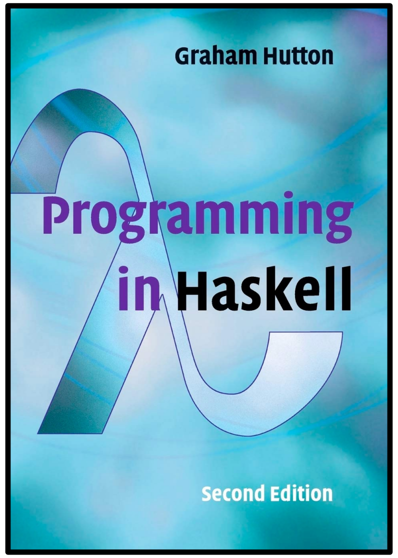
Functional Core and Imperative Shell
Game of Life Example

Polyglot **FP** for **F**un and **P**rofit – **Haskell** and **Scala**

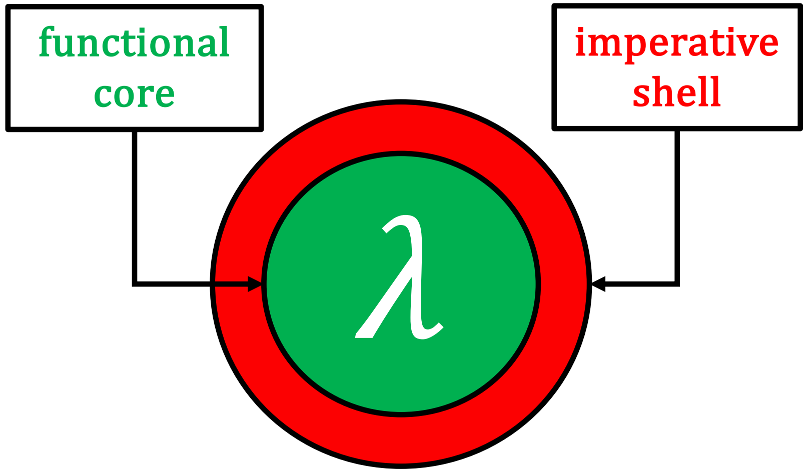
See a **program structure flowchart** used to highlight how an FP program breaks down into a **functional core** and **imperative shell**

View a **program structure flowchart** for the **Game of Life**

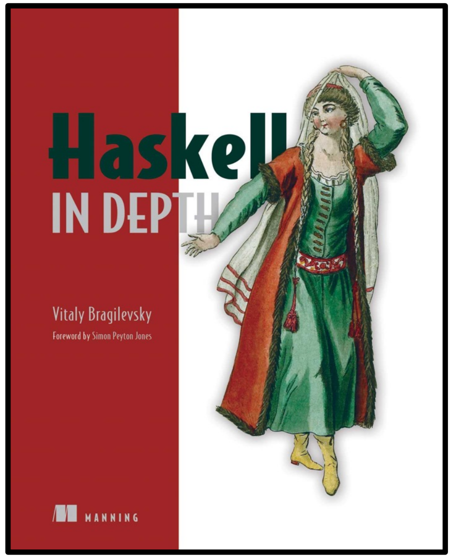
See the code for **Game of Life**'s **functional core** and **imperative shell**, both in **Haskell** and in **Scala**



