Introduction to Functional Programming

Martín Escardó

Welcome to the module!

Learning objectives

- A brief and incomplete visit to the zoo of functional programming languages.
- A brief overview of our chosen functional programming language, namely Haskell.

→ Each concept mentioned here will be developed in a future lecture.

The Zoo of functional programming languages

untyped →

- LISP (lots of stupid an infernal parentheses – old joke :))
  1958
  - scheme, emacs lisp, racket

typed →

- ML
  1973
  - SML, 1983
  - OCAML, 1996
  - F#, 2005 (Microsoft)
- Hope, 1970
- Miranda, 1985
- Haskell, 1990 (After logician Haskell Curry)
- Coq, 1989
- Agda, 1999
- Lean, 2013

← dependently typed, can write proofs.

Other languages with functional features (list not exhaustive)

- Java
  - λ (lambda)
  - functional interfaces
  - streams
- Kotlin
  - The preferred language for Android development since 2019.
  - More heavily based on functional programming ideas.

Introduction to Haskell - the game of life

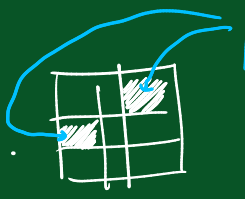
Conway's Game of Life

The Wikipedia page is a good and reliable reference.

See also our own lecture notes.

Life

- Takes place in a grid of cells (infinite, 2-dimensional)
- Each cell is live or dead

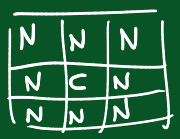


live

all the other cells are dead.

- Each cell interacts with its eight neighbours

The neighbours of C ⟶



Life continued

The game proceeds in discrete steps of time:

1. Any live cell with 0 or 1 live neighbours dies by underpopulation.
2. Any live cell with 2 or 3 live neighbours survives.
3. Any live cell with more than 3 live neighbours dies by overpopulation.
4. Any dead cell with 3 live neighbours comes to life by reproduction.

Let's do this in Haskell now.

(There will be lectures explaining each feature we use in detail)

We make the following type declarations:

```
type Cell = (Int,Int)
type Grid = [Cell]
```

This says that a cell is represented by a pair of integers. (Its xy coordinates)

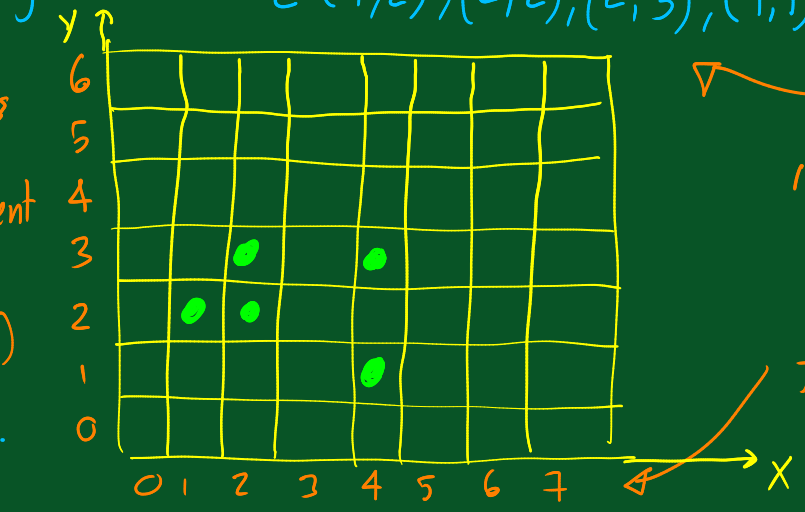
A grid is represented by a list of cells.

Example: The pentagenarian starting pattern.

```
pentagenarian :: Grid
pentagenarian = [(1,2),(2,2),(2,3),(4,1),(4,3)]
```

type declaration.

definition.



