

Pong: simples implementação em Haskell

Luis Paulo Lima

24 de novembro de 2017

Sumário

1	Introdução	1
2	Tipos	1
3	Constantes globais	3
3.1	Constates básicas	3
3.2	Consates derivadas	3
4	Estado inicial e funções principais	4
4.1	Funções principais	5
4.1.1	Desenhar na tela	5
4.1.2	Entrada humana	5
4.1.3	Iteração do universo	6
5	Renderizando o jogo	6
6	Entrada das teclas e movimento	7
6.1	Movimentação	7
7	Pontuação	8

1 Introdução

O propósito deste trabalho é entender como funciona programação literária em Haskell e praticar algumas funcionalidades da própria linguagem. Dentre elas, usei principalmente **lens** e a mônada **Maybe**. Toda a implementação é baseada no pacote **gloss**. Todas as ações I/O são realizadas por ele, sendo que IO só foi usada durante o desenvolvimento para *debugging*, quando necessário.

```
{-# LANGUAGE TemplateHaskell #-}
```

```
module Main where
```

```
import Control.Lens
```

```
import Data.Maybe
```

```
import Graphics.Gloss
```

```
import Graphics.Gloss.Interface.IO.Game
import System.Random
```

Seguindo a ideia da programação literária, eu começo o desenvolvimento criando os tipos que serão usados e as lentes para operar em alguns deles. Depois, algumas constantes globais que serão usadas ao decorrer da implementação, e o estado inicial do jogo e o procedimento de início da partida.

2 Tipos

Os tipos básicos são usados para representar a direção de movimento dos jogadores e posições relativas à tela.

```
data Direcao = Cima | Baixo | Parado deriving (Eq, Show)
data Lado = Topo | Base | Dir | Esq deriving (Eq, Show)
```

Quanto ao universo do jogo, são usados registros (*records*) para armazenar o estado da bola e dos jogadores.

```
data Jogador = Jogador
    { _posY      :: Float          -- Posicao vertical da barra
    , _dirMov    :: Direcao        -- Direcao atual do movimento
    , _pts       :: Int            -- Pontuacao do jogador
    } deriving (Eq, Show)

data Bola = Bola
    { _veloc     :: Float          -- Magnitude da velocidade
    , _angulo    :: Float          -- Angulo do movimento da bola
    , _posXY     :: (Float,Float)  -- Posicao XY da bola
    } deriving (Eq, Show)
```

Para o jogo como um todo, além de armazenar dois jogadores mais a bola, alguns dados adicionais são necessários:

1. Onde ocorreu o último toque: para evitar o *bug* de uma bola muito lenta ficar rebatendo indefinidamente caso estiver muito próxima de um objeto.
2. De que lado ocorreu o último ponto: usado para determinar de que lado a bola começa no ponto seguinte.
3. Contador de tempo: para atrasar o recomeço quando houver ponto.

```
data Jogo = Jogo
    { _player1    :: Jogador      -- Jogador 1
    , _player2    :: Jogador      -- Jogador 2
    , _pong       :: Bola         -- Bola do jogo
    , _ultimoToque :: Lado        -- Armazena lado do ultimo toque
    , _ultimoPonto :: Lado        -- Lado onde foi feito o ultimo ponto
    , _atrasoPonto :: Float       -- Contador de tempo para atraso
    } deriving (Eq, Show)
```

Perecebe-se que cada campo do registro começa por um traço inferior (`_`). Eles uma exigência para facilitar a criação das lentes:

```
makeLenses ''Jogador
makeLenses ''Bola
makeLenses ''Jogo
```

A função `makeLenses` usa metaprogramação para criar as lentes dos registros, a qual já foi habilitada pela extensão `TemplateHaskell` no cabeçalho. Cada campo definido num registro torna-se por se só uma função de acesso aos dados armazenados:

```
_campo :: a → b
```

Quando a lente é criada, uma nova função é criada para ser usada com `over`, `set` ou `view`, do pacote `lens`:

```
campo :: Functor f => (b → f b) → a → f a
```

Por exemplo, `set` é usada para sobrescrever algum dado num registro. Se usada para alterar a bola (`_pong`) no estado geral do jogo (`Jogo`), tem-se:

```
set pont :: Bola → Jogo → Jogo
```

3 Constantes globais

Por se tratar de um jogo gráfico, definir algumas constantes globais torna o programa mais legível: em vez de passar vários argumentos para várias funções que dependem da geometria dos objetos na tela, é mais fácil apenas invocar o valor global. Há três constantes básicas, e outras cinco definidas em função dessas primeiras.