Graham Hutton
@haskellhutt



Vitaly Bragilevsky
@VBragilevsky



slides by  @philip_schwarz slideshare https://www.slideshare.net/pjschwarz

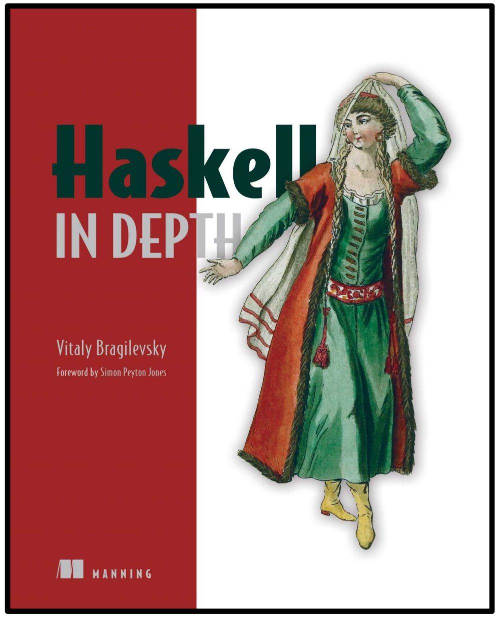
@philip_schwarz

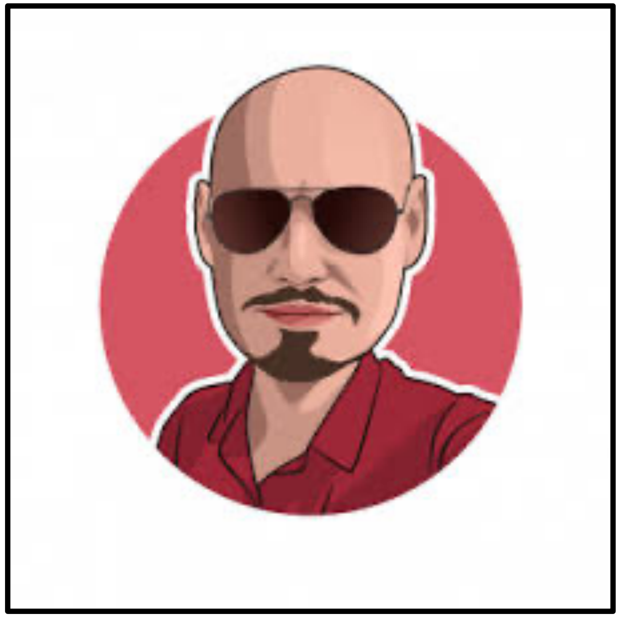
In **Haskell in Depth**, **Vitaly Bragilevsky** visualises certain aspects of his programs using **program structure flowcharts**.

