Introducció a la Programació Funcional en *Haskell*

Jordi Forga

Primavera 2023

Índex

1 Introducció

1.1 Qué és *Haskell*

Haskell és un llenguatge de programació funcional creat a final dels anys 1980 per un comité d'acadèmics. Va ser dissenyat agafant les millors idees de diversos llenguatges de programació funcional ja existents, i incorporant-hi de noves pròpies.

Haskell és un llenguatge de programació **funcional**, **pur**, **tipad estàticament** i **amb avaluació per necessitat**.

Programació Funcional

No hi ha un significat precís i acceptat del terme "funcional". Però quan parlem d'un llenguate funcional normalment significa dues coses:

1. Les funcions són de primera classe, és a dir, les funcions són valors que poden ser usats en el llenguatge igual que qualsevol altre classe de valor.
2. El significat dels programes funcionals es centra en *avaluar expressions* en lloc d'*executar instruccions*.

Els dos aspectes junts resulten en una manera totalment diferent de pensar la programació. L'objectiu d'aquesta pràctica consisteix en introdüir i explorar aquesta manera de pensar.

Funcions pures (sense efectes laterals)

En Haskell les expressions són sempre *referencialment transparents*, això vol dir:

- Cap modificació! Tot (variables, estructures de dades, ...) és *immutable*.
- Les expressions mai tenen "efectes laterals" (com la modificació de variables globals o la sortida a la pantalla).
- La crida de la mateixa funció amb els mateixos arguments resulta en la mateixa sortida cada vegada. Els programs són *deterministes*.

Això pot semblar absurd en aquest punt. Cóm és possible fer alguna cosa útil sense modificacions ni efectes laterals? Certament es requereix un canvi en la manera de pensar (sobretot si l'estudiant ha usat el paradigma imperatiu o orientat a objecte). Però una vegada fet aquest canvi apareixen una quantitat d'avantatges molt satisfactoris:

- *Raonament equacional i refactorització*: En Haskell sempre podem "substituir igual per igual", de la mateixa manera que hem après en una classe de matemàtiques o àlgebra.
- *Paral·lelisme*: L'avaluació d'expressions en paral·lel és fàcil i clara quan es garanteix que cada expressió no afecta les altres.
- *Menys mal de caps*: L'us de procediments amb efectes sense restriccions fa que els programes siguin difícils de depurar, mantenir i raonar.

Tipus estàtics

En Haskell, cada expressió té un tipus determinat en temps de compilació i que no canvia en el temps d'execució. Els programes amb errors de tipus no compilen, evitant problemes majors en temps d'execució.

Avaluació per necessitat (*lazy*)

En Haskell, les expressions *no són avaluades fins que el seu resultat és realment requerit*. Aquesta estratègia d'avaluació té importants conseqüències que veurem més endavant. Algunes d'aquestes conseqüències inclouen:

- Resulta fàcil obtenir noves *estructures de control* simplement definint funcions.
- És possible definir i operar amb *estructures infinites de dades*.
- Habilita un estil de programació per composició de components.
- El principal inconvenient, en canvi, és que el raonament sobre l'us del temps i de l'espai durant l'execució és molt més complicat!

1.2 Entorn de programació

En aquesta pràctica usarem l'eina `stack`, disponible per a la majoria de sistemes operatius, per a la construcció i execució dels diferents exercicis.

Els exercicis, a completar, d'aquesta primera pràctica es troben en el directori `prj/prac-prog-fun`. Aquests exercicis depenen, a més dels paquets proveïts per la pròpia eina `stack`, d'un paquet `drawing` per la creació i manipulació de simples objectes gràfics i la generació d'una sortida en format SVG. Això ens permetrà realitzar diferents exercicis amb una presentació visual dels resultats més clara i agradable.

Abans de realitzar els exercicis caldrà primer construir aquest paquet `drawing` i totes les dependències del projecte. Per això, ens situem al directori del projecte i fem:

```
~/dat-2023p$ cd prj
~/dat-2023p/prj$ stack build
```

Un cop construides totes les dependències del projecte, per executar els diferents exercicis ens situem al directori de la pràctica i usem `stack runghc` per executar l'exercici:

```
~/dat-2023p/prj$ cd prac-prog-fun
~/dat-2023p/prj/prac-prog-fun$ stack runghc NomFitxer.hs
```

2 Primera part: *Haskell* bàsic

Començem veient un simple exemple de programa:

```
module Main where
import Drawing

myDrawing :: Drawing
myDrawing = blank

main :: IO ()
main = putSvg myDrawing
```

Exercici 1

> Editeu un fitxer amb aquest programa i executeu-lo amb `stack runghc` passant-li com argument el nom del fitxer. El resultat és un contingut SVG que podreu visualitzar amb el navegador.