Second component of the pair (y coordinate)

the order of the pairs in the list doesn't matter.

first component of the pair (x coordinate)

Checking whether a cell is live or dead:

```
isLive, isDead :: Cell -> Grid -> Bool
isLive c g = c `elem` g
isDead c g = not (isLive c g)
```

two type declarations separated by comma

definitions of the two declared functions

The two functions isLive and isDead take two inputs (namely a Cell c and a Grid g) and return a Boolean True or False.

The list of neighbours of a cell

```
neighbours :: Cell -> [Cell]
neighbours (x,y) = [(x+i, y+j) | i <- [-1..1],
                                 j <- [-1..1],
                                 not (i==0 && j==0)]
```

definition by "pattern matching"

"list comprehension"

This is the list of all $(x+i, y+j)$ where i and j range from -1 to 1, excluding the case in which both are zero.

The live neighbours of a cell in a grid

```
liveNeighbours :: Grid -> Cell -> [Cell]
liveNeighbours g c = [ c' | c' <- neighbours c, isLive c' g ]
```

a list comprehension again

The evolution function step of the game of life

```
step :: Grid -> Grid
step g = [ (x,y) | x <- [minX-1, maxX+1],
                   y <- [minY-1, maxY+1],
                   (isDead (x,y) g && length (liveNeighbours g (x,y)) == 3)
                || (isLive (x,y) g && length (liveNeighbours g (x,y)) `elem` [2,3])
         ]
  where
    minX = minimum [x | (x,y) <- g]
    maxX = maximum [x | (x,y) <- g]
    minY = minimum [y | (x,y) <- g]
    maxY = maximum [y | (x,y) <- g]
```

That's it!

- We now need to display each step to get an animation of Life.
- We will adopt a simple minded approach.
- We will use a terminal.
- Pause for demonstrating the animation in a terminal.

Code for animating the game in a terminal

```
terminalWidth = 70
terminalHeight = 22
```

← We don't need to declare the types. They are inferred automatically.

The terminal accepts control commands as special characters.

Clear the terminal:

```
cls :: IO ()
```

← type of programs that perform input/output actions and return "nothing".

```
cls = putStr "\ESC[2J"
```

← Terminal control sequence (see table) (ANSI escape code)

Move cursor to a given position, or cell

```
goto :: Cell -> IO ()
goto (x,y) = putStr ("\ESC[" ++ show (terminalHeight - y) ++ ";" ++ show (x+1) ++ "H")
```

corresponds to Java's toString()

Printing a cell in the terminal

```
printCell :: Cell -> IO ()
printCell (x,y) | x >= 0 && x < terminalWidth
                && y >= 0 && y < terminalHeight = do
                    goto (x,y)
                    putChar 'O'
                | otherwise = return ()
```

prints the letter O and returns the value ()

Performs no I/O action and returns the value ().

Render a grid in the terminal

```
terminalRender :: Grid -> IO ()
terminalRender g = do
  cls
  sequence [printCell c | c <- g]
  goto (0, terminalHeight)
```

List of I/O actions, one for each cell c in g.

Performs a list of I/O actions sequentially, one after the other.

10^{th} of a second delay

```
delayTenthSec n = threadDelay (n * 10^5)
```

threadDelay: Library function from the import Control.Concurrent.

Whoooops! we forgot to declare the type of this function.

→ It is automatically inferred to be `Int -> IO ()`.

→ But I strongly recommend to add types, for both debugging and documentation. It makes programs more readable.

Finally, we animate the game in the terminal

```
life :: Grid -> IO ()
life seed = f 0 seed
  where
    f n g = do terminalRender g
               putStrLn (show n)
               delay TenthSec 1
               f (n+1) (step g)
```

We have also developed a version in Kotlin for you

> Check the resources for the module for both the Haskell and the Kotlin files. Read the lecture notes too.

> Can you translate them to Java, while maintaining their functional style?

Summary

We used the Game of Life to illustrate the features of Haskell that we'll learn in future lectures.