One of the things shown by his diagrams is how the programs break down into a **pure part** and an **I/O part**, i.e. into a **functional core** and an **imperative shell**.

In this short slide deck we do the following:

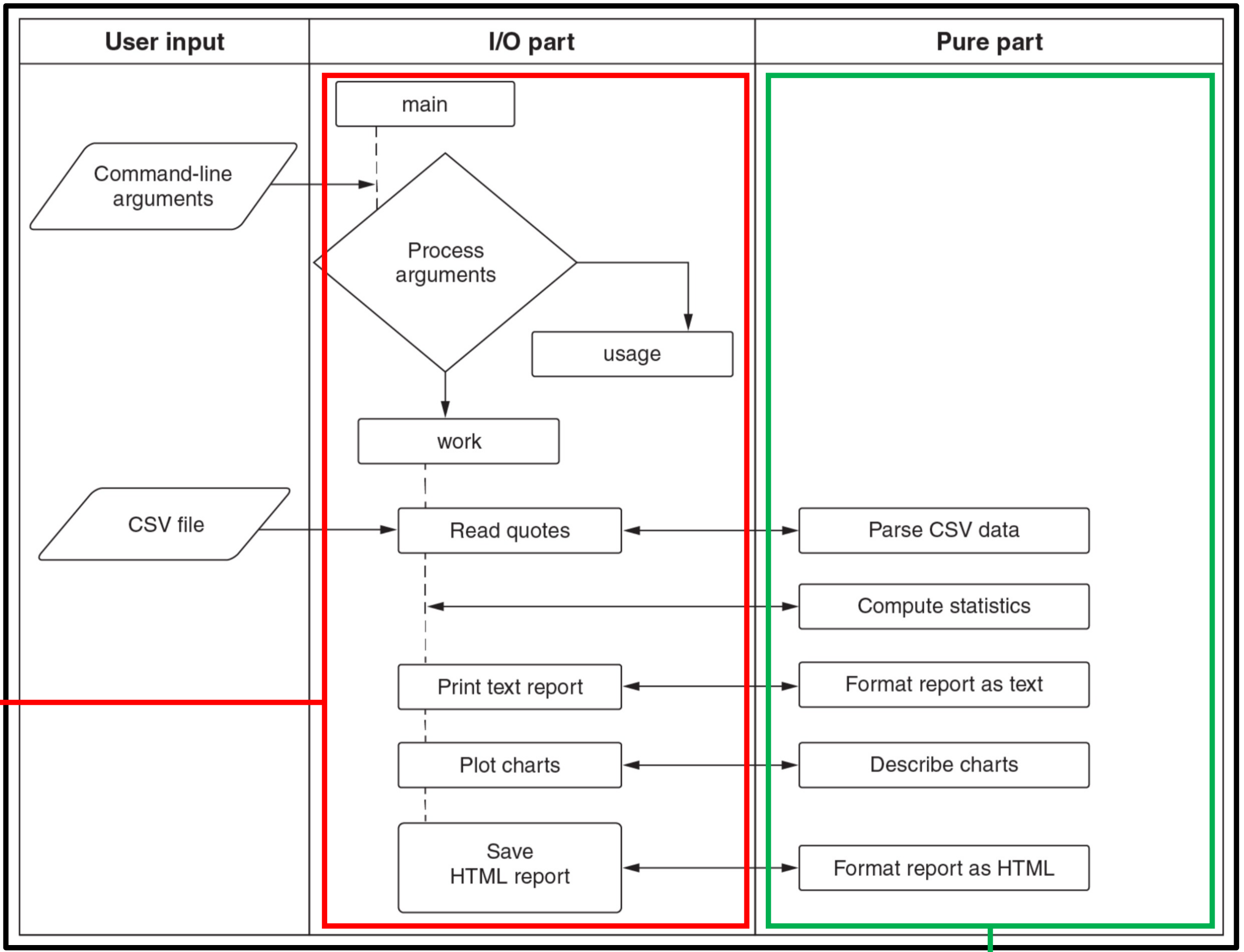
- create a **program structure flowchart** for the **Game of Life**
- show how **Game of Life** code consists of a **functional core** and an **imperative shell**