Qué veieu en el visualitzador? ... Res ! O millor dit, només el marc que envolta l'àrea de visualització (aquest marc és sempre generat per la funció `putSvg`).

2.1 Declaracions i variables

Modifiquem el programa canviant la corresponent línia per veure alguna cosa:

```
myDrawing = solidCircle 1
```

Ara veuriem un cercle sòlid de color negre i radi 1.

Aquest codi declara una variable amb el nom `myDrawing` i amb el tipus `Drawing`, i defineix el seu valor a l'expressió `solidCircle 1`.

Observeu que aquest *serà el valor de `myDrawing` per sempre* en aquest programa. El valor de `myDrawing` no es pot canviar.

Si intentem escriure un codi com el següent

```
myDrawing = solidCircle 1
myDrawing = solidCircle 2
```

aleshores el compilador ens informarà amb un missatge d'error `Multiple declarations of ‘myDrawing’`.

En Haskell, *les variables no són ubicacions mutables*; són exactament noms pels valors!

Dit d'altra manera, `=` no denota *assignació* com en molts altres llenguatges (imperatius). Denota *definició*, com en matemàtiques. Això vol dir que `myDrawing = solidCircle 1` no es pot llegir com "assigna `solidCircle 1` a `myDrawing`", sinó com "`myDrawing` *és* `solidCircle 1`".

2.2 El tipus Drawing i funcions

2.2.1 Funcions primitives

En la documentació del mòdul Drawing (pàgina doc/haddock/drawing-0.2.2.0/Drawing.html) podeu veure la declaració dels tipus de les funcions:

```
solidCircle :: Double -> Drawing
circle :: Double -> Drawing
rectangle :: Double -> Double -> Drawing
...
```

La fletxa (→) indica que `solidCircle` és una funció amb un paràmetre de tipus `Double` (número amb coma flotant) i un resultat de tipus `Drawing`.

`rectangle` és una funció amb 2 paràmetres (de tipus `Double`, i que corresponen a l'amplada i alçada). Exemple: expressió corresponent a un rectangle de dimensions 2x1:

```
rectangle 2 1
```

Observeu que l'aplicació d'una funció no reqereix indicar els arguments dins de parèntesis.

Ara bé si un argument és més complex que un simple nom o número, aleshores caldrà posar-lo entre parèntesis. Exemple:

```
rectangle 2 (3+4)
```

2.2.2 Modificació

La biblioteca usada en aquesta pràctica disposa de múltiples funcions per a modificar dibuixos (veieu la pàgina doc/haddock/drawing-0.2.2.0/Drawing.html). Per exemple, per a donar un color a un dibuix tenim la funció:

```
colored :: Color -> Drawing -> Drawing
```

Té 2 paràmetres: el color i el dibuix al qual es vol aplicar. En realitat no canvia el dibuix que es passa com a paràmetre sinó que obté el nou dibuix colorejat. En Haskell els valors no són mutables. Exemple:

```
myDrawing = colored green (solidCircle 1)
```

2.2.3 Composició

Com podem obtenir més d'un dibuix ? El que necessitem és una funció que obtingui la combinació de 2 dibuixos:

```
(<>) :: Drawing -> Drawing -> Drawing
```

Pot semblar un nom una mica extrany per a una funció, però es perfectament vàlid. Pot usar-se `(<>)` com un nom normal de funció; però també pot usar-se `<>` (sense parèntesis) com un operador binari.

Aquest operador combina 2 dibuixos obtenint el dibuix que consisteix amb el segon dibuix (el de la dreta de l'operador) pintat al damunt del primer (el de l'esquerra de l'operador).