3.1 Constates básicas

As três principais são:

1. Taxa de quadros por segundo para atualização da tela (Hz).

```
fps      :: Int
fps      = 60
```

2. Tempo de atraso entre os pontos (s).

```
delayInit :: Float
delayInit  = 1
```

3. Dimensões (x, y) da janela (px).

```
tamJanela :: (Int, Int)
tamJanela  = (800, 600)
```

3.2 Consates derivadas

As demais são definidas todas elas em função do tamanho da janela, e entre si mesmas:

1. Raio da bola (px).

```
raioBola    :: Float
raioBola    = 0.02 * (fromIntegral $ snd tamJanela)
```

2. Largura do jogador (px).

```
ladoJogador :: Float
ladoJogador  = raioBola
```

3. Comprimento do jogador (px).

```
compJogador :: Float
compJogador  = 10 * ladoJogador
```

4. Passo do movimento do jogador a cada quadro (px).

```
passoJogador :: Float
passoJogador  = 0.2 * compJogador
```

5. Limites da janela a partir do centro (px): superior, inferior, esquerda e direita.

```
limJanela    :: (Float, Float, Float, Float)
limJanela    = (,,,)
              -- Fundo direito
              ((fromIntegral (fst tamJanela) ) / 2)
              -- Fundo esquerdo
              ((fromIntegral (-(fst tamJanela))) / 2)
              -- Superior e ponto na esquerda
              ((fromIntegral (snd tamJanela) ) / 2)
              -- Inferior e ponto na direita
              ((fromIntegral (-(snd tamJanela))) / 2)
```

4 Estado inicial e funções principais

Já tendo os tipos e constantes definidas, é possível definir o estado inicial do jogo: a bola, os jogadores e o universo por completo.

```

jogador0 = Jogador { _posY    = 0
                    , _dirMov = Parado
                    , _pts     = 0 }

bola0    = Bola    { _angulo = 45
                    , _posXY  = (0,0)
                    , _veloc  = 7 }

jogo0    = Jogo    { _player1   = jogador0
                    , _player2   = jogador0
                    , _pong      = bola0
                    , _ultimoToque = Topo
                    , _ultimoPonto = Esq
                    , _atrasoPonto = delayInit }

```

O jogo é baseado na função `playIO` do `gloss`. Ela exige a descrição da janela, cor de fundo, taxa atualização da tela, uma única constante representando o universo do jogo, uma função para renderizar o jogo, outra para lidar com eventos do jogador, e outra para adiatnar o estado do jogo segundo a passagem do tempo.

```

jogarIO j = playIO
  (InWindow "Pong" tamJanela (40,40)) -- Janela do jogo
  black                                -- Cor de fundo
  fps                                  -- Taxa FPS
  j                                    -- Entrada (tipo Jogo)
  renderizar                           -- Converto o mundo para Picture
  eventoTecla                           -- Lidar com eventos IO
  passoGeral                            -- Lida com a passagem do tempo

```

Na função principal `main`, a inicialização do jogo começa por definir um ângulo aleatório para o movimento da bola. Percebe-se que em `bola0`, a posição inicial é `(0,0)`, que corresponde ao centro da tela. Diferente do Pong original, ou até mesmo do Ping Pong anterior a ele, nesta implementação, a bola sempre partirá do centro da tela, num ângulo aleatório, começando pela direita.

```

main :: IO ()
main = let f x = if odd x then x else (-x)
      in do a <- randomRIO (5, 45) :: IO Int
            s <- newStdGen
            jogarIO $ set (pong · angulo) (fromIntegral · f $ a)
                      $ jogo0

```

4.1 Funções principais

Por exigência de `playIO`, `renderizar`, `eventoTecla` e `passoGeral` devem ser tais que:

```

j          :: a
renderizar :: a → IO Picture
eventoTecla :: Event → a → IO a
passoGeral  :: Float → a → IO a

```

E já definí-las ajuda no processo de desenvolvimento do programa: começando pelas funções de mais alta ordem e partindo para as elementares, na medida do necessário.

4.1.1 Desenhar na tela

A função `renderizar` usa `_player1` e `_player2` de `Jogo` para desenhá-los na tela, mais `_pong` para desenhar a bola.

```
renderizar :: Jogo → IO Picture
renderizar jogo =
  let js = zip [Esq,Dir] [_player1 jogo, _player2 jogo]
      b = _pong jogo
  in return · pictures $ (renderBola b) : (map renderJogador js)
```

Acabamos de definir, portanto, que `_player1` fica do lado esquerdo da tela, e `_player2` fica do lado direito.

4.1.2 Entrada humana

`gloss` define o tipo `Event` para encapsular todas as possíveis ações do usuário. No caso deste jogo, fica definido que:

1. As teclas W e S movimentam o jogador à esquerda da tela (`_player1`).
2. As teclas ↑ e ↓ movimentam o jogador à direita da tela (`_player2`).
3. Qualquer outra entrada (tecla, cliques, movimento do mouse) não realizam nenhuma ação.

Nesse sentido, fica clara a conveniência de se usar `Maybe` para tratar os eventos de tecla. Como definimos que `Jogador` carrega sua direção do movimento, `_dirMov`, basta que este valor seja alterado segundo a entrada do jogador.

```
eventoTecla :: Event → Jogo → IO Jogo
eventoTecla evento jogo =
  let tecla = lerTecla evento
  in if isNothing tecla
      then return jogo
      else let (p,d) = fromJust tecla
            in do return $ set (p · dirMov) d jogo
```

Daqui também já fica claro que a função elementar `lerTecla` deverá ter como saída uma dupla contendo a lente do jogador (`player1` ou `player2`), e a traduzir a tecla pressionada num valor do tipo `Direcao`.

4.1.3 Iteração do universo

A última função principal é responsável por atualizar o jogo na seguinte sequência:

1. Conta o tempo de atraso de cada ponto até zerar;
2. Detecta e age se a bola está em posição de colisão;

3. Move os jogadores para cima ou para baixo;
4. Move a bola;
5. Detecta se há ponto (bola fora).

```
passoGeral :: Float → Jogo → IO Jogo
passoGeral dt jogo = if _