Vitaly Bragilevsky

@VBragilevsky



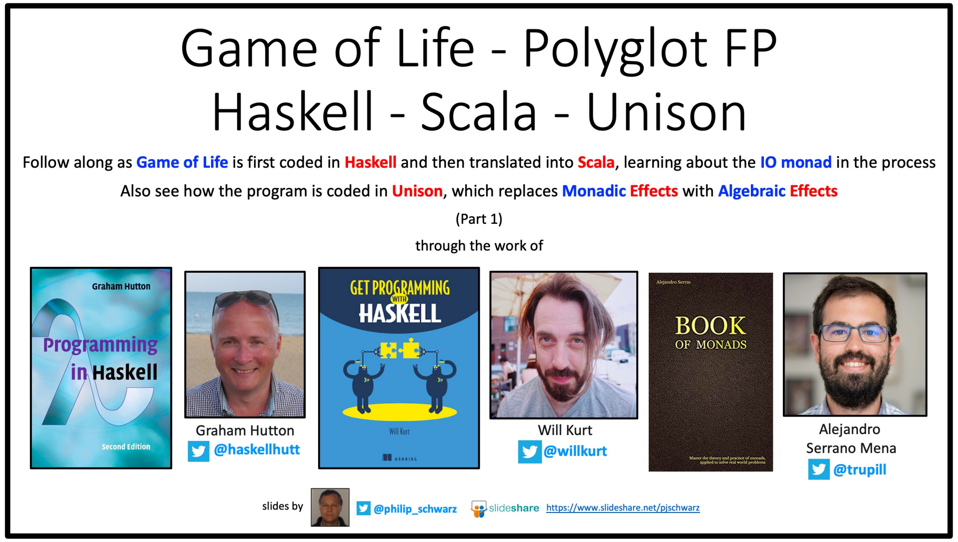
READING A PROGRAM STRUCTURE FLOWCHART

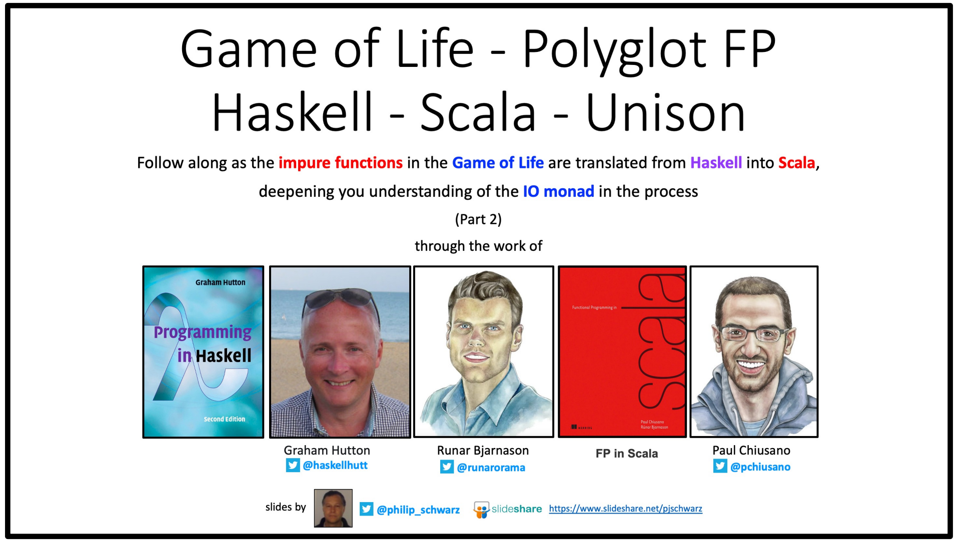
I've tried to present all the components of the program in a **program structure flowchart**: user input, **actions** in the **I/O** part of the program, and their relations with the **pure functions**. I use the following notation

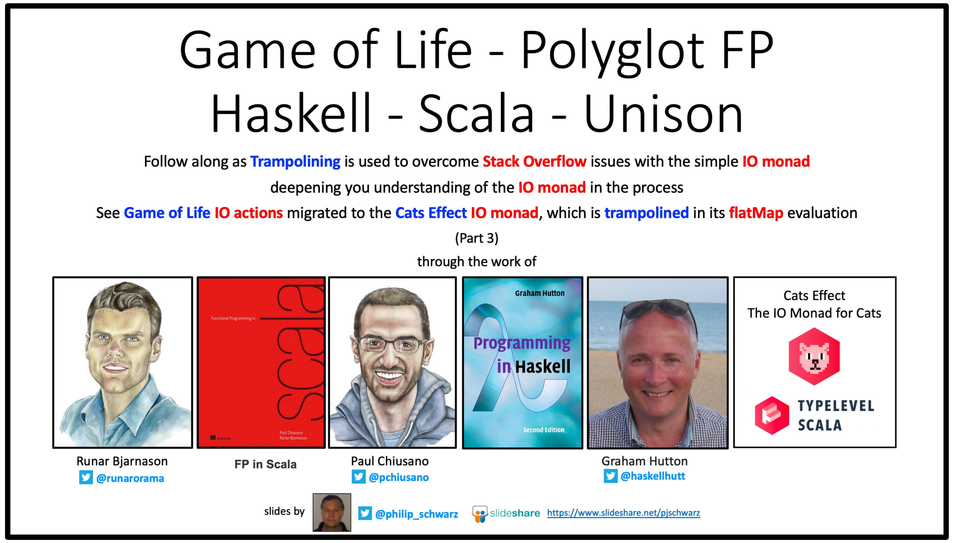
- User input is represented by parallelograms.
- All functions are represented by rectangles.
- **Some of the functions are executing I/O actions. These are shown in the central part of the flowchart.**
- **Other functions are pure. They are given on the right-hand side**.
- Diamonds traditionally represent choices made within a program.
- Function calls are represented by rectangles below and to the right of a caller.
- Several calls within a function are combined with a dashed line.
- Arrows in this flowchart represent moving data between the user and the program and between functions within the program.

If you would like an introduction to the notion of '**functional core**, **imperative shell**', see slides 15-20 of the second slide deck below.

If you want an explanation of the **Game of Life** code that we'll be looking at next, see the first slide deck for **Haskell**, and the remaining two for **Scala**.