Exemple:

```
myDrawing = solidCircle 2 <> colored green (solidCircle 1)
```

Cal tenir en consideració que *l'applicació de funció té major precedència que qualsevol operador binari.* Un altre exemple:

```
myDrawing = colored red (translated 0 1.5 (solidCircle 1)) <>
    colored green (translated 0 (-1.5) (solidCircle 1))
```

2.3 Definició de funcions

Hem vist com usar funcions ja definides, però també necessitem definir de noves. De fet quan escrivim `variable = expressio` estem definint una funció que no té paràmetres. Si volem que tingui paràmetres s'indiquen els seus noms a continuació del nom de la funció. Exemple:

```
botCircle c = colored c (translated 0 (-1.5) (solidCircle 1))
topCircle c = colored c (translated 0   1.5  (solidCircle 1))
frame = rectangle 2.5 5.5
trafficLight = frame <> topCircle red <> botCircle green

myDrawing :: Drawing
myDrawing = trafficLight
```

Exercici 2

Realitzeu un programa que generi el dibuix d'un semàfor centrat a l'origen amb 3 llums de colors vermell, groc i verd (la figura 1 mostra el dibuix resultant). Definiu una funció `lightBulb` que dibuixi un llum, amb 2 paràmetres corresponents al color i la posició en l'eix Y; i realitzeu el programa usant aquesta funció.

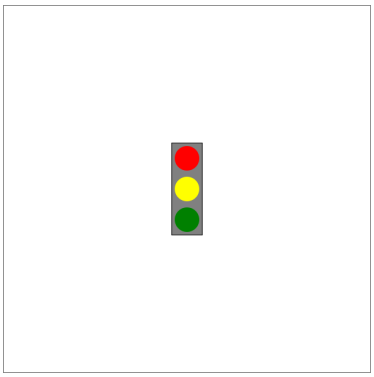

Figura 1: Resultat de l'exercici 2

NOTA: *Per ajudar en el disseny dels dibuixos, podeu mostrar el pla de coordenades canviant la definició de* **`main`**:

```
-- main = putSvg myDrawing          (definicio original)
main = putSvg (coordinatePlane <> myDrawing)  -- nova definicio
```

2.4 Funcions recursives

En Haskell no hi ha instruccions de control per a expressar iteracions. Com podem expressar per tant repeticions de coses ? La resposta és amb funcions recursives. Exemple:

```
lights :: Int -> Drawing
lights 0 = blank
lights n = trafficLight <> translated 3 0 (lights (n-1))

myDrawing = translated (-3) 0 (lights 3)
```

La definició de `lights` és una forma típica de recursió: tenim un cas base en el que no s'usa (invoca) la funció recursiva ($n == 0$), i tenim els casos recursius ($n > 0$).

Una altra manera equivalent[1]:

```
lights 0 = blank
lights n = translated (3 * fromIntegral n) 0 trafficLight <> lights
    (n - 1)

myDrawing = translated (-6) 0 (lights 3)
```

[1] Useu la funció `fromIntegral` per a la conversió de `Int` a `Double`.

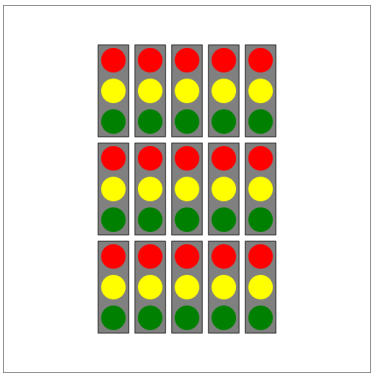

Figura 2: Resultat de l'exercici 3

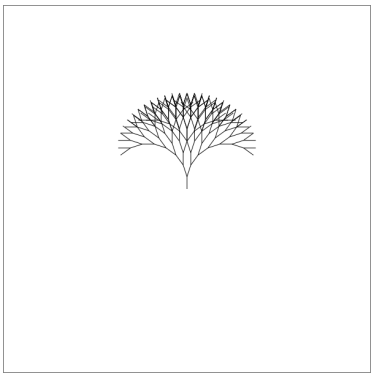

Figura 3: Resultat de l'exercici 4a

Proveu aquest programa que dibuixa una fila de 3 semàfors.

Exercici 3

Realitzeu un programa que dibuixi un *array* de 3 x 5 semàfors (la figura 2 mostra el dibuix resultant).

Exercici 4

a. Realitzeu un programa que dibuixi l'arbre de la figura 3 de 8 nivells.
b. Modifiqueu el programa de manera que dibuixi petites flors (simples cercles grocs) en les branques (veieu la figura 4).

