit tweedling



```
import Data.Vector (Vector, (!))
import qualified Data.Vector as V

data Chimera a =
  Chimera (Vector (Vector a))

tabulate :: (Word -> a) -> Chimera a
tabulate f = Chimera $ V.singleton (f 0) `V.cons`
  V.generate 64 (\k -> V.generate (2^k) (f . (+ 2^k)))

index :: Chimera a -> Word -> a
index (Chimera xss) 0 = xss ! 0 ! 0
index (Chimera xss) n = xss ! (64-clz) ! (n - 2^(63-clz))
  where
    clz = countLeadingZeros n
```

# Representable functor for `Word`

$$\mathrm{Chimera}(a) \cong \mathrm{Hom}(\mathrm{Word}, a)$$

$$\mathrm{Chimera} \cong \mathrm{Hom}(\mathrm{Word}, \cdot)$$

```
import Data.Functor.Rep
instance Representable Chimera where
  type Rep Chimera = Word
  tabulate = tabulate
  index = index
```

GHC can derive `Functor`, `Foldable`, `Traversable`.
And `adjunctions` package is able to provide `Applicative`, `Monad`, `MonadFix`, `MonadZip`, `Distributive`.

# Cellular automata 1D

**Rule 90.** An infinite in both directions line of cells, each being dead (`False`) or alive (`True`). If two neighbours of a cell are equal, it becomes dead at the next step, otherwise alive.

```
step :: (Int -> Bool) -> (Int -> Bool)
step current = \n -> current (n - 1) /= current (n + 1)

step :: Chimera Bool -> Chimera Bool
step current = tabulate $ \n ->
  current `index` (n - 1) /= current `index` (n + 1)
```

Use continuous mappings:

```
>>> map intToWord [-5..5]
[9,7,5,3,1,0,2,4,6,8,10]
>>> map wordToInt [0..10]
[0,-1,1,-2,2,-3,3,-4,4,-5,5]
```

# Cellular automata 2D

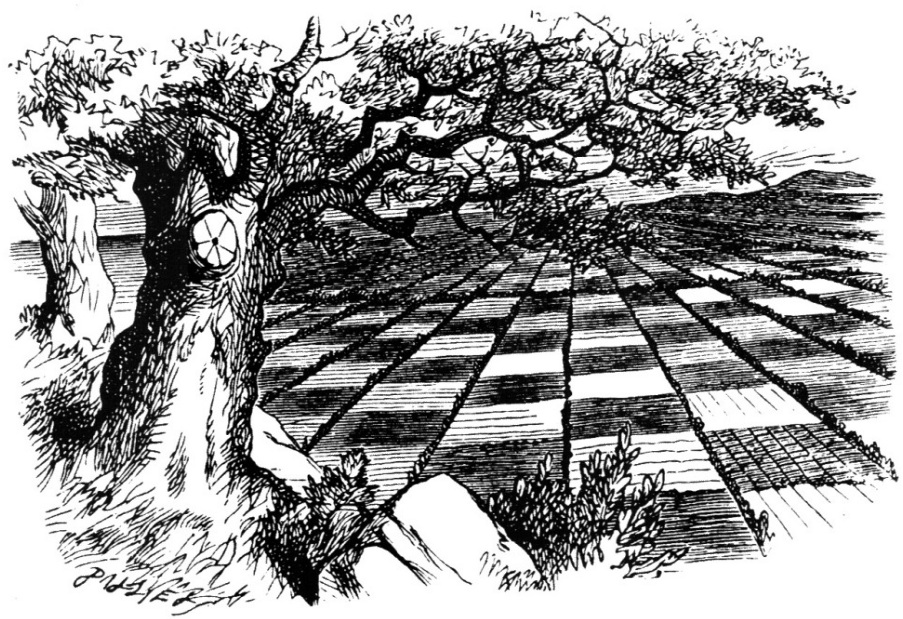
**Conway's Game of Life.** An infinite in all directions plane of cells. If a live cell has 2 or 3 neighbours, it remains alive at the next step, otherwise dies. An empty cell with exactly 3 neighbours becomes alive at the next step.

```
step :: (Int -> Int -> Bool) -> (Int -> Int -> Bool)
```

```
step :: (Word -> Word -> Bool) -> (Word -> Word -> Bool)
```

Interleave binary coordinate values:

```
> toZCurve 0b0000 0b1111
0b01010101
> fromZCurve 0b010101010
(0b0000, 0b1111)
```



# Unboxed vectors and predicates

```
newtype Chimera v a = Chimera (Vector (v a))

type UChimera = Chimera Data.Vector.Unboxed.Vector

isPrime :: Word -> Bool
isPrime n = n>1 && and [n `rem` d /= 0 | d <- [2..n-1]]
```

Use `bitvec` package to pack booleans densely:

```
import Data.Bit

isPrime' :: UChimera Bit
isPrime' = tabulate
  (Bit . isPrime)
```

@ andrew.lelechenko@gmail.com

github.com/Bodigrim/chimera

github.com/Bodigrim/my-talks

# Thank you!

Illustrations by Sir John Tenniel