If we run the upcoming **Game of Life** program with a 20 by 20 board configured with the **first generation** of a **Pulsar**, the program cycles forever through the following three patterns

```haskell
type Pos = (Int,Int)

type Board = [Pos]
```

```haskell
width :: Int
width = 20

height :: Int
height = 20
```

```
  OOO   OOO

O    O O    O
O    O O    O
O    O O    O
  OOO   OOO

  OOO   OOO
O    O O    O
O    O O    O
O    O O    O

  OOO   OOO
```

```
    O     O
    O     O
    OO   OO

OOO  OO OO  OOO
  O O O O O O
    OO   OO

    OO   OO
  O O O O O O
OOO  OO OO  OOO

    OO   OO
    O     O
    O     O
```

```
   OO     OO
    OO   OO
 O  O O O O  O
 OOO OO OO OOO
  O O O O O O
   OOO   OOO

   OOO   OOO
  O O O O O O
 OOO OO OO OOO
 O  O O O O  O
    OO   OO
   OO     OO
```

```haskell
pulsar :: Board
pulsar =
  [(4, 2),(5, 2),(6, 2),(10, 2),(11, 2),(12, 2),
          (2, 4),(7, 4),( 9, 4),(14, 4),
          (2, 5),(7, 5),( 9, 5),(14, 5),
          (2, 6),(7, 6),( 9, 6),(14, 6),
   (4, 7),(5, 7),(6, 7),(10, 7),(11, 7),(12, 7),
   (4, 9),(5, 9),(6, 9),(10, 9),(11, 9),(12, 9),
          (2,10),(7,10),( 9,10),(14,10),
          (2,11),(7,11),( 9,11),(14,11),
          (2,12),(7,12),( 9,12),(14,12),
   (4,14),(5,14),(6,14),(10,14),(11,14),(12,14)]
```

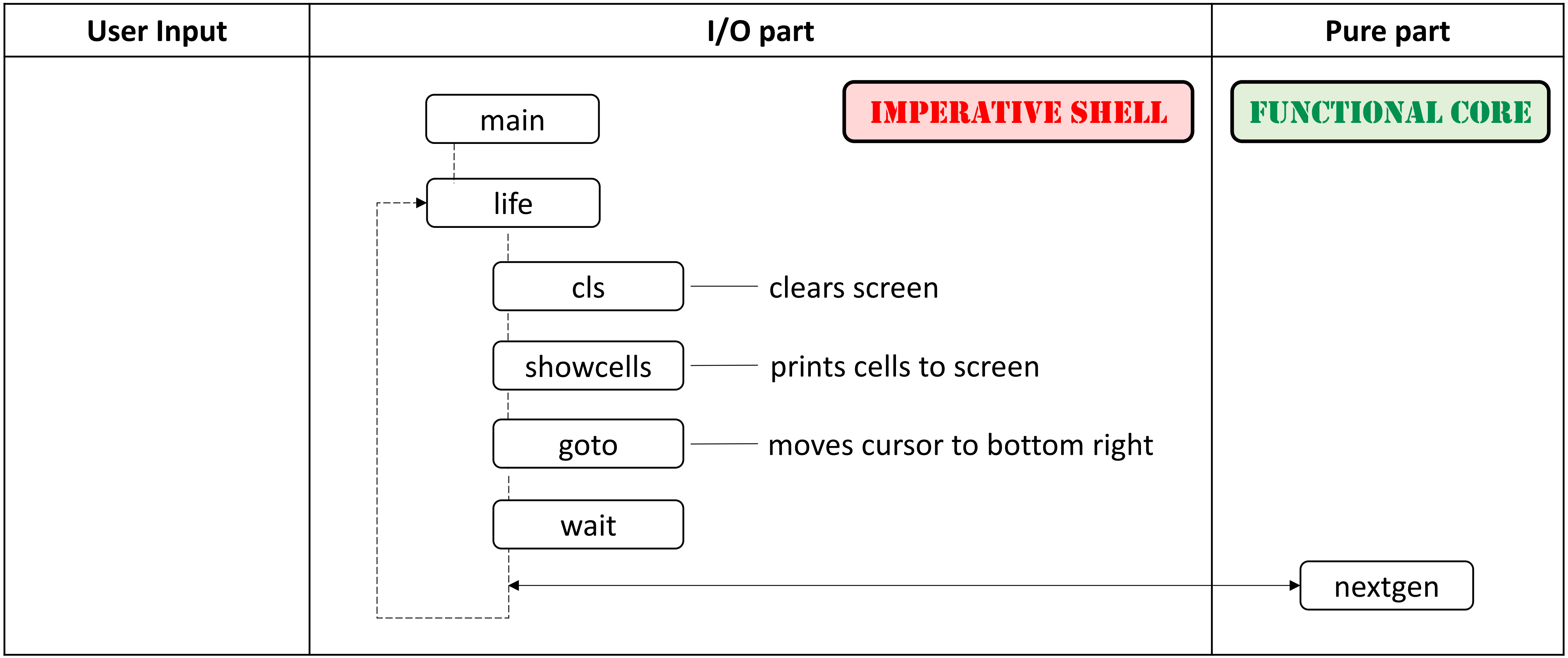
The next slide shows a simple **program structure flowchart** for the **Game of Life** program.