2.5 Funcions d'ordre superior

L'exemple de la secció anterior i l'exercici 3 tenen un patró comú: repeteixen una cosa una quantitat de vegades. Per que no fem abstracció de la cosa i de la

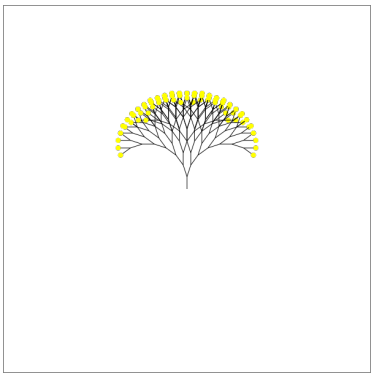

Figura 4: Resultat de l'exercici 4b

quantitat de vegades definint una funció més general i que puguem reutilitzar en els 2 casos ? Definirem una funció `repeatDraw` que repeteix `thing` una quantitat `n` de vegades. El dibuix de cada repetició depèn d'un enter que valdrà de 1 a n en cada repetició.

```
repeatDraw :: (Int -> Drawing) -> Int -> Drawing
repeatDraw thing 0 = blank
repeatDraw thing n = ...
```

Usant aquesta funció l'exercici 3 es pot realitzar de la següent manera:

```
myDrawing = translated (-9) (-16) (repeatDraw lightRow 3)

lightRow :: Int -> Drawing
lightRow r = repeatDraw (light r) 5

light :: Int -> Int -> Drawing
light r c = translated (3 * fromIntegral c) (8 * fromIntegral r)
    trafficLight
```

Observeu com la funció `light` (de 2 paràmetres) s'aplica **parcialment** a un sol paràmetre `r` en `lightRow`.

Exercici 5

Completeu la funció `repeatDraw` i executeu el programa anterior.

En la biblioteca de Haskell ja existeix un funció més general que `repeatDraw` per resoldre el problema general del recorregut d'una estructura i reducció a un valor resultant:

```
foldMap :: (Foldable t, Monoid m) => (a -> m) -> t a -> m
```

La classe `Monoid m` defineix una operació binària associativa i l'element neutre d'aquesta operació:

```
(<>) :: m -> m -> m
mempty :: m
```

Exercici 6

Resoleu el següent problema:
`trafficLights :: [(Double, Double)] -> Drawing`

> Donada una llista de n punts dibuixar n semàfors situats en les corresponents posicions, usant les funcions `foldMap` i `trafficLight`).

Observació: El tipus llista pertany a la classe `Foldable` i el tipus `Drawing` pertany a la classe `Monoid`. En el monoid `Drawing` l'operació `(<>)` és la composició de figures i l'element neutre és `blank`.

Aleshores el mateix problema (del exercici 3) es pot resoldre trivialment aplicant la funció `trafficLights` a la llista dels 15 punts corresponents.

3 Segona part: Interacció i modificació d'estat

En la segona part d'aquesta pràctica realitzarem una simulació del Joc de la Vida, inventat per John Conway l'any 1970. Consistirà en una aplicació que visualitza gràficament l'estat del joc (conjunt de cel·les vives) i en la que l'usuari podrà interactuar modificant l'estat, visualitzant-se el nou estat resultant.

IMPORTANT: Per realitzar aquesta segona part de la pràctica cal que us situeu en el directori `prj/prac-prog-fun/part2`. Aquest serà el directori de treball en aquesta segona part de la pràctica i ens hi referirem com el **directori de la part2**..

3.1 Creació d'una aplicació interactiva

Però ens podem preguntar: Cóm podem tenir interacció en un món sense efectes laterals ? Cóm podem guardar l'estat actual de l'aplicació en un món sense variables modificables ?

Un programa interactiu canvia l'estat quan es produeix un nou esdeveniment d'entrada. Bàsicament doncs, l'aplicació la podrem veure com una funció pura que obté el nou estat a partir de l'esdeveniment i de l'estat actual.

La biblioteca *Drawing* usada en aquesta pràctica ofereix la funció activityOf amb el següent tipus:

```
activityOf :: Double -> Double
           -> state
           -> (Event -> state -> state)
           -> (state -> Drawing)
           -> IO ()
```

Aquesta és una declaració polimòrfica, on `state` és una variable de tipus que fa referència a un tipus qualsevol corresponent a l'estat de l'aplicació.

