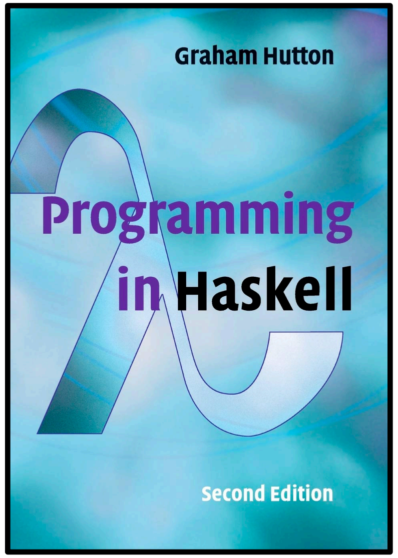
Game of Life - Polyglot FP
Haskell - Scala - Unison

Follow along as **Game of Life** is first coded in **Haskell** and then translated into **Scala**, learning about the **IO monad** in the process

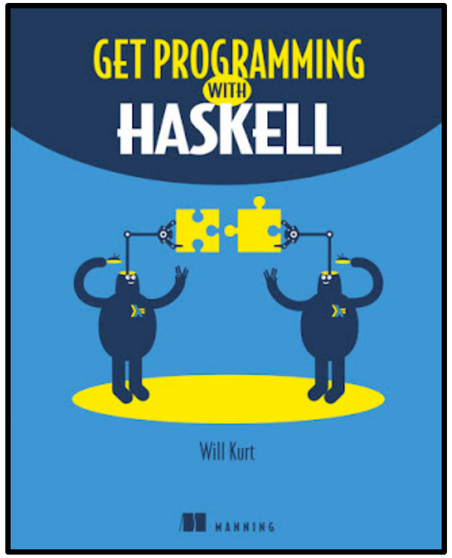
Also see how the program is coded in **Unison**, which replaces **Monadic Effects** with **Algebraic Effects**

(Part 1)

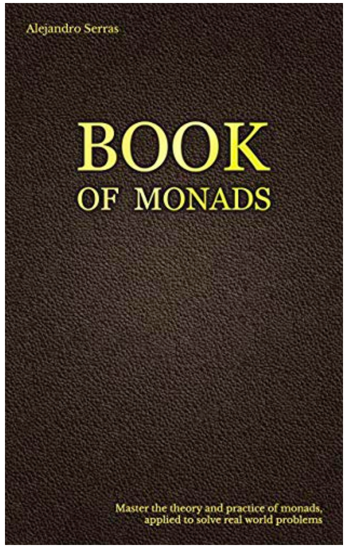
through the work of



Graham Hutton
@haskellhutt



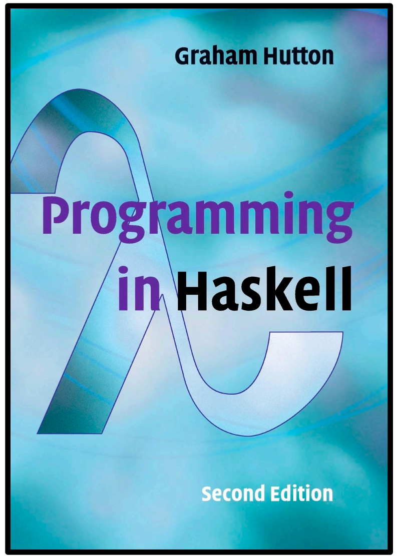
Will Kurt
@willkurt



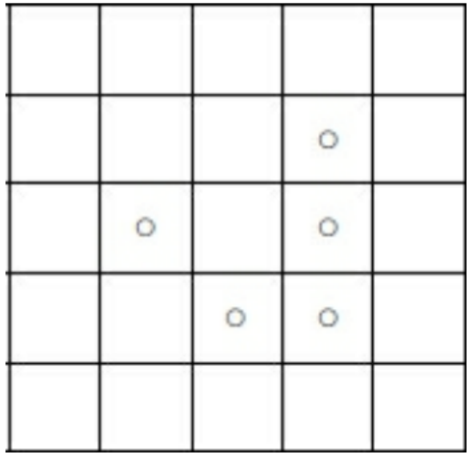
Alejandro Serrano Mena
@trupill

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

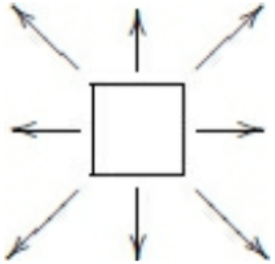
Graham Hutton
@haskellhutt



Our third and final interactive programming example concerns the **game of life**. The game models a simple evolutionary system based on **cells**, and is played on a two-dimensional **board**. Each **square** on the **board** is either **empty**, or contains a **single living cell**, as illustrated in the following example:

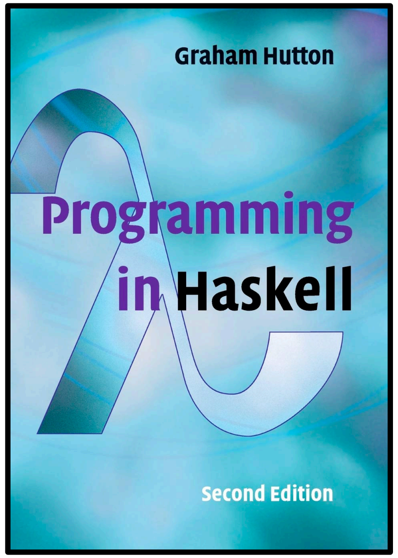


Each **internal square** on the **board** has **eight immediate neighbours**:



For uniformity, each **external square** on the **board** is also viewed as having **eight neighbours**, by assuming that the **board** wraps around from top-to-bottom and from left-to-right. That is, we can think of the **board** as really being a torus, the surface of a three-dimensional doughnut shaped object.

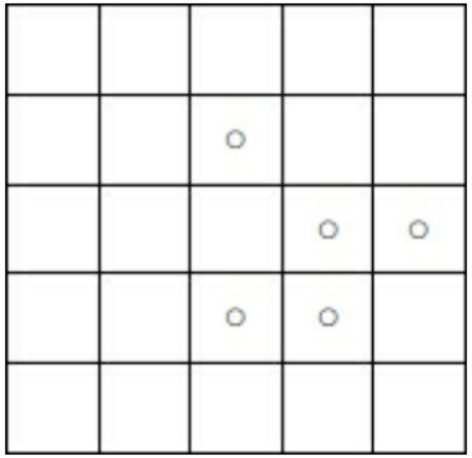
Graham Hutton
@haskellhutt



Given an initial configuration of the **board**, the **next generation** of the **board** is given by simultaneously applying the following **rules** to all **squares**:

- **a living cell survives if it has precisely two or three neighbouring squares that contain living cells, and**
- **an empty square gives birth to a living cell if it has precisely three neighbours that contain living cells, and remains empty otherwise**.

For example, applying these **rules** to the above **board** gives:



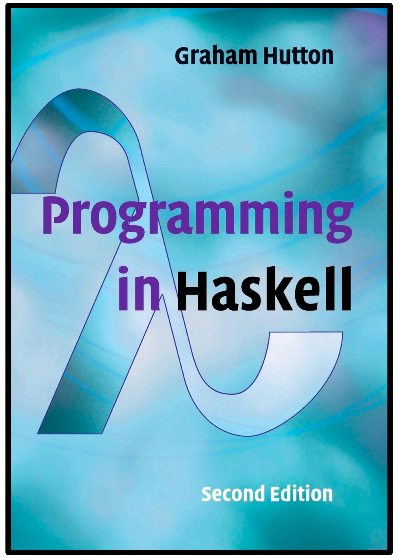
By repeating this **procedure** with the new **board**, an infinite sequence of **generations** can be produced. **By careful design of the initial configuration, many interesting patterns of behaviour can be observed in such sequences**. For example, the above arrangement of **cells** is called a **glider**, and **over successive generations will move diagonally down the board**. Despite its simplicity, the **game of life** is in fact **computationally complete**, in the sense that **any computational process can be simulated within it** by means of a suitable encoding. In the remainder of this section we show **how the game of life can be implemented in Haskell**.

@philip_schwarz

The next section of the book is called **Screen utilities** and involves functions that have the **side-effect** of **writing to the screen**.

We prefer to look at **pure** functions first, and **side-effecting** functions next, so we are going to skip that section of the book for now and come back to it later, except for the first few lines of the next slide, which are from that section and introduce the use of a **Pos** type to represent **coordinate positions**.

Graham Hutton
@haskellhutt



By convention, the **position of each character on the screen is given by a pair (x,y) of positive integers, with (1,1) being the top-left corner**.

We represent such **coordinate positions** using the following type:

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

For increased flexibility, we allow the board size for **life** to be modified, by means of two integer values that specify the size of the **board** in **squares**:

```
width :: Int
width = 10

height :: Int
height = 10
```

We represent a board as a list of the (x,y) positions at which there is a living cell, using the same coordinate convention as the screen:

```
type Board = [Pos]
```

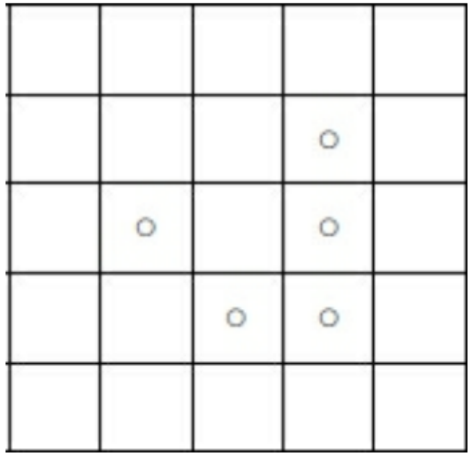
For example, the initial example **board** above would be represented by:

```
glider :: Board
glider = [(4,2),(2,3),(4,3),(3,4),(4,4)]
```

Just in case it helps, here is how the **positions** of the first **generation** of the **glider** map to a 5 x 5 **board**. Similarly for the second **generation**, generated fom the first by applying the rules.

```haskell
type Board = [Pos]
```

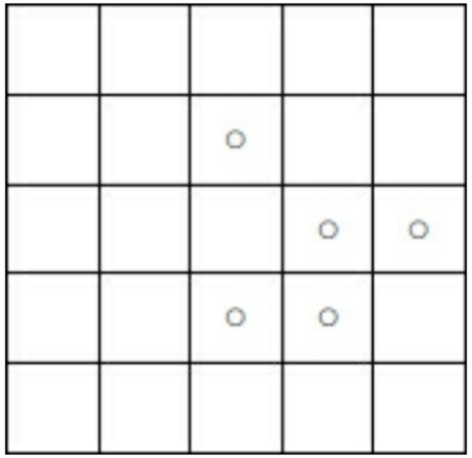
```haskell
glider :: Board
glider = [(4,2),(2,3),(4,3),(3,4),(4,4)]
```



	COL 1	COL 2	COL 3	COL 4	COL 5
ROW 1					
ROW 2				O	
ROW 3		O		O	
ROW 4			O	O	
ROW 5					

```
 C,R   C,R   C,R   C,R   C,R
[(4,2),(2,3),(4,3),(3,4),(4,4)]
 X,Y   X,Y   X,Y   X,Y   X,Y
```

Rules
1. a **living cell** **survives** if it has **precisely** **two or three** **neighbouring squares** that contain **living cells**, and
2. an **empty square** **gives birth** to a **living cell** if it has **precisely** **three neighbours** that contain **living cells**, and remains **empty** otherwise



	COL 1	COL 2	COL 3	COL 4	COL 5
ROW 1					
ROW 2			O		
ROW 3				O	O
ROW 4			O	O	
ROW 5					

```
 C,R   C,R   C,R   C,R   C,R
[(3,2),(4,3),(5,3),(3,4),(4,4)]
 X,Y   X,Y   X,Y   X,Y   X,Y
```

As we progress through the **Haskell** program, we are going to translate it into **Scala**.

We begin on the next slide by translating the code we have seen so far.

Here is the Haskell code we have seen so far, and next to it, the Scala equivalent.