The rest of the slides show the following:

1. **Haskell** code for the program's **imperative shell**
2. **Haskell** code for its **functional core**
3. **structure flowchart** for the program (**Scala** version)
4. **Scala** code for the **imperative shell**
5. **Scala** code for the **functional core**

The **Haskell** **Game of Life** code is the one found in **Graham Hutton**'s book, **Programming in Haskell**, with a handful of very minor changes, e.g.

- added an extra invocation of a function in order to move the cursor out of the way after drawing a **generation**
- added data for the **first pulsar generation**




```haskell
main :: IO ()
main = life(pulsar)
```

```haskell
life :: Board -> IO ()
life b = do
  cls
  showcells b
  goto (width+1,height+1)
  wait 500000
  life (nextgen b)
```

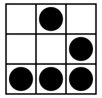
```haskell
cls :: IO ()
```

```haskell
showcells :: Board -> IO ()
```

```haskell
goto :: Pos -> IO ()
```

```haskell
wait :: Int -> IO ()
```

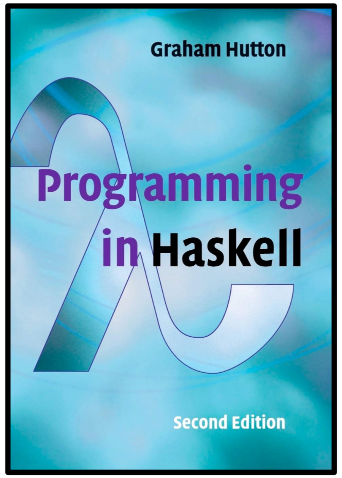
```haskell
nextgen :: Board -> Board
```



IMPERATIVE SHELL Haskell

Graham Hutton

@haskellhutt




```haskell
main :: IO ()
main = life(pulsar)

life :: Board -> IO ()
life b = do cls
            showcells b
            goto (width + 1, height + 1)
            wait 500000
            life (nextgen b)
```

```haskell
cls :: IO ()
cls = putStr "\ESC[2J"

showcells :: Board -> IO ()
showcells b = sequence_ [writeat p "O" | p <- b]

wait :: Int -> IO ()
wait n = sequence_ [return () | _ <- [1..n]]
```

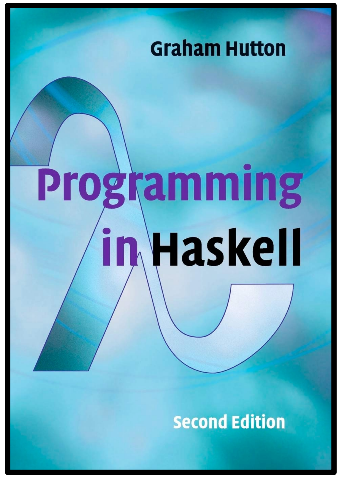
```haskell
writeat :: Pos -> String -> IO ()
writeat p xs = do goto p
                  putStr xs

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

```haskell
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

FUNCTIONAL CORE Haskell

Graham Hutton
@haskellhutt




```
nextgen :: Board -> Board
nextgen b = survivors b ++ births b

survivors :: Board -> [Pos]
survivors b =
  [p | p <- b,
       elem (liveneighbs b p) [2,3]]

births :: Board -> [Pos]
births b = [p | p <- rmdups (concat (map neighbs b)),
                isEmpty b p,
                liveneighbs b p == 3]
```

```
rmdups :: Eq a => [a] -> [a]
rmdups [] = []
rmdups (x:xs) = x : rmdups (filter (/= x) xs)

isEmpty :: Board -> Pos -> Bool
isEmpty b p = not (isAlive b p)

liveneighbs :: Board -> Pos -> Int
liveneighbs b = length.filter(isAlive b).neighbs

isAlive :: Board -> Pos -> Bool
isAlive b p = elem p b
```