La funció té els 5 paràmetres següents:

- Les dimensions (amplada i alçada) en unitats gràfiques que tindrà l'àrea de visualització. Cal tenir en compte que cada unitat ocupa 20 pixels i que l'origen de coordenades està en el centre de l'àrea. Valor adeqüats d'aquests paràmetres per a la majoria de navegadors són `60.0` i `30.0` respectivament.
- L'estat inicial de l'aplicació.
- La funció que a partir de l'esdeveniment d'entrada i de l'estat actual obté el nou estat.
- Y la funció que obté la sortida gràfica a partir de l'estat actual.

La funció `activityOf` obté una aplicació (servei HTTP) que mantindrà l'estat i processarà els esdeveniments generats des d'una pàgina HTML que interacciona amb l'usuari.

Veiem un exemple molt simple d'aplicació interactiva. Dins el directori de la part2, teniu el fitxer `exemple.hs` amb el següent contingut:

```
{-# LANGUAGE OverloadedStrings #-}

module Main where
import Drawing

main :: IO ()
main = activityOf 60 30 initial handle draw

initial = 0

handle (KeyDown " ") angle = angle + pi / 4
handle _             angle = angle

draw angle =
    rotated angle $
        polyline [(0, 0), (2, 0)]
          <> translated 2 0 (colored red $ solidCircle 0.5)
```

En aquest exemple observem que:

- L'estat de l'aplicació és un número de tipus `Double` que representa un angle en radians.
- L'estat inicial és `0` radians.
- Quan es produeix l'esdeveniment de tecla d'espai polsada s'incrementa l'angle en `pi/4` radians. Per a qualsevol altre esdeveniment l'angle no canvia.
- La visualització de l'estat (angle) consisteix en un simple dibuix rotat.

L'aplicació s'executarà fent:

```
stack runghc -- exemple.hs -p PORT
```

L'aplicació consisteix en un servidor HTTP pel port indicat amb l'opció `-p`.

Un cop instalat, amb un navegador visiteu l'exemple amb la URL `http://localhost:PORT/` (on *PORT* és el port indicat amb la opció `-p` de l'aplicació).

3.2 Realització del joc

La pràctica consistirà en la realització de 5 passos. En cadascún dels passos anirem introdüint noves funcionalitats fins arribar a la versió final (podeu provar el funcionament de la versió final).

El codi de cada pas estarà format pel mòdul principal `Main` (fitxers `life-1.hs` ... `life-4.hs`) i els mòduls `Life.Board` i `Life.Draw`. Els mòduls `Life.Board`

i `Life.Draw` són comuns a tots els passos i contenen la definició del taulell del joc i de la funció que el dibuixa respectivament.

La definició del taulell del joc i de les seves funcions es dona ja feta en el mòdul `Life.Board` (veieu el fitxer prj/prac-prog-fun/part2/Life/Board.hs). Aquest mòdul defineix el tipus `Board` i les funcions seguents:

```
-- The type of the board
data Board = ...

-- Each cell in the board is determined by a pair of Int's (column,
    row)
type Pos = (Int, Int)

-- The empty board (all cells are dead)
initBoard :: Board

-- 'cellIsLive pos board' is true iff the cell at 'pos' of 'board'
    is live
cellIsLive :: Pos -> Board -> Bool

-- 'liveCells board' is the list of the positions of the live cells
    of 'board'
liveCells :: Board -> [Pos]

-- Get the minimum position (column, row) of the live cells
minLiveCell :: Board -> Pos

-- Get the maximum position (column, row) of the live cells
maxLiveCell :: Board -> Pos

-- 'setCell live pos board' change the liveness of cell at 'pos' of
    'board' (to live if 'live' is 'True' or to dead if 'live' is '
    False')
setCell :: Bool -> Pos -> Board -> Board

-- 'nextGeneration board' gets the next generation of 'board'
nextGeneration :: Board -> Board
```

3.2.1 Primer pas

En el primer pas de la pràctica l'estat del joc consistirà senzillament en el taulell del joc.

El mòdul principal d'aquest primer pas (veieu el fitxer `life-1.hs` del directori de la part2, i que ja se us dona fet) conté el punt d'inici de l'aplicació i defineix el tipus `Game` per a l'estat del joc i les funcions d'inicialització, tractament dels esdeveniments i visualització de l'estat.

S'ha definit el tipus `Game` corresponent a l'estat del joc (usant sintaxis *record*) amb un sol camp corresponent al taulell del joc:

```
data Game = Game
        { gmBoard :: Board       -- last board generation
        }
```

```
    deriving (Show, Read)
```

Observeu com s'ha usat la directiva `deriving` que indica que es generin automàticament les instàncies de les classes `Show` i `Read` per al tipus `Game`.

A més les funcions:

```
gmBoard :: Game -> Board      -- Aquesta funcio es deriva de
                              -- la sintaxis record
setGmBoard :: Board -> Game -> Game
```

ens permetran, respectivament, obtenir el taulell de l'estat del joc i modificar el taulell de l'estat del joc.

Els únics esdeveniments que tractarem ara seran:

- tecla `N` polsada. Aquest esdeveniment canvia el taulell a la següent generació.
- botó del ratolí polsat. Aquest esdeveniment canvia la casella *clickada* d'estat viva a morta o viceversa.

  Per obtenir la posició de la casella (columna, fila) a partir de les coordenades del ratolí (x, y), considerarem per ara que cada casella ocupa 1x1 unitats del sistema de coordenades, i que l'origen de coordenades (centre de la vista) correspon a la posició (0, 0) del taulell.

  Així doncs, la conversió de coordenades gràfiques a posicions en el taulell consistirà bàsicament en usar la funció `round` que obté, a partir d'un valor de tipus `Double`, el seu arrodoniment (de tipus `Int`):

  ```
  round :: (RealFrac a, Integral b) => a -> b
  ```

Per a la visualització de l'estat usarem la funció `drawBoard` del mòdul `Life.Draw` (fitxer `Life/Draw.hs`) que haureu de completar.

Per això, definiu una funció

```
drawCell :: Pos -> Drawing
```

que dibuixi un rectangle sòlid de dimensions 1x1 en la coordenada corresponent a la posició indicada. Useu la funció `fromIntegral` per a la conversió de `Int` a `Double`.

Un cop definida `drawCell`, la funció `drawBoard` la podeu definir fàcilment usant les funcions `liveCells`, `foldMap` i `drawCell`.

Treball

1. Completeu el mòdul `Life.Draw` (fitxer `Life/Draw.hs`).
2. Executeu el programa `life-1.hs`.
3. Visiteu la URL
   `http://localhost:PORT/` i comprobeu que funciona correctament.

3.3 Segon pas

El segon pas consistirà en afegir un control per mostrar/ocultar una graella en la visualització del taulell.

Per això, definim el tipus `GridMode` per representar el mode de graella i un nou camp `gmGridMode` en l'estat del joc `Game` (veieu el fitxer `life-2.hs` del directori de la part2). Els valors possibles de `GridMode` són:

- `NoGrid`: Visualitza sense graella.
- `LivesGrid`: Visualitza amb una graella que inclou les cel.les vives.
- `ViewGrid`: Visualitza amb una graella que ocupa tota la vista.

Haureu d'afegir el tractament de l'event KeyDown "G" (tecla `G` polsada). Aquest esdeveniment canvia el mode de graella entre els 3 valors posibles.

També haureu d'extendre la funció de visualització per a que generi el dibuix de la graella segons el mode de l'estat actual. En mode `LivesGrid`, els extrems de la graella els podeu obtenir amb les funcions `minLiveCell` i `maxLiveCell` del mòdul `Life.Board`.

En mode `ViewGrid`, els extrems de la graella els haureu d'obtenir a partir de les coordenades dels extrems de la vista `viewWidth` i `viewHeight`.

Treball

1. Completeu el mòdul `Main` (fitxer `life-2.hs`).
2. Executeu el programa.
3. Visiteu la URL
   `http://localhost:PORT/` i comprobeu que funciona correctament.

3.4 Tercer pas

En el tercer pas introdüirem controls per poder fer *zoom* i desplaçaments en la visualització del taulell.

Per això, definim nous camps `gmZoom` i `gmShift` en l'estat del joc `Game` (veieu el fitxer `life-3.hs` del directori de la part2).

Haureu d'afegir el tractament de nous esdeveniments per canviar l'estat de `gmZoom` i `gmShift`. Useu les tecles `O`/`I` per reduir/ampliar (*zoom out*/*zoom in*) la vista del taulell.

Pot ser convenient limitar el rang dels possibles valors del camp `gmZoom`. Per exemple:

```
...
handleEvent (KeyDown "I") game = -- Zoom in
    if gmZoom game < 2.0 then setGmZoom (gmZoom game * 2.0) game
    else game
...
```

Useu les tecles de direcció (fletxes) per desplaçar el centre de la vista del taulell:

`KeyDown "ARROWUP"` Mou amunt, desplaçament del taulell cap avall.

`KeyDown "ARROWDOWN"` Mou avall, desplaçament del taulell cap amunt.

`KeyDown "ARROWRIGHT"` Mou a la dreta, desplaçament del taulell cap a l'esquerra.

`KeyDown "ARROWLEFT"` Mou a l'esquerra, desplaçament del taulell cap a la dreta.

Us serà molt convenient usar les funcions del mòdul Drawing.Vector de la biblioteca. Per exemple:

```
...
handleEvent (KeyDown "ARROWUP") game = -- Down shift
    setGmShift (gmShift game ^-^ (1.0 / gmZoom game) *^ (0, 5))
    game
...
```

En el tractament dels esdeveniments del ratolí, per obtenir la posició de la casella (columna, fila) a partir de les coordenades del ratolí (x, y), us serà molt útil definir i usar la següent funció:

```
pointToPos :: Point -> Game -> Pos
pointToPos p game =
    let (gx, gy) = (1.0 / gmZoom game) *^ p ^-^ gmShift game
    in (round gx, round gy)
```

Aquesta funció també us pot ajudar, en la visualització de l'estat, per obtenir les posicions corresponents als extrems de la vista.

També haureu de modificar la funció de visualització, aplicant l'escalat i traslació adeqüats segons els valors actuals de `gmZoom` i `gmShift`.

Treball

1. Completeu el mòdul `Main` (fitxer `life-3.hs`).
2. Executeu el programa.
3. Visiteu la URL `http://localhost:PORT/` i comprobeu que funciona correctament.

3.5 Quart pas

En el quart pas introdüirem el temps, evolució automàtica de generacions, i controls per poder canviar el mode de pausa/evolució automàtica i la velocitat de l'evolució automàtica.

Per això, definim nous camps `gmPaused`, `gmInterval` i `gmElapsedTime` en l'estat del joc `Game` (veieu el fitxer `life-4.hs` del directori de la part2).

Haureu d'afegir el tractament de nous esdeveniments per canviar els aspectes de temps de l'estat del joc:

`KeyDown " "` Canvia el mode de pausa/evolució automàtica de generacions (camp `gmPaused`).

`KeyDown "+"` Aumenta la velocitat d'evolució automàtica de generacions. Això ho farem disminuint l'interval entre generacions (camp `gmInterval`) a la meitat, per exemple.

`KeyDown "-"` Disminueix la velocitat d'evolució automàtica de generacions. Això ho farem aumentant l'interval entre generacions al doble, per exemple.

`TimePassing dt` Esdeveniment produit amb el pas del temps. L'argument `dt` és el temps que ha passat (en unitats de segon) des de l'esdeveniment `TimePassing` anterior.

Usarem el camp `gmElapsedTime` per anar acumulant el temps passat des de l'última generació automàtica.
En mode d'evolució automàtica, i en el cas de que `gmElapsedTime game + dt >= gmInterval game`, aleshores cal evolucionar a una nova generació (i restablir el camp `gmElapsedTime` a `0`). En cas contrari, senzillament acumulem el temps `dt` al camp `gmElapsedTime`.

Cal considerar que la implementació actual no permet refrescar els dibuixos a freqüències massa elevades (i sempre depenent del RTT i la qüalitat de la connexió entre el navegador i el CGI). Per tant limiteu la velocitat a un màxim de 8 per segon (`gmInterval` igual a `0.125`).

Treball

1. Completeu el mòdul `Main` (fitxer `life-4.hs`).
2. Executeu el programa.
3. Visiteu la URL
   `http://localhost:PORT/` i comprobeu que funciona correctament.

3.6 Cinquè pas

En aquest últim pas afegirem en el dibuix text que mostri l'estat d'alguns paràmetres del joc i, en mode de pausa, una ajuda de les tecles que canvien els paràmetres del joc (veieu el funcionament de la versió final).

Observeu que per aquesta part només haureu de modificar la funció de dibuix.

Treball

1. Modifiqueu la funció de dibuix del mòdul `Main` de la versió anterior, per tal que mostri l'estat i, en mode pausa, una ajuda del joc.
2. Executeu el programa.
3. Visiteu la URL i comprobeu que funciona correctament.