@philip_schwarz

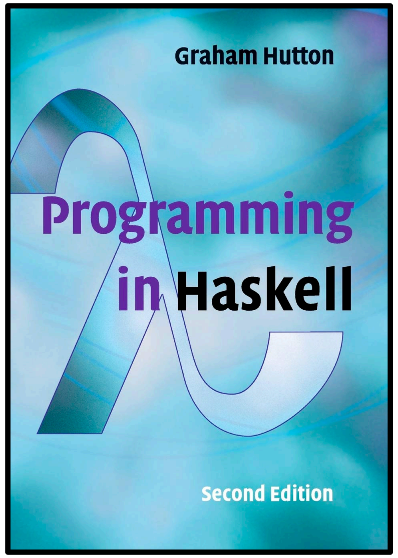
```haskell
type Pos = (Int,Int)

width :: Int
width = 10

height :: Int
height = 10

type Board = [Pos]

glider :: Board
glider = [(4,2),(2,3),(4,3),(3,4),(4,4)]
```

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


val width = 20


val height = 20

type Board = List[Pos]


val glider: Board = List((4,2),(2,3),(4,3),(3,4),(4,4))
```

On the next slide, **Graham Hutton** looks at a function for displaying **living cells** on the **screen**. Because that function is **side-effecting**, we'll skip it for now and come back to it later.

Graham Hutton

@haskellhutt



Using this representation of the board, it is easy to display living cells on the screen, and to decide if a given position is alive or empty:

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

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

Next, we define a function that returns the neighbours of a position:

```
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)]
```

The auxiliary function wrap takes account of the wrapping around at the edges of the board, by subtracting one from each component of the given position, taking the remainder when divided by the width and height of the board, and then adding one to each component again:

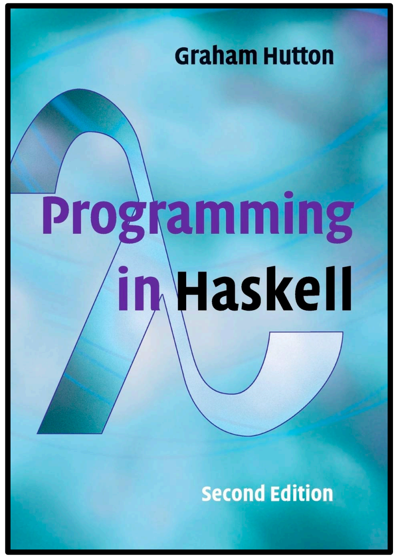
```
wrap :: Pos -> Pos
wrap (x,y) = (((x-1) `mod` width) + 1,
              ((y-1) `mod` height) + 1)
```

Let’s write the Scala equivalent of the Haskell functions we have just seen.

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

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

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)
```

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

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

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)
}
```

Graham Hutton
@haskellhutt



Using function composition, we can now define a function that calculates the number of live neighbours for a given position by producing the list of its neighbours, retaining those that are alive, and counting their number:

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

Using this function, it is then straightforward to produce the list of living positions in a board that have precisely two or three living neighbours, and hence survive to the next generation of the game:

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

In turn, the list of empty positions in a board that have precisely three living neighbours, and hence give birth to a new cell, can be produced as follows:

```
births :: Board -> [Pos]
births b = [(x,y) | x <- [1..width],
                    y <- [1..height],
                    isEmpty b (x,y),
                    liveneighbs b (x,y) == 3]
```

@philip_schwarz

Here is the Scala equivalent of the Haskell functions we have just seen.

See the next few slides for the reason why I translated liveneighbs the way I did.

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

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


births :: Board -> [Pos]
births b =
  [(x,y) | x <- [1..width],
           y <- [1..height],
           isEmpty b (x,y),
           liveneighbs b (x,y) == 3]
```

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

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 {
    x <- List.range(1,width + 1)
    y <- List.range(1,height + 1)
    if isEmpty(b)((x,y))
    if liveneighbs(b)((x,y)) == 3
  } yield (x,y)
```

The reason why in **Haskell** it is possible to implement **liveneighbs** as the **composition** of **neighbs**, **filter(isAlive b)** and **length**, is that their signatures align: the output type of **neighbs** is the input type of **filter(isAlive b)** and the otput type of the latter is the input type of **length**.

```
liveneighbs :: [Pos] -> Pos -> Int
liveneighbs = length . filter(isAlive b) . neighbs

                          length     [a] -> Int
                          filter     (a -> Bool) -> [a] -> [a]
                         isAlive     [Pos] -> Pos -> Bool
                               b     [Pos]
                    (isAlive b)      Pos -> Bool
              filter(isAlive b)      [Pos] -> [Pos]
                         neighbs     Pos -> [Pos]
    filter(isAlive b) . neighbs      Pos -> [Pos]
length . filter(isAlive b) . neighbs Pos -> Int

liveneighbs =          length        .      filter(isAlive b)       .      neighbs
[Pos] -> Int ::        [Pos] -> Int  .      [Pos] -> [Pos]          .      Pos -> [Pos]
```

But in **Scala**, the signatures of **neighbs**, **filter(isAlive b)** and **length** do not align because **length** and **filter** are not functions that take a `List[Pos]` parameter, but rather they are functions provided by `List`:

Scala

```
List[A] – length: Int
List[A] – filter: (A) => Boolean => List[A]
neighbs: Pos => List[Pos]
isAlive: List[Pos] => Pos => Bool
```

Haskell

```
length :: [a] -> Int
filter :: (a -> Bool) -> [a] -> [a]
neighbs :: Pos -> [Pos]
isAlive :: Board -> Pos -> Bool
```

What we can do is use **Scala**'s predefined **length** and **filter** functions to define the equivalent of **Haskell**'s **length** and **filter** functions. i.e. we can define two anonymous functions that have the signatures we desire, but are implemented in terms of the **Scala**'s predefined **length** and **filter** functions.

So we are going to replace the following **Haskell function composition**

```
liveneighbs = length . filter(isAlive b) . neighbs
```

with the following **Scala** pseudocode

```
liveneighbs = λx.x.length compose λx.x.filter(isAlive(b)) compose neighbs
```

which maps to the following **Scala** code:

```
def liveneighbs(b:Board): Pos => Int =
  ((x:List[Pos]) => x.length) compose ((x:List[Pos]) => x.filter(isAlive(b))) compose neighbs
```

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

                           length         [a] -> Int
                           filter         (a -> Bool) -> [a] -> [a]
                          isAlive         [Pos] -> Pos -> Bool
                                b         [Pos]
                      (isAlive b)         Pos -> Bool
               filter(isAlive b)          [Pos] -> [Pos]
                          neighbs         Pos -> [Pos]
       filter(isAlive b) . neighbs        Pos -> [Pos]
length . filter(isAlive b) . neighbs      Pos -> Int

liveneighbs =        length          .        filter(isAlive b)     .       neighbs
[Pos] -> Int ::      [Pos] -> Int    .        [Pos] -> [Pos]        .       Pos -> [Pos]
```

```
def liveneighbs(b:Board): Pos => Int =
  (((x:List[Pos]) => x.length) compose ((x:List[Pos]) => x.filter(isAlive(b)))) compose neighbs
```

λx.x.length compose λx.x.filter(isAlive(b)) compose neighbs (pseudo code)

```
                                        λx.x.length      List[A] => Int
                                             filter      (A) => Boolean => List[A]
                                            isAlive      List[Pos] => Pos => Bool
                                                  b      List[Pos]
                                        (isAlive b)      Pos => Bool
                              λx.x.filter(isAlive(b))    List[Pos] => List[Pos]
                                            neighbs      Pos => List[Pos]
            λx.x.filter(isAlive(b)) compose neighbs      Pos => List[Pos]
λx.x.length compose λx.x.filter(isAlive(b)) compose neighbs      Pos => Int

liveneighbs =       λx.x.length          compose    λx.x.filter(isAlive(b))   compose   neighbs
Pos => Int  :       List[Pos] => Int     compose    List[Pos] => List[Pos]    compose   Pos => List[Pos]
```

We can stay faithful to the Haskell code by defining a couple of anonymous functions that wrap Scala's predefined **length** and **filter** functions and provide us with **length** and **filter** functions that have the same signatures as the Haskell **length** and **filter** functions.

@philip_schwarz

We can do better than this though:

```
((x:List[Pos]) => x.length) compose ((x:List[Pos]) => x.filter(isAlive(b))) compose neighbs
```

If we turn the two anonymous functions into named methods,

```
def length(b: Board): Int =b.length

def filter[A](f: A => Boolean)(b: List[A]): List[A] =b filter f
```

then the function composition looks pretty much the same as in **Haskell**

```
length _ compose filter(isAlive(b)) compose neighbs _
```

(the underscores are needed to convert the length and filter methods to functions). Alternatively, if we just use **Scala**'s **length** and **filter** functions the way they are intended to be used, the result is pretty clear

```
neighbs(p).filter(isAlive(b)).length
```

So I think in this case it is not worth bothering with function **composition**.

By the way, just for fun, if in the above, we insert a space either side of each dot, then it looks deceptively similar to the **Haskell function composition**!!!

```
neighbs(p) . filter(isAlive(b)) . length
```

Haskell

```
length . filter(isAlive b) . neighbs
```

So that was the reason why I translated the **liveneighbs** function the way I did below, i.e. without using the **high order function** for **function composition**.

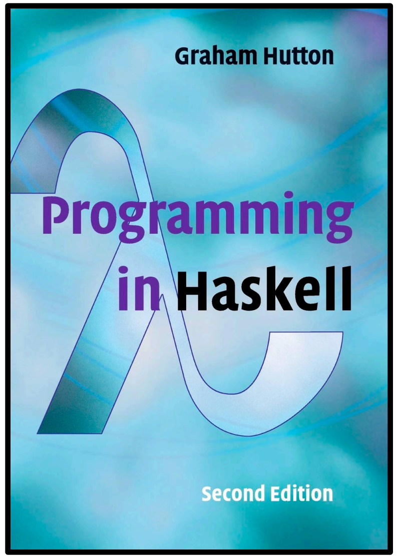
```haskell
liveneighbs :: Board -> Pos -> Int
liveneighbs b =
  length . filter(isAlive b) . neighbs

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


births :: Board -> [Pos]
births b =
  [(x,y) | x <- [1..width],
           y <- [1..height],
           isEmpty b (x,y),
           liveneighbs b (x,y) == 3]
```

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

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 {
    x <- List.range(1,width + 1)
    y <- List.range(1,height + 1)
    if isEmpty(b)((x,y))
    if liveneighbs(b)((x,y)) == 3
  } yield (x,y)
```

Next, we go back to **Graham Hutton**'s book, at the point where he has just introduced the **births** function.

@philip_schwarz

Graham Hutton
@haskellhutt




```haskell
births :: Board -> [Pos]
births b =
  [(x,y) | x <- [1..width],
           y <- [1..height],
           isEmpty b (x,y),
           liveneighbs b (x,y) == 3]
```

However, this definition considers **every position** on the **board**. A **more refined approach**, which may be **more efficient for larger boards**, is to **only consider the neighbours of living cells, because only such cells can give rise to new births**. Using this approach, the function **births** can be rewritten as follows:

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

The auxiliary function **rmdups** removes duplicates from a list, and is used above to ensure that each potential new cell is only considered once:

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

The **next generation** of a **board** can now be produced simply by appending the list of **survivors** and the list of **new births**

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

Let’s write the **Scala** equivalent of the **Haskell** functions we have just seen.