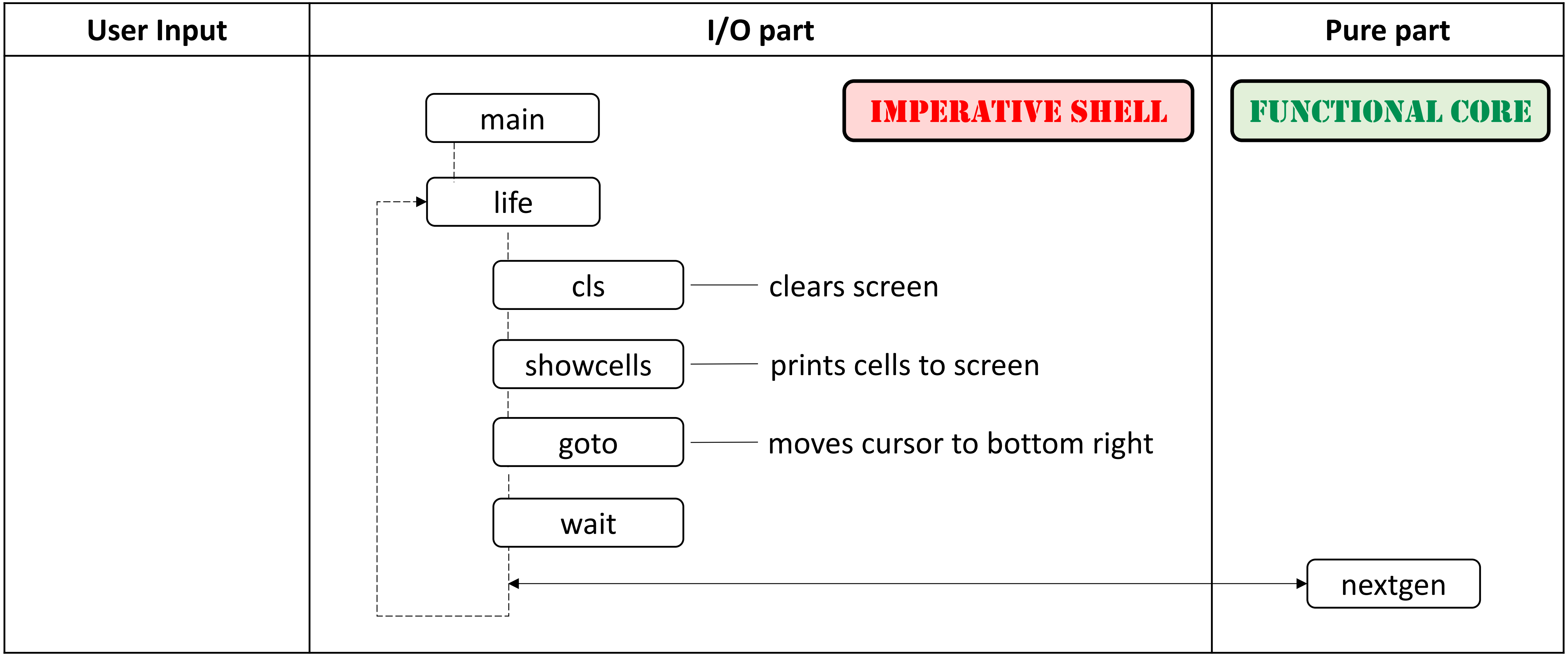
```
neighbs :: Pos -> [Pos]
neighbs (x,y) = map wrap [(x-1, y-1), (x,   y-1),
                          (x+1, y-1), (x-1, y  ),
                          (x+1,   y), (x-1, y+1),
                          (x,   y+1), (x+1, y+1)]

wrap :: Pos -> Pos
wrap (x,y) = (((x-1) `mod` width) + 1,
              ((y-1) `mod` height) + 1)

width :: Int        height :: Int
width = 20          height = 20
```

```
type Pos = (Int,Int)
type Board = [Pos]
```

```
pulsar :: Board
pulsar =
  [(4, 2),(5, 2),(6, 2),(10, 2),(11, 2),(12, 2),
          (2, 4),(7, 4),( 9, 4),(14, 4),
          (2, 5),(7, 5),( 9, 5),(14, 5),
          (2, 6),(7, 6),( 9, 6),(14, 6),
   (4, 7),(5, 7),(6, 7),(10, 7),(11, 7),(12, 7),
   (4, 9),(5, 9),(6, 9),(10, 9),(11, 9),(12, 9),
          (2,10),(7,10),( 9,10),(14,10),
          (2,11),(7,11),( 9,11),(14,11),
          (2,12),(7,12),( 9,12),(14,12),
   (4,14),(5,14),(6,14),(10,14),(11,14),(12,14)]
```




```
val main: IO[Unit] =
  life(pulsar)
```

```
def life(b: Board): IO[Unit] =
  cls *>
  showCells(b) *>
  goto(width+1,height+1) *>
  wait(1_000_000) >>
  life(nextgen(b))
```

```
def cls: IO[Unit]
```

```
def showCells(b: Board): IO[Unit]
```

```
def goto(p: Pos): IO[Unit]
```

```
def wait(n:Int): IO[Unit]
```

```
def nextgen(b:Board):Board
```