```haskell
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)


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

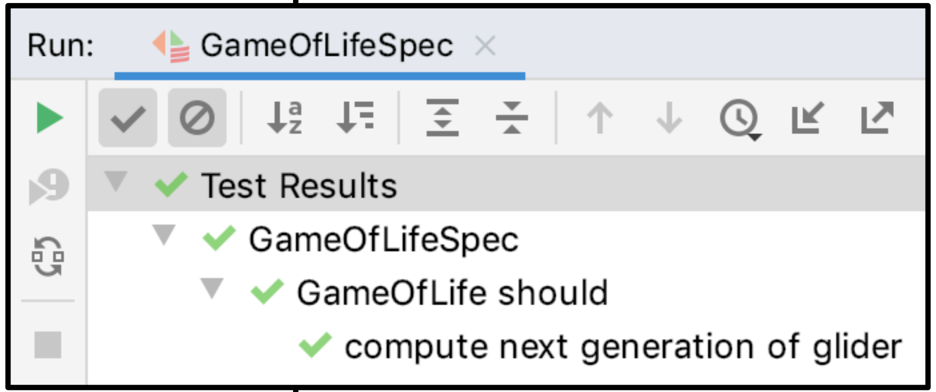
```scala
def births(b: Board): List[Pos] =
  for {
    p <- rmdups(b flatMap neighbs)
    if isEmpty(b)(p)
    if liveneighbs(b)(p) == 3
  } yield p

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

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

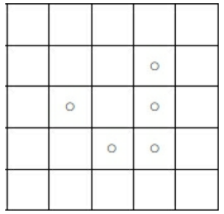
While a literal translation of `(concat (map neighbs b))` would be `((b map neighbs).flatten)`,
I simplified the latter to `(b flatMap neighbs)`

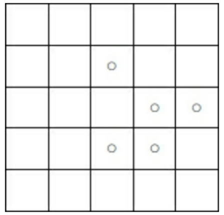
Let’s write a simple **Scala** test verifying that **nextgen** correctly computes the next **generation** of **glider**.

The test passes.

```scala
import gameoflife.GameOfLife._
import org.specs2.execute.Result
import org.specs2.mutable._

class GameOfLifeSpec extends Specification {

  "GameOfLife" should {
    "compute next generation of glider" in test
  }

  def test: Result = {
    val glider: Board = List((4,2),(2,3),(4,3),(3,4),(4,4))
    val gliderNext: Board = List((3,2),(4,3),(5,3),(3,4),(4,4))
    nextgen(glider) must containTheSameElementsAs(gliderNext)
  }

}
```







We have now finished looking at the **pure functions** needed to implement the **game of life.**

But a game cannot be implemented simply using **pure functions** because games need to interact with the outside world and so implementing them also requires **side-effecting functions**.

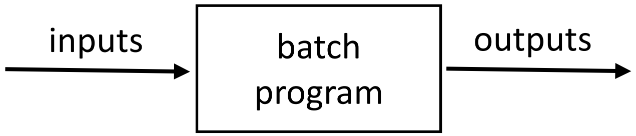
In the case of the **game of life**, it needs to display on the screen both the initial generation of live cells and the subsequent generations of live cells that it computes.

Before we look at the **side-effecting functions** required to implement **the game of life**, let's see how **Graham Hutton** introduces **the problem of modeling interactive programs as pure functions and how he explains the solution adopted by** Haskell.

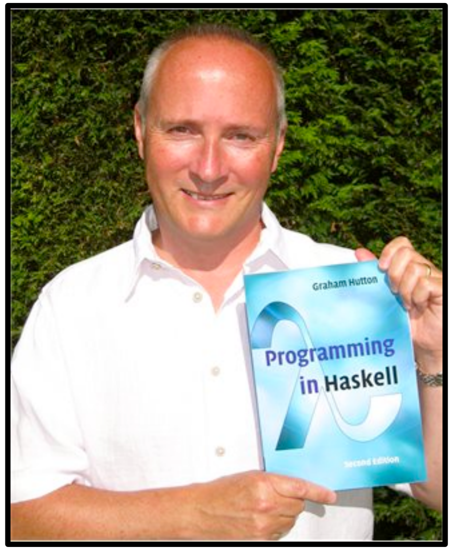
In the early days of computing, most programs were **batch programs** that were run in isolation from their users, to maximise the amount of time the computer was performing useful work.

For example, a compiler is a **batch program** that takes a high-level program as its input, silently performs a large number of operations, and then produces a low-level program as its output.

In part I of the book, we showed how **Haskell can be used to write batch programs**. **In Haskell such programs, and more generally all programs, are modelled as pure functions that take all their inputs as explicit arguments, and produce all their outputs as explicit results**, as depicted below:



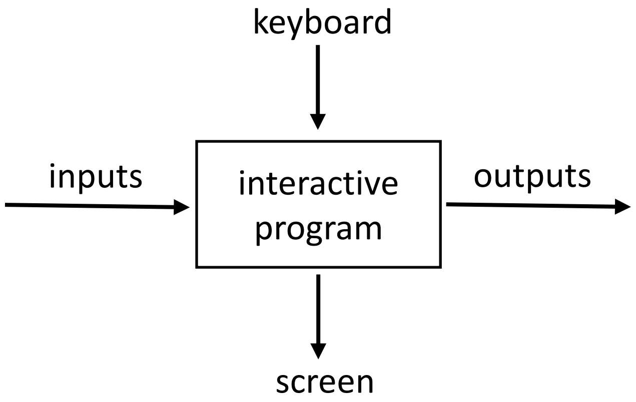
For example, a compiler such as GHC may be modelled as a function of type `Prog -> Code` that transforms a high-level program into low-level code.



Graham Hutton
@haskellhutt

In the modern era of computing, most programs are now interactive programs that are run as an ongoing dialogue with their users, to provide increased flexibility and functionality.

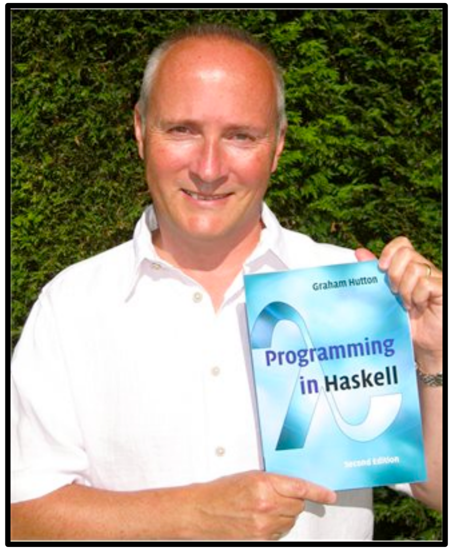
For example, an interpreter is an interactive program that allows expressions to be entered using the keyboard, and immediately displays the result of evaluating such expressions on the screen:



How can such programs be modelled as pure functions?

At first sight, this may seem impossible, because interactive programs by their very nature require the side-effects of taking additional inputs and producing additional outputs while the program is running.

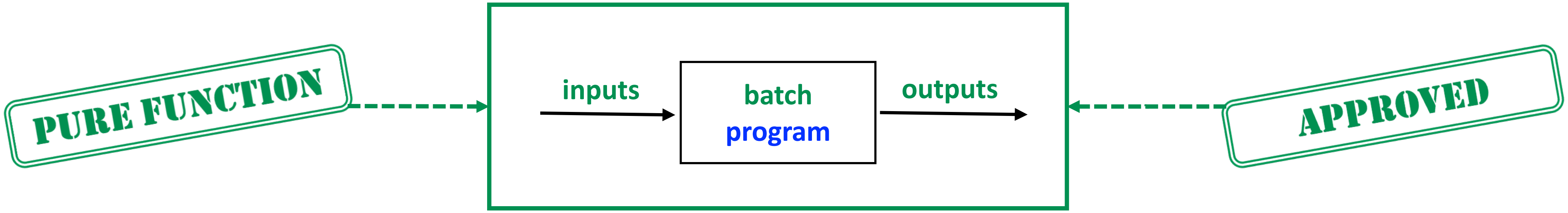
For example, how can an interpreter such as GHCi be viewed as a pure function from arguments to results?

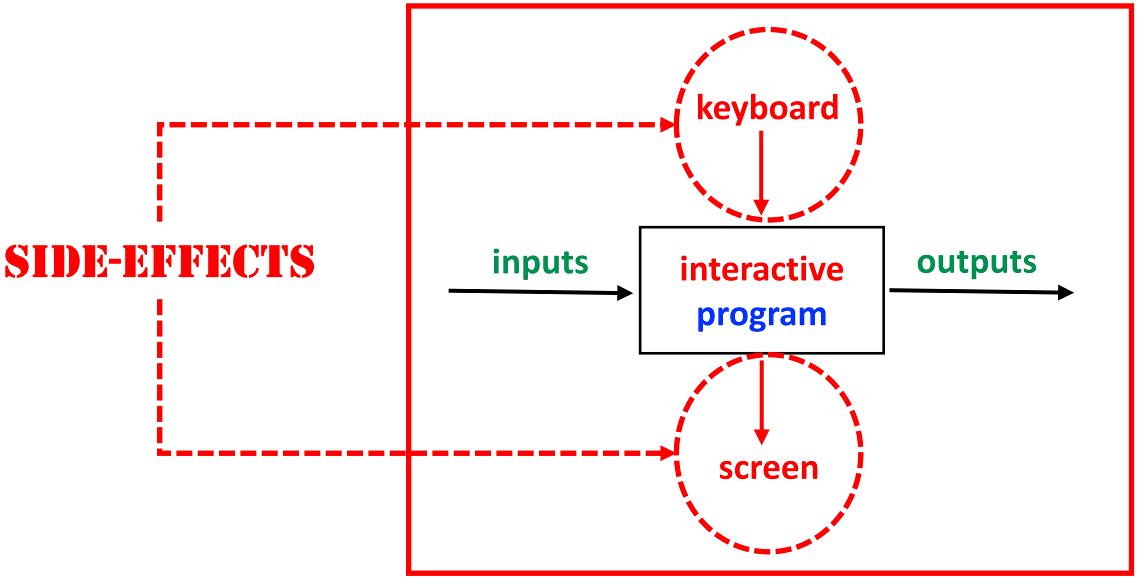


Graham Hutton
@haskellhutt

recap

@philip_schwarz





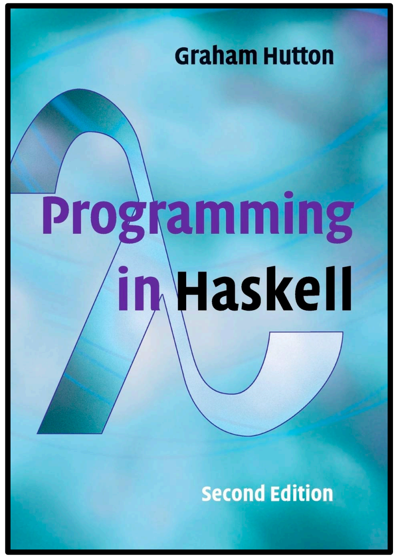
How can such programs be modelled as **pure functions**?

Graham Hutton

@haskellhutt

Graham Hutton

@haskellhutt



Over the years many approaches to the problem of combining the use of pure functions with the need for side-effects have been developed.

In the remainder of this chapter we present the solution that is used in Haskell, which is based upon a new type together with a small number of primitive operations.

As we shall see in later chapters, the underlying approach is not specific to interaction, but can also be used to program with other forms of effects.

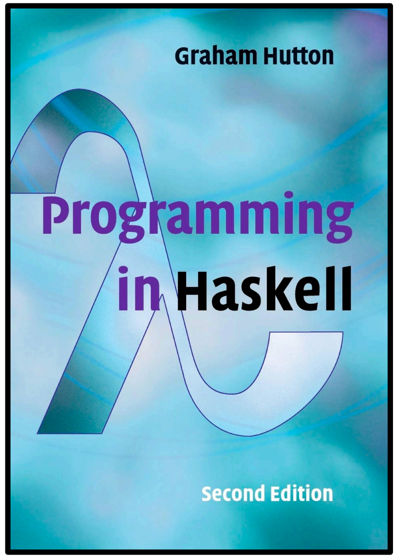
10.2 The solution

In Haskell an interactive program is viewed as a pure function that takes the current state of the world as its argument, and produces a modified world as its result, in which the modified world reflects any side-effects that were performed by the program during its execution.

Hence, given a suitable type `World` whose values represent states of the world, the notion of an interactive program can be represented by a function of type `World -> World` which we abbreviate as `IO` (short for input/output) using the following type declaration:

```
type IO = World -> World
```

Graham Hutton
@haskellhutt



In general, however, an interactive program may return a result value in addition to performing side-effects.

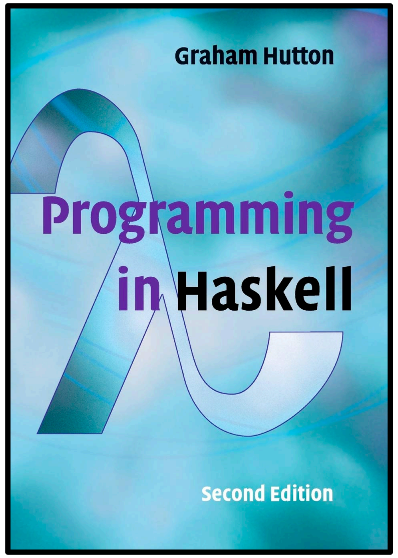
For example, a program for reading a character from the keyboard may return the character that was read.

For this reason, we generalise our type for interactive programs to also return a result value with the type of such values being a parameter of the `IO` type:

```
type IO a = World -> (a,World)
```

Expressions of type `IO a` are called actions. For example, `IO Char` is the type of actions that return a character, while `IO ()` is the type of actions that return the empty tuple `()` as a dummy result value. Actions of the latter type can be thought of as purely side-effecting actions that return no result value and are often useful in interactive programming.

Graham Hutton
@haskellhutt



In addition to **returning a result value**, **interactive programs** may also require **argument values**.

However, **there is no need to generalise the `IO` type further to take account of this**, because this behaviour can already be achieved by exploiting **currying**.

For example, an **interactive program** that takes a character and returns an integer would have type `Char -> IO Int`, which abbreviates the curried function type `Char -> World -> (Int,World)`.

At this point the reader may, quite reasonably, be concerned about the **feasibility of passing around the entire state of the world when programming with actions**! Of course, this isn't possible, and in reality the type `IO a` is provided as a primitive in Haskell, rather than being represented as a function type.

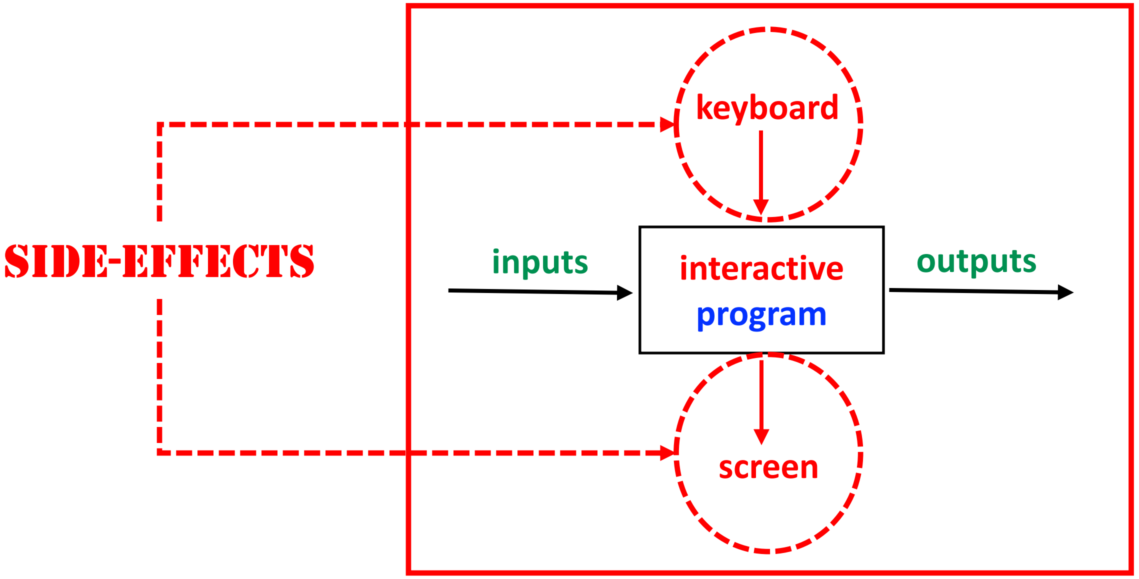
However, the above explanation is useful for understanding how actions can be viewed as pure functions, and the implementation of **actions** in **Haskell** is consistent with this view.

For the remainder of this chapter, **we will consider `IO a` as a built-in type whose implementation details are hidden**:

```
data IO a = ...
```

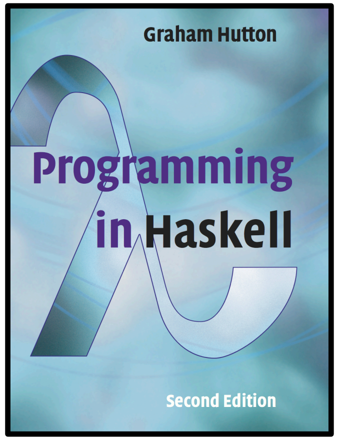
recap

Problem



How can such programs be modelled as **pure functions**?

Graham Hutton
@haskellhutt




```
type IO a = World -> (a,World)
```

`IO` : short for **input/output**

`Char` ⟶ `IO Int` — `IO` **action** returning an `Int`

`Char` ⟶ `World` ⟶ `(Int,World)`

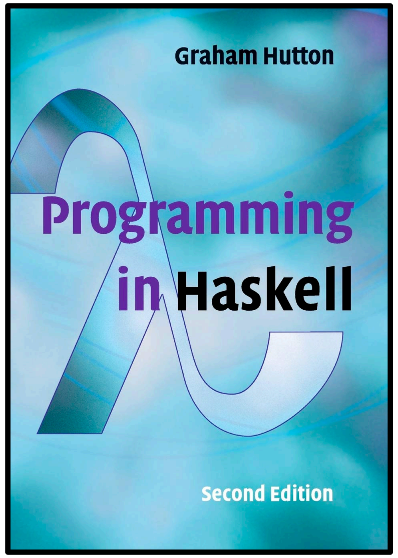
Solution

Haskell

‘A’ — current state of the world “ ” ⟶ **interactive program** ⟶ (65, “ ”) — modified world

reflects any **side-effects** that were performed by the program during its execution.

Graham Hutton
@haskellhutt



10.3 Basic actions

We now introduce three **basic IO actions** that are provided in **Haskell**. First of all, the action **getChar** reads a character from the keyboard, echoes it to the screen, and returns the character as its result value.

```
getChar :: IO Char
getChar = ...
```

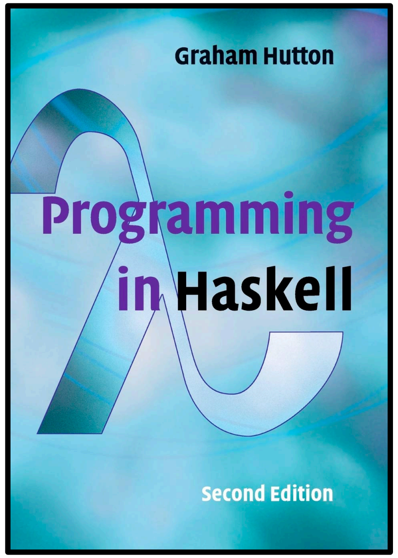
(The actual definition for **getChar** is built into the GHC system.) If there are no characters waiting to be read from the keyboard, **getChar** waits until one is typed. The **dual action**, **putChar c**, writes the character **c** to the screen, and **returns no result value, represented by the empty tuple**:

```
putChar :: Char -> IO ()
putChar c = ...
```

Our final **basic action** is **return v**, which **simply returns the result value v without performing any interaction with the user**:

```
return :: a -> IO a
return v = ...
```

Graham Hutton
@haskellhutt



The function **return** provides a **bridge** from **pure expressions** without **side-effects** to **impure actions** with **side-effects**.

```
return :: a -> IO a
return v = ...
```

Crucially, there is no **bridge** back — once we are **impure** we are **impure** for ever, with no possibility for redemption!

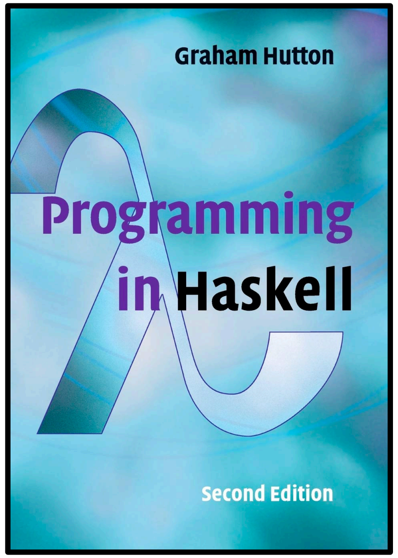
As a result, we may suspect that **impurity** quickly permeates entire programs, but in practice this is usually not the case. For most **Haskell** programs, the vast majority of functions do not involve **interaction**, with this being handled by a relatively small number of **interactive** functions at the outermost level.

pure expressions

`return :: a -> IO a`

side effects

Graham Hutton
@haskellhutt



10.4 Sequencing

In Haskell, a sequence of IO actions can be combined into a single composite action using the do notation, whose typical form is as follows:

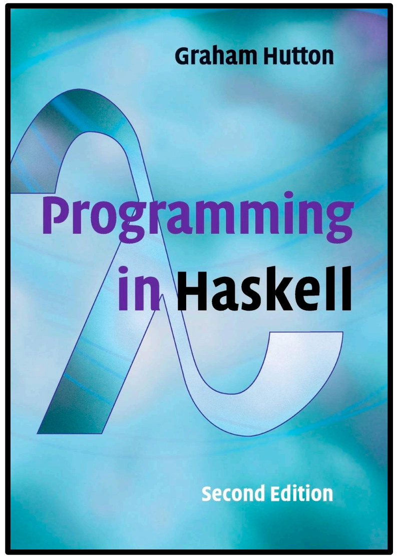
```
do v1 <- a1
   v2 <- a2
   .
   .
   .
   vn <- an
   return (f v1 v2 ... vn)
```

Such expressions have a simple operational reading: first perform the action a1 and call its result value v1; then perform the action a2 and call its result value v2; ...; then perform the action an and call its result value vn; and finally, apply the function f to combine all the results into a single value, which is then returned as the result value from the expression as a whole.

There are three further points to note about the do notation.

- First of all, the layout rule applies, in the sense that each action in the sequence must begin in precisely the same column, as illustrated above.
- Secondly, as with list comprehensions, the expressions vi <- ai are called generators, because they generate values for the variables vi.
- And finally, if the result value produced by a generator vi <- ai is not required, the generator can be abbreviated simply by ai, which has the same meaning as writing _ <- ai.

Graham Hutton
@haskellhutt

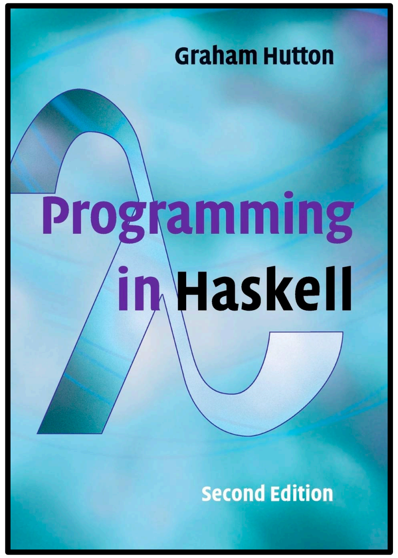


For example, an **action** that reads three characters, discards the second, and returns the first and third as a pair can now be defined as follows:

```
act :: IO (Char,Char)
act = do x <- getChar
         getChar
         y <- getChar
         return (x,y)
```

Note that omitting the use of **return** in this example would give rise to a type error, because (x,y) is an expression of type **(Char,Char)**, whereas in the above context we require an action of type **IO (Char,Char)**.

Graham Hutton
@haskellhutt



10.5 Derived primitives

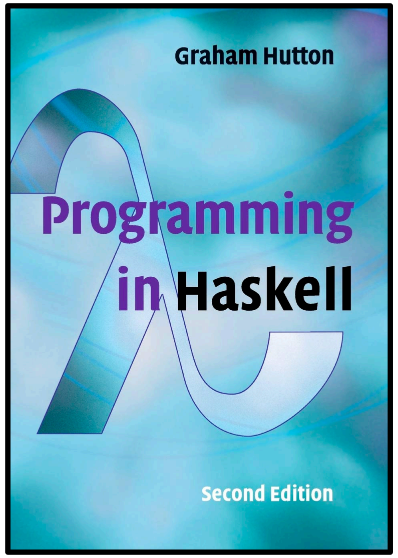
Using the three basic actions together with sequencing, we can now define a number of other useful action primitives that are provided in the standard prelude. First of all, we define an action getLine that reads a string of characters from the keyboard, terminated by the newline character '\n':

```
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then
               return []
             else
               do xs <- getLine
                  return (x:xs)
```

Note the use of recursion to read the rest of the string once the first character has been read. Dually, we define primitives putStr and putStrLn that write a string to the screen, and in the latter case also move to a new line:

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

putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

Graham Hutton
@haskellhutt



For example, using these primitives we can now define an action that prompts for a string to be entered from the keyboard, and displays its length:

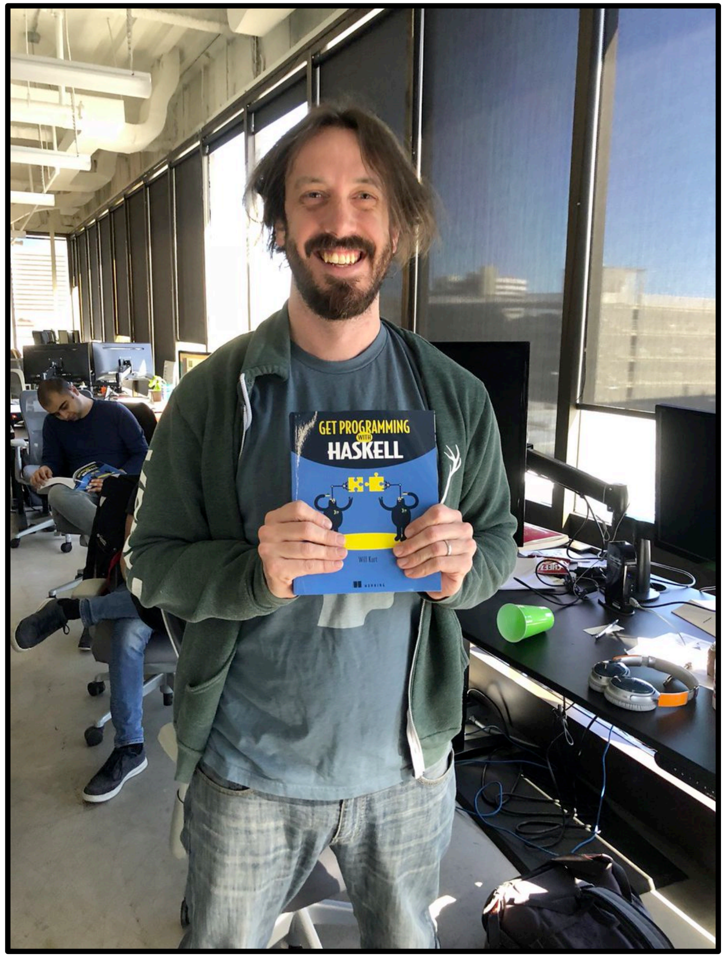
```haskell
strLen :: IO ()
strLen = do putStr "Enter a string: "
            xs <- getLine
            putStr "The string has "
            putStr (show (length xs))
            putStrLn " characters"
```

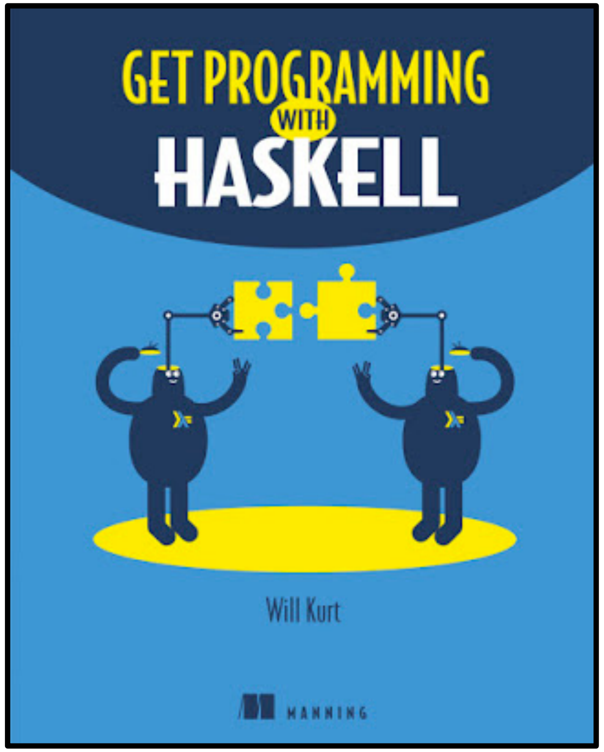
For example

```
> strLen
Enter a string: Haskell
The string has 7 characters
>
```

To reinforce and expand on the **IO** concept just explained by **Graham Hutton**, we now turn to **Will Kurt**'s book, **Get Programming with Haskell**.

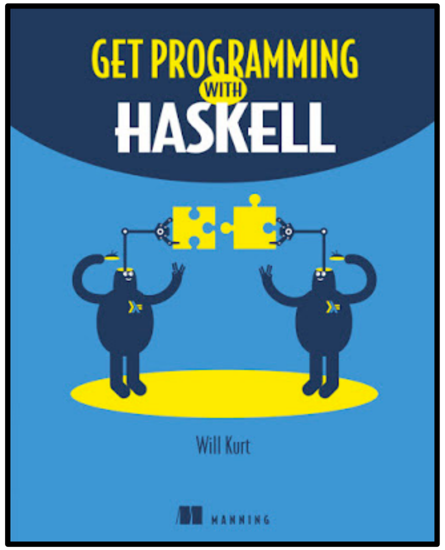
@philip_schwarz





Will Kurt
@willkurt

Will Kurt
@willkurt



In this lesson, you'll revisit a similar program to get a better sense of how **I/O** works in **Haskell**. Here's an example program using **I/O** that reads a **name** from the command line and prints out **"Hello <name>!"**.

```
helloPerson :: String -> String
helloPerson name = "Hello" ++ " " ++ name ++ "!"

main :: IO ()
main = do
   putStrLn "Hello! What's your name?"
   name <- getLine
   let statement = helloPerson name
   putStrLn statement
```

21.1. IO types—dealing with an impure world

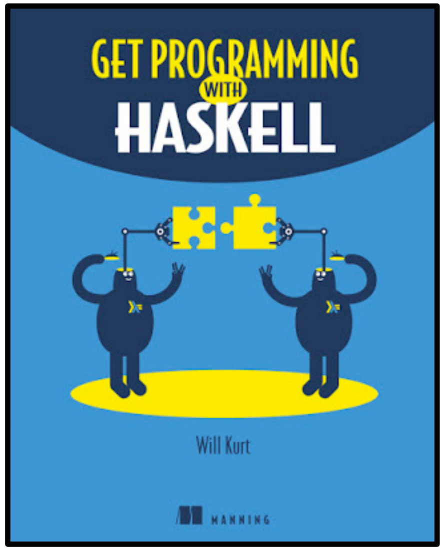
As is often the case with **Haskell**, if you're unsure of what's going on, it's best to look at the types!

The first type you have to understand is the **IO type**.

In the preceding unit, you ended by looking at the **Maybe type**. **Maybe** is a **parameterized type** (a type that takes another type as an argument) that **represents a context when a value may be missing**.

IO in Haskell is a parameterized type that's similar to Maybe. The first thing they share in common is that they're **parameterized types** of the **same kind**.

Will Kurt
@willkurt



You can see this by looking at the **kind** of **IO** and of **Maybe**:

```
> :kind Maybe
Maybe :: * -> *
> :kind IO
IO :: * -> *
>
```

The other thing that Maybe and IO have in common is that (unlike List or Map) they describe a context for their parameters rather than a container. The context for the IO type is that the value has come from an input/output operation. Common examples of this include reading user input, printing to standard out, and reading a file.

With a Maybe type, you're creating a context for a single specific problem: sometimes a program's values might not be there.

With IO, you're creating context for a wide range of issues that can happen with IO.

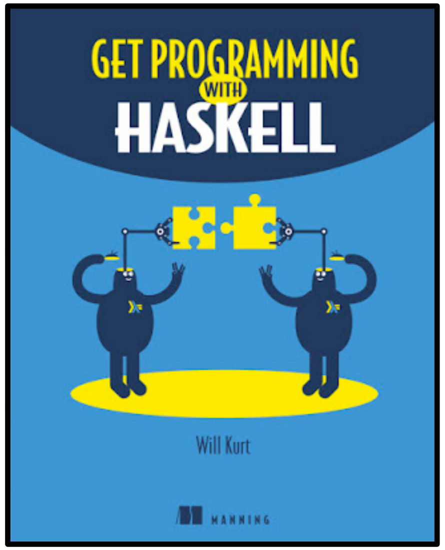
Not only is IO prone to errors, but it's also inherently stateful (writing a file changes something) **and also often impure** (calling getLine many times could easily yield a different result each time if the user enters different input).

Although these may be issues in I/O, they're also essential to the way I/O works.

What good is a program that doesn't change the state of the world in some way?

To keep Haskell code pure and predictable, you use the IO type to provide a context for data that may not behave the way all of the rest of your Haskell code does. IO actions aren't functions.

Will Kurt
@willkurt



In your example code, you only see one **IO type** being declared, the type of your **main**:

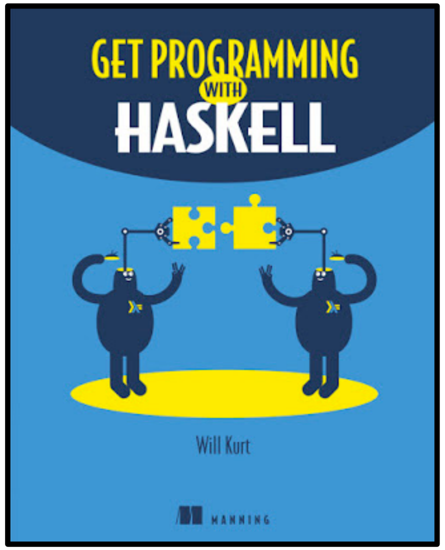
```
main :: IO ()
```

At first **()** may seem like a special symbol, but in reality **it's just a tuple of zero elements**. In the past, we've found tuples representing pairs or triples to be useful, but **how can a tuple of zero elements be useful?** Here are some similar types with **Maybe** so you can see that **IO ()** is just **IO** parameterized with (), and can try to figure out why () might be useful:

```
> :type Just (1,2)
Just (1,2) :: (Num a, Num b) => Maybe (a, b)
> :type Just (1)
Just (1) :: Num a => Maybe a
> :type Just ()
Just () :: Maybe ()
>
```

For **Maybe**, being parameterized with () is useless. It can have only two values, **Just** () and **Nothing**. But arguably, **Just** () ***is*** **Nothing**. It turns out that **representing nothing** **is exactly why you want to parameterize** **IO** **with an empty tuple**.

Will Kurt
@willkurt



You can understand this better by thinking about **what happens when your main is run**. Your last line of code is as follows:

```
putStrLn statement
```

As you know, this prints your statement. What type does **putStrLn** return? It has sent a message out into the world, but it's not clear that anything meaningful is going to come back. In a literal sense, **putStrLn** **returns nothing at all**. **Because Haskell needs a type to associate with your main, but your main doesn't return anything, you use the () tuple to parameterize your IO type**. **Because () is essentially nothing, this is the best way to convey this concept to Haskell's type system**.

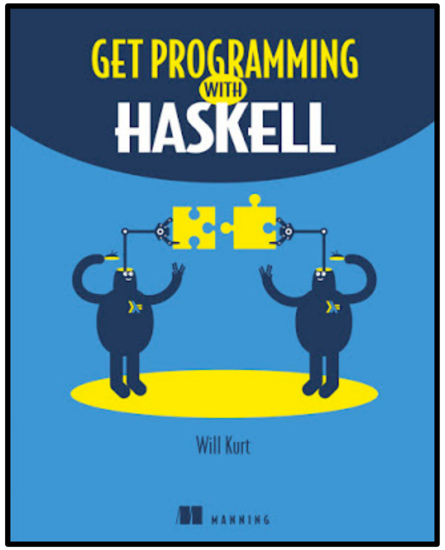
Although you may have satisfied **Haskell**'s type system, **something else should be troubling you about your main**. In the beginning of the book, we stressed **three properties of functions that make functional programming so predictable and safe**:

- **All functions must take a value.**
- **All functions must return a value.**
- **Anytime the same argument is supplied, the same value must be returned (referential transparency).**

Clearly, **main** doesn't return any meaningful value; it simply performs an ***action***. It turns out that **main *isn't* a function, because it breaks one of the fundamental rules of functions: it doesn't return a value**.

Because of this, **we refer to main as an *IO action***. **IO actions work much like functions except they violate at least one of the three rules** we established for functions early in the book. **Some IO actions return no value, some take no input, and others don't always return the same value given the same input**.

Will Kurt
@willkurt



21.1.1. Examples of IO actions

If main isn't a function, it should follow that neither is putStrLn. You can quickly clear this up by looking at **putStrLn**'s type:

```
putStrLn :: String -> IO ()
```

As you can see, **the return type of putStrLn is IO ()**. **Like main, putStrLn is an IO action because it violates our rule that functions must return values**.

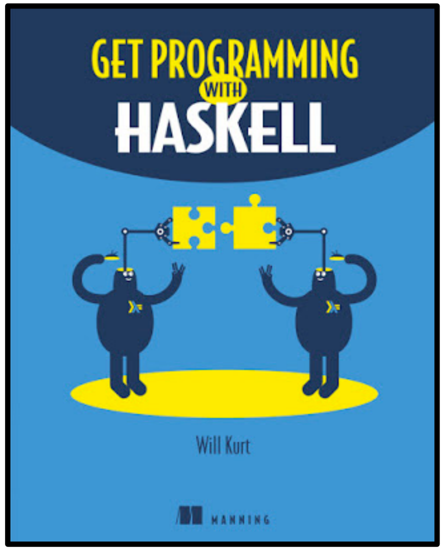
The next confusing function should be **getLine**. Clearly, this works differently than any other function you've seen because **it doesn't take an argument!** Here's the type for **getLine**:

```
getLine :: IO String
```

Unlike **putStrLn**, which takes an argument and returns no value, **getLine takes no value but returns a type IO String**. This means **getLine violates our rule that all functions must take an argument**. Because **getLine** violates this rule of functions, it's also **an IO action**.

Now let's look at **a more interesting case**. If you import **System.Random**, you can use **randomRIO**, which takes a pair of values in a tuple that represents the minimum and maximum of a range and then **generates a random number** in that range.

Will Kurt
@willkurt



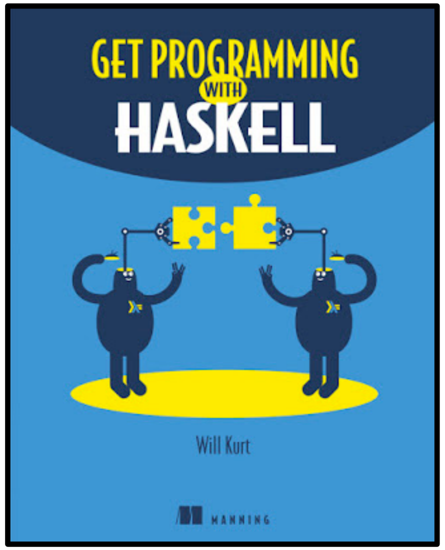
Here's a simple program called roll.hs that uses **randomRIO** and, when run, acts like rolling a die.

```
import System.Random

minDie :: Int
minDie = 1

maxDie :: Int
maxDie = 6

main :: IO ()
main = do
  dieRoll <- randomRIO (minDie,maxDie)
  putStrLn (show dieRoll)
```

You can compile your program with GHC and "roll" your die:

```
$ ghc roll.hs
$ ./roll
2
```

Will Kurt
@willkurt



What about **randomRIO**?

It takes an argument (the min/max pair) and returns an argument (an **IO type** parameterized with the type of the pair), so **is it a function**?

If you run your program more than once, you'll see the problem:

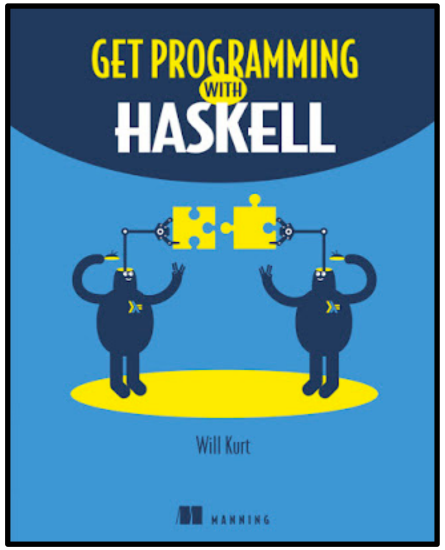
$ ./roll
4
$
./roll
6

Each time you call randomRIO, you get a different result, even with the same argument.

This violates the rule of referential transparency. So **randomRIO**, just like **getLine** and **putStrLn**, is an **IO action**.

Will Kurt

@willkurt



21.1.2. Keeping values in the context of IO

The interesting thing about **getLine** is that you have a useful return value of the type **IO String**. **Just as a Maybe String means that you have a type that might be missing**, **IO String means that you have a type that comes from I/O**.

In lesson 19 we discussed the fact that **a wide range of errors is caused by missing values that Maybe prevents from leaking into other code**. **Although null values cause a wide variety of errors**, **think of how many errors you've ever encountered caused by I/O**!

Because I/O is so dangerous and unpredictable, **after you have a value come from I/O**, **Haskell doesn't allow you to use that value outside of the context of the IO type**. For example, **if you fetch a random number using randomRIO**, **you can't use that value outside main or a similar IO action**. You'll recall that with **Maybe** you could use pattern matching to take a value safely out of the context that it might be missing. This is because only one thing can go wrong with a **Maybe** type: the value is **Nothing**. **With I/O**, **an endless variety of problems could occur. Because of this, after you're working with data in the context of IO, it must stay there.**

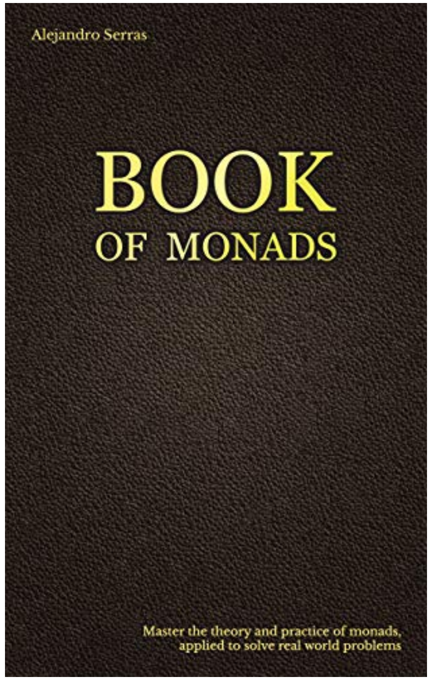
This initially may seem like a burden. After you're familiar with the way **Haskell separates I/O logic from everything else**, you'll likely want to replicate this in other programming languages (though you won't have a powerful type system to enforce it).

That was Get Programming with Haskell, by Will Kurt, and it was great! I found that material very very useful.

Before we go back to Graham Hutton's book, to see him complete his game of life program by writing the requisite impure functions, let's go over a few more aspects of IO actions which Alejandro Mena covers in his Book of Monads and which will help us not only to understand Graham's code but also to translate that code into Scala.

Alejandro Serrano Mena

@trupill



Interfacing with the Real World

The **IO monad** is as powerful as a spaceship but also as powerful as Pandora's box. **In Haskell, the IO monad grants access to external libraries, to the file system, to the network, and to an imperative model of execution**. **We need it** — we want to communicate with other systems or with the user, don't we? — **but we want to stay far away from it as much as possible. It is impossible to describe all the possibilities inherent in Haskell**'s **IO monad**. **For the sake of simplicity, we are going to restrict ourselves to simple actions**. **The following actions allow us to show and obtain information through the console**:

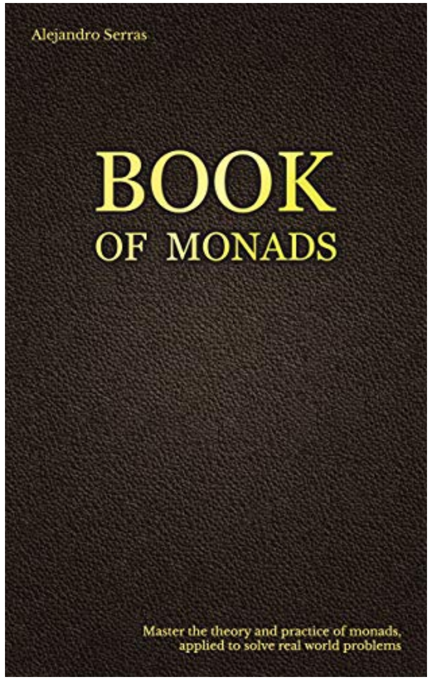
```
putStr   :: String -> IO ()
putStrLn :: String -> IO () -- end with a newline
getChar  :: IO Char
getLine  :: IO String
```

Using these **primitives**, we can write a simple program that asks for a name and uses it to print a greeting:

```
greet :: IO ()
greet = do putStr "Enter your name: "
           name <- getLine
           putStrLn (" Hello, " + + name + + "!")
```

Alejandro Serrano Mena

@trupill



Another functionality that lives in the **IO monad** is **randomness**. Each data type that supports a notion of a random value has to implement the **Random** type class. This means that the following two operations are available for that type:

```
randomIO  :: Random a => IO a
randomRIO :: Random a => (a, a) -> IO a -- within bounds
```

Purity.

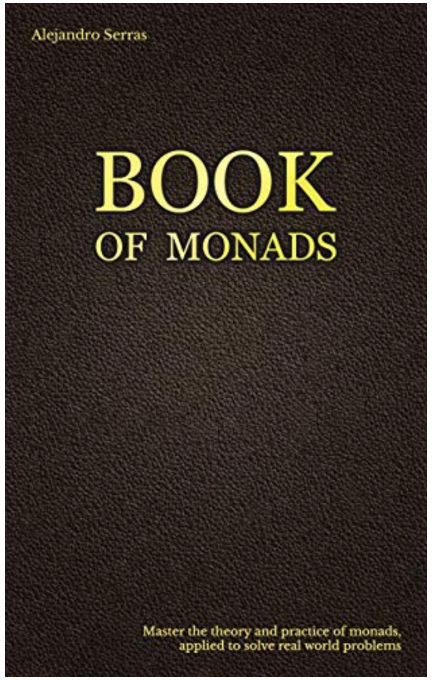
Haskellers **often emphasize that their language is** **purely functional**. **A** **pure language** **is one that embodies the idea that** “**equals can be substituted for equals**.” **This idea is also important in mathematics**. For example, we know that 1 + 2 = 3. This means that if we have an expression like (1 + 2) ^{2}, we can just turn it into 3^2. **Almost everything in** **Haskell** **works in this way**. For example, the definition of the length of a list tells us that:

```
length [] = 0
```

If we take the expression (length []) * 2, we can safely rewrite it as 0 * 2. This property also holds for local bindings, so we can turn **let** x = 0 in x * x into 0 * 0.

Alejandro Serrano Mena

@trupill



Imagine now that **random generation** would have the signature

```
random :: Random a => a.
```

The rule of "**equals can be substituted for equals**" tells us that

```
let r = random in r == r
```

could be rewritten to

```
random == random
```

But those two expressions have completely different meanings. In the first one we produce a random value once, which is checked with itself for equality, and thus always returns True. In the second case, two random values are generated, so the outcome is equally random.

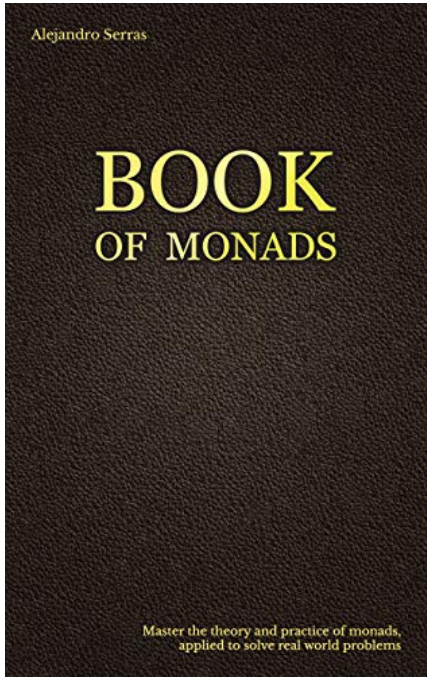
Haskell's **solution is to mark those values for which purity does not hold with IO**. **Since randomRIO generates two values of type IO a, we cannot directly apply the equality operator to them, as no instance for IO a exists**. In addition, the compiler knows that whereas it is safe to inline or manipulate any other expression in a program, it should never touch an **IO** action.

Here is the error we get when we try to apply the **equality operator** directly to two values of type **IO a** generated by **randomRIO** (since no **Eq** instance for **IO a** exists).

@philip_schwarz

```
> :t randomRIO
randomRIO :: Random a => (a, a) -> IO a

> randomRIO(1,10)
6

> randomRIO(1,10) == randomRIO(1,10)

<interactive>:42:1: error:
    • No instance for (Eq (IO Integer)) arising from a use of ‘==’
    • In the expression: randomRIO (1, 10) == randomRIO (1, 10)
      In an equation for ‘it’:
          it = randomRIO (1, 10) == randomRIO (1, 10)

> let r = randomRIO(1,10) in r == r

<interactive>:47:28: error:
    • No instance for (Eq (IO Integer)) arising from a use of ‘==’
    • In the expression: r == r
      In the expression: let r = randomRIO (1, 10) in r == r
      In an equation for ‘it’: it = let r = randomRIO ... in r == r
>
```

Alejandro
Serrano Mena
@trupill



Description versus execution.

IO values are treated like any other value in Haskell: they can be used as arguments to functions, put in a list, and so on. This raises the question of when the results of such actions are visible to the outside world.

Take the following small expression:

```
map putStrLn ["Alejandro", "John"]
```

If you try to execute it, you will see that nothing is printed on the screen. What we have created is a description of a list of actions that write to the screen. You can see this in the type assigned to the expression, [IO ()]. The fact that IO actions are not executed on the spot goes very well with the lazy nature of Haskell and allows us to write our own imperative control structures:

```
while :: IO Bool -> IO () -> IO ()
while cond action = do c <- cond
                       if c then action >> while cond action
                       else return ()
```

Such code would be useless if the actions given as arguments were executed immediately.

Alejandro said that if we take the expression ‘`map putStrLn ["Alejandro", "John"]`‘ and try to execute it, we will see that nothing is printed on the screen. Let’s try it:

```
> map putStrLn ["Alejandro", "John"]
<interactive>:158:1: error:
    • No instance for (Show (IO ())) arising from a use of ‘print’
    • In a stmt of an interactive GHCi command: print it
>
```

That’s in contrast to a single **IO action**, e.g. ‘`putStrLn "Alejandro"`‘

```
> :t putStrLn "Alejandro"
putStrLn "Alejandro" :: IO ()
```

which we *are* able to execute:

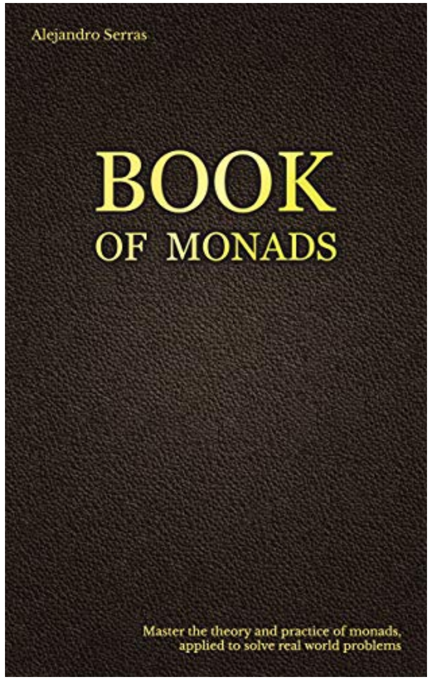
```
> putStrLn "Alejandro"
Alejandro
```

He also said that what we have created is a description of a list of **actions** that write to the screen and that we can see this in the type assigned to the expression, **[IO ()]**. Let’s see:

```
> :t map putStrLn ["Alejandro", "John"]
map putStrLn ["Alejandro", "John"] :: [IO ()]
```

On the next slide, Alejandro explains how to execute **IO actions** that are in a list.

Alejandro
Serrano Mena
@trupill



There are only two ways in which we can execute the description embodied in an IO action. One is entering the expression at the GHC interpreter prompt. The other is putting it in the call trace that starts in the main function of an executable. In any case, only those expressions that have IO as their outer constructor are executed. This is the reason why the previous expression would not print anything, even in main. To get the work done, we need to use sequence_ or mapM_:

```
sequence_ (map putStrLn ["Alejandro", "John"])
-- or equivalently
mapM_ putStrLn ["Alejandro", "John"]
```

This distinction between description and execution is at the core of the techniques explained in this book for creating your own, fine-grained monads. But even for a monad with so many possible side-effects like IO, it is useful for keeping the pure and impure parts of your code separated.

@philip_schwarz

Alejandro said that to get the work done, we need to use **sequence_** or **mapM_**:

```
sequence_ (map putStrLn ["Alejandro", "John"])
-- or equivalently
mapM_ putStrLn ["Alejandro", "John"]
```

What happens if we pass a list of **IO actions** to **sequence_** ?

```
> :t sequence_ (map putStrLn ["Alejandro", "John"])
sequence_ (map putStrLn ["Alejandro", "John"]) :: IO ()
>
```

It returns a single **IO action** of an empty tuple. Let's do it:

```
> sequence_ (map putStrLn ["Alejandro", "John"])
Alejandro
John
>
```

So the IO actions in the list got executed, their results were ignored, and a single **IO** of an empty tuple was returned. Similarly for **mapM_**:

```
> mapM_ putStrLn ["Alejandro", "John"]
Alejandro
John
>
```

What are the signatures of **sequence_** and **mapM_** and where are they defined?

```
> :info sequence_ mapM_
sequence_ :: (Foldable t, Monad m) => t (m a) -> m ()
  -- Defined in ‘Data.Foldable’
mapM_ :: (Foldable t, Monad m) => (a -> m b) -> t a -> m ()
  -- Defined in ‘Data.Foldable’
>
```

They are defined in **Foldable**.

As we see below, **sequence_** executes the **monadic actions** in a **Foldable** structure from left to right, ignoring the results. On the previous slide we saw that it executed the **IO actions** in a list. As for **mapM_**, it maps a function that returns a **monadic action** (e.g. an **IO action**) onto a **Foldable** structure (e.g. a list) and then does the same as **sequence_** with the result.

mapM_ :: (Foldable t, Monad m) => (a -> m b) -> t a -> m () # Source

Map each element of a structure to a monadic action, evaluate these actions from left to right, and ignore the results. For a version that doesn't ignore the results see `mapM`.

As of base 4.8.0.0, `mapM_` is just `traverse_`, specialized to `Monad`.

sequence_ :: (Foldable t, Monad m) => t (m a) -> m () # Source

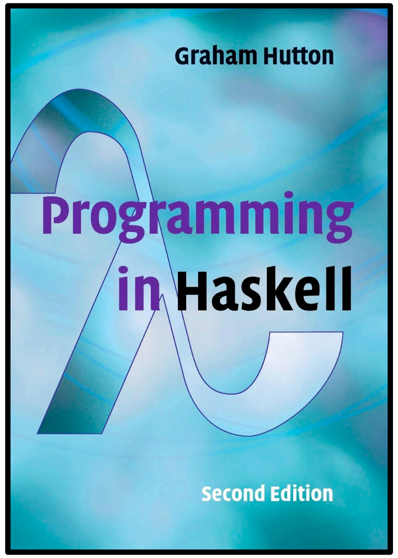
Evaluate each monadic action in the structure from left to right, and ignore the results. For a version that doesn't ignore the results see `sequence`.

As of base 4.8.0.0, `sequence_` is just `sequenceA_`, specialized to `Monad`.

Armed with a pretty decent understanding of **IO actions**, let's now return to **Graham Hutton**'s book and watch him write the **impure functions** that are needed for the **game of life**.

Graham Hutton

@haskellhutt



Screen utilities

We begin with some useful **output utilities** concerning the **screen** on which the **game** will be played. First of all, we define an **action** that **clears the screen**, which can be achieved by **displaying the appropriate control characters**:

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

By convention, the position of each character on the **screen** is given by a pair (x,y) of positive integers, with (1,1) being the top-left corner. We represent such **coordinate positions** using the following type:

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

We can then define a function that displays a string at a given position by **using control characters to move the cursor** to this position:

```
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")
```

Let’s try out the **cls** and **writeat** utilities that we have just see by writing code that uses them to first clear the screen and then display a 3 x 3 grid of **X** characters in the top left corner of the screen.

We need to call **writeat** nine times, once for each coordinate pair (x,y) where both x and y are numbers in the range 1 to 3 inclusive.

Each call to **writeat** will result in an **IO ()** action, so we’ll be creating a list of nine such actions which we can then execute using the **sequence_** function that we saw earlier.

Finally, we call **writeat** again with a blank string to move the **cursor** to line 4 of the screen, so that the screen prompt that gets drawn by the REPL, after our program has run, is out of the way, on a separate line.

```haskell
main :: IO ()
main = do cls
          sequence_ [ writeat (x,y) "X" | x <- [1,2,3], y <- [1,2,3] ]
          writeat (1,4) ""
```

Now let’s run our program by typing **main** at the REPL:

```
XXX
XXX
XXX
>
```

Remember how on slide 9, when **Graham Hutton** was about to write his first **impure function**, we decided to temporarily skip that function with a view to coming back to it later?

On the next slide, **Graham Hutton** looks at a function for displaying **living cells** on the **screen**. Because that function is **side-effecting**, we'll skip it for now and come back to it later.

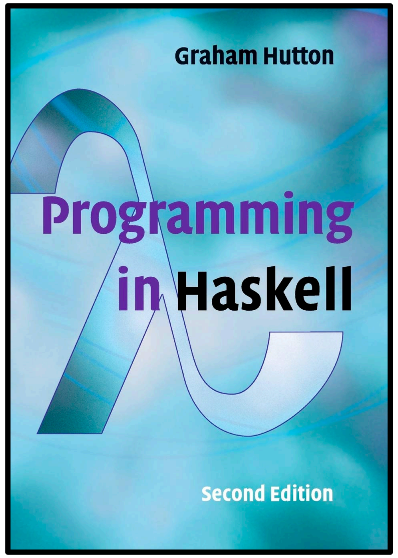
@philip_schwarz

Now is that time and here is the function, which is called **showcells** and uses the **sequence_** function:

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

For each point on the board, the function creates an **IO ()** action that prints a **O** character at the point's coordinates. The function then uses **sequence_** to execute all the resulting **IO () actions**.

Graham Hutton
@haskellhutt



Finally, we define a function **life** that implements the **game of life** itself, by clearing the **screen**, showing the **living cells** in the current **board**, **waiting** for a moment, and then continuing with the next **generation**:

```
life :: Board -> IO ()
life b = do cls
            showcells b
            wait 500000
            life (nextgen b)
```

The **life** function ends by calling itself, so it is **tail recursive** and able to run forever.

The function **wait** is used to **slow down** the game to a reasonable speed, and can be implemented by performing a given number of **dummy actions**:

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

For fun, you might like to try out the **life** function with the **glider** example, and **experiment with some patterns of your own**.

Let's run the **life program** with a 20 by 20 board configured with the **first generation** of a **Pulsar**, which cycles forever through three patterns.

```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)]

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

Screen pattern for the first generation of the **Pulsar**

```
  000   000

0    0 0    0
0    0 0    0
0    0 0    0
  000   000

  000   000
0    0 0    0
0    0 0    0
0    0 0    0

  000   000
```

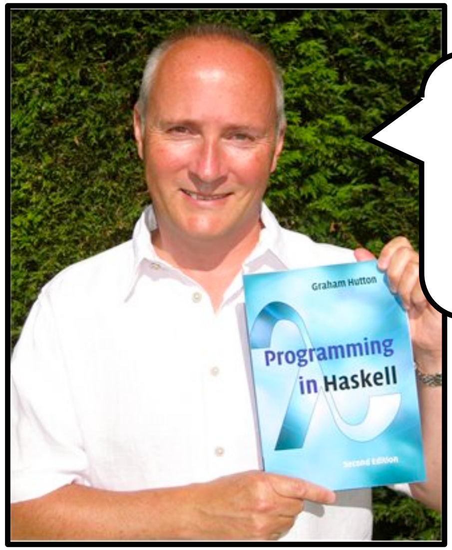
@philip_schwarz

Screen patterns for the three **generations** that the **Pulsar** cycles through.

```
  000   000

0    0 0    0
0    0 0    0
0    0 0    0
  000   000

  000   000
0    0 0    0
0    0 0    0
0    0 0    0

  000   000
```

```
    0     0
    0     0
    00   00

000  00 00  000
  0 0 0 0 0 0
    00   00

    00   00
  0 0 0 0 0 0
000  00 00  000

    00   00
    0     0
    0     0
```

```
  00     00
   00   00
0  0 0 0 0  0
000 00 00 000
 0 0 0 0 0 0
  000   000

  000   000
 0 0 0 0 0 0
000 00 00 000
0  0 0 0 0  0
   00   00
  00     00
```



Graham Hutton
@haskellhutt

Note also that most of the definitions used to implement the **game of life** are **pure functions**, with **only a small number of top-level definitions involving input/output**. Moreover, the definitions that do have such **side-effects** are clearly distinguishable from those that do not, through the presence of IO in their types.

In the next two slides we recap by seeing all the **game of life** code together, but split into **pure functions** and **impure functions**.

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

type Board = [Pos]

width :: Int
width = 10

height :: Int
height = 10
```

```haskell
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)
```

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

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

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

PURE FUNCTIONS

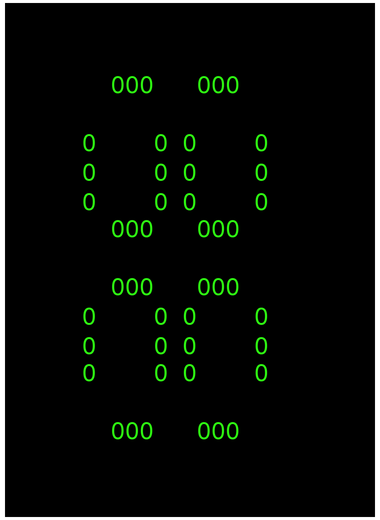
```haskell
survivors :: Board -> [Pos]
survivors b =
  [p | p <- b,
       elem (liveneighbs b p) [2,3]]

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)

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

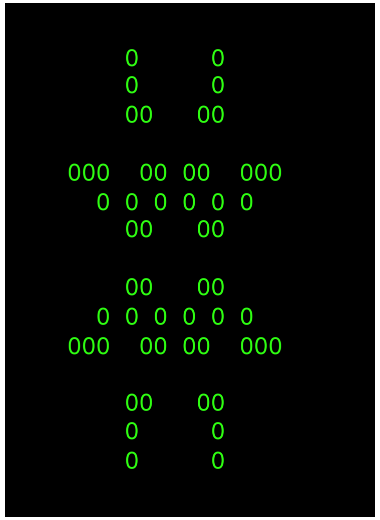
```haskell
glider :: Board
glider = [(4,2),(2,3),(4,3),(3,4),(4,4)]

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)]
```

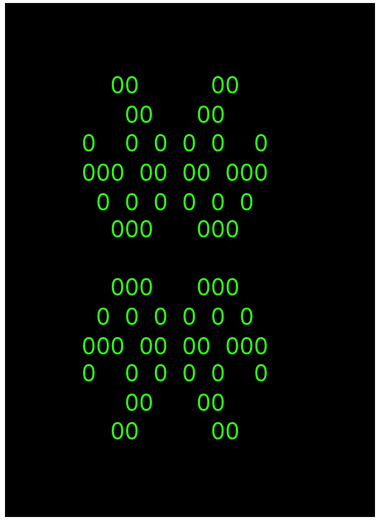
```haskell
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
                   putStr xs

putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'

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

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")
```

IMPURE FUNCTIONS

```haskell
life :: Board -> IO ()
life b = do cls
            showcells b
            wait 500000
            life (nextgen b)

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

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

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

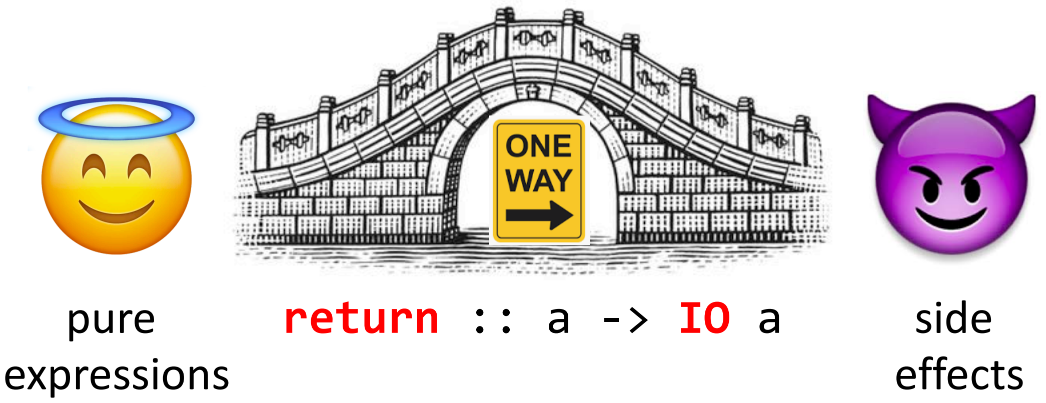


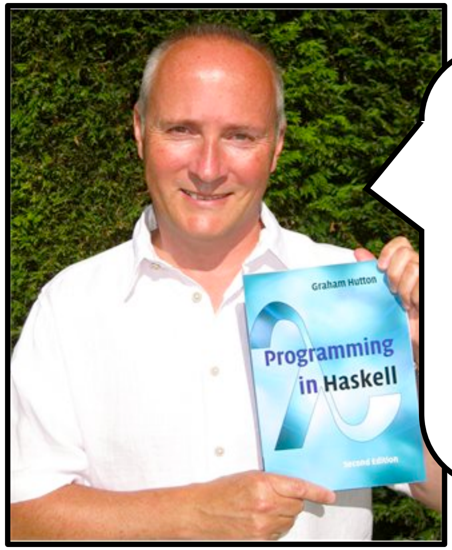




While I have also included **putsStr** and **putStrLn**, they are of course predefined (derived) **primitives**.

Remember when Graham Hutton explained how the return function provides a one-way bridge from pure expressions without side-effects to impure actions with side-effects?

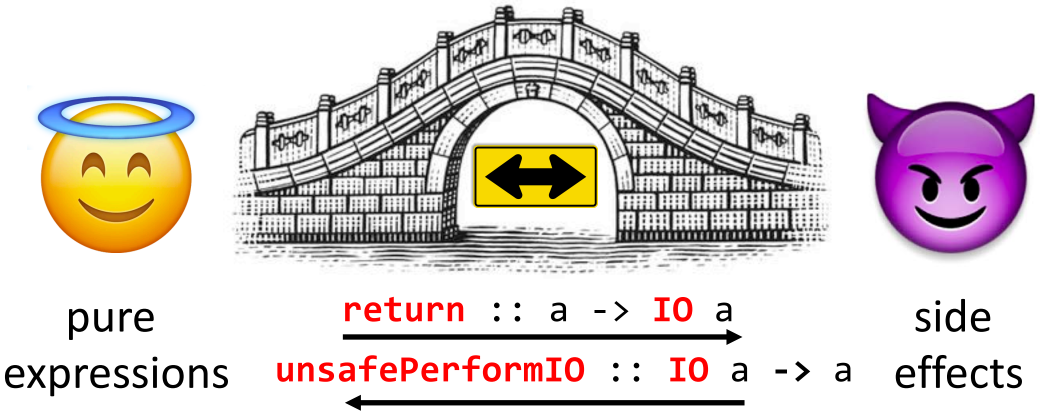




Graham Hutton
@haskellhutt

10.9 Chapter remarks

The use of the IO type to perform other forms of side effects, including reading and writing from files, is discussed in the Haskell Report [4], and a formal meaning for this type is given in [15]. For specialised applications, a bridge back from impure actions to pure expressions is in fact available via the function unsafePerformIO :: IO a -> a in the library System.IO.Unsafe. However, as suggested by the naming, this function is unsafe and should not be used in normal Haskell programs as it compromises the purity of the language.

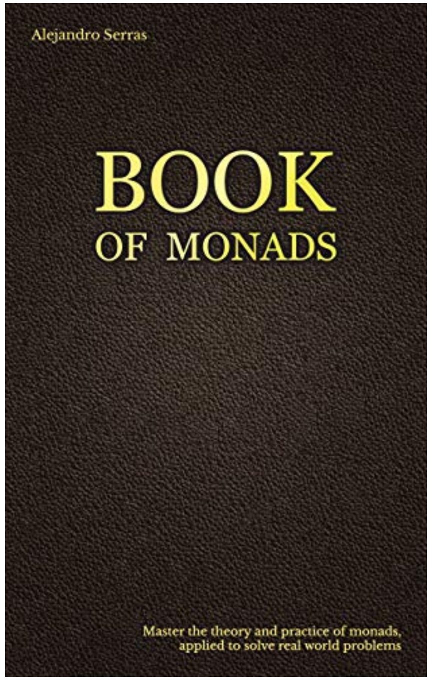


[15] S. Peyton Jones, “Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell,” in Engineering Theories of Software Construction. IOS Press, 2001.

[4] S. Marlow, Ed., Haskell Language Report, 2010, available on the web from: https://www.haskell.org/definition/haskell2010.pdf.

Alejandro Serrano Mena

@trupill



…people even use IO in languages in which side-effects are available everywhere, such as Scala. In particular, the Scalaz ZIO library defines an IOApp class as the entry point of a side-effectful computation, which is represented as an IO action. Note that in ZIO, the IO type takes two arguments — the first one represents the range of exceptions that may be thrown during the execution of the side-effects:

```
trait IOApp extends RTS {
  def run( args: List[ String]): IO[ Void, ExitStatus]
  final def main( args0: Array[ String]): Unit =
    unsafePerformIO( run( args0. toList))
    ...
}
```

```
unsafePerformIO :: IO a -> a
```

Wait a minute! You are now looking at the devil itself: unsafePerformIO. The type of that function is IO a -> a. In other words, it allows us to break the barrier between purity and impurity. You should know that this function exists only so you never use it. The situations in which you would need it are extremely rare and mostly involve interfacing with external systems. If the moment ever comes, you will know.

We conclude part 1 with a single slide recapping the **game of life** functions that we have translated into **Scala**, i.e. only the **pure functions**.

We still have a fair bit to do. In part 2 we will translate the **impure functions** into **Scala** using first a handrolled **IO monad** and then the **Cats Effect IO Monad**.

We will also translate the **game of life** program into the **Unison** language, which uses **Algebraic Effects** in preference to **Monadic Effects**.

PURE FUNCTIONS

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

type Board = List[Pos]

val width = 20

val height = 20
```

```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    ),
    (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) }
```

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

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

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

```scala
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

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

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

```scala
val glider: Board = List((4,2),(2,3),(4,3),(3,4),(4,4))

val gliderNext: Board = List((3,2),(4,3),(5,3),(3,4),(4,4))

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)])
```

To be continued in part 2