Scala

IMPERATIVE SHELL Scala

```scala
import cats.implicits._, cats.effect.IO
```

```scala
val main: IO[Unit] = life(pulsar)

def life(b: Board): IO[Unit] =
  cls *>
  showCells(b) *>
  goto(width+1,height+1) *>
  wait(1_000_000) >>
  life(nextgen(b))
```

```scala
def cls: IO[Unit] = putStr("\u001B[2J")

def showCells(b: Board): IO[Unit] =
  ( for { p <- b } yield writeAt(p, "O") ).sequence_

def wait(n:Int): IO[Unit] = List.fill(n)(IO.unit).sequence_
```

```scala
def writeAt(p: Pos, s: String): IO[Unit] =
  goto(p) *> putStr(s)

def goto(p: Pos): IO[Unit] = p match {
  case (x,y) => putStr(s"\u001B[${y};${x}H")
}
```

```scala
def putStr(s: String): IO[Unit] =
  IO { scala.Predef.print(s) }
```

Cats Effect
The IO Monad for Cats

TYPELEVEL SCALA

```scala
main.unsafeRunSync
```

FUNCTIONAL CORE

Scala

```scala
def nextgen(b: Board): Board = survivors(b) ++ births(b)

def survivors(b: Board): List[Pos] =
  for {
    p <- b
    if List(2,3) contains liveneighbs(b)(p)
  } yield p

def births(b: Board): List[Pos] =
  for {
    p <- rmdups(b flatMap neighbs)
    if isEmpty(b)(p)
    if liveneighbs(b)(p) == 3
  } yield p
```

```scala
def rmdups[A](l: List[A]): List[A] = l match {
  case Nil => Nil
  case x::xs => x::rmdups(xs filter(_ != x)) }

def isEmpty(b: Board)(p: Pos): Boolean =
  !(isAlive(b)(p))

def liveneighbs(b: Board)(p: Pos): Int =
  neighbs(p).filter(isAlive(b)).length

def isAlive(b: Board)(p: Pos): Boolean =
  b contains p
```

```scala
def neighbs(p: Pos): List[Pos] = p match {
  case (x,y) => List(
    (x - 1, y - 1), (x, y - 1), (x + 1, y - 1),
    (x - 1, y    ), /* cell */  (x + 1, y    ),
    (x - 1, y + 1), (x, y + 1), (x + 1, y + 1)
  ) map wrap }

def wrap(p: Pos): Pos = p match {
  case (x, y) => (((x - 1) % width) + 1,
                  ((y - 1) % height) + 1) }

val width = 20      val height = 20
```

```scala
type Pos = (Int, Int)
type Board = List[Pos]
```

```scala
val pulsar: Board = List(
    (4, 2),(5, 2),(6, 2),(10, 2),(11, 2),(12, 2),
          (2, 4),(7, 4),( 9, 4),(14, 4),
          (2, 5),(7, 5),( 9, 5),(14, 5),
          (2, 6),(7, 6),( 9, 6),(14, 6),
    (4, 7),(5, 7),(6, 7),(10, 7),(11, 7),(12, 7),
    (4, 9),(5, 9),(6, 9),(10, 9),(11, 9),(12, 9),
          (2,10),(7,10),( 9,10),(14,10),
          (2,11),(7,11),( 9,11),(14,11),
          (2,12),(7,12),( 9,12),(14,12),
    (4,14),(5,14),(6,14),(10,14),(11,14),(12,14)])
```

@philip_schwarz

If you want to run the programs, you can find them here:

- https://github.com/philipschwarz/functional-core-imperative-shell-scala
- https://github.com/philipschwarz/functional-core-imperative-shell-haskell

That's all.

I hope you found it useful.

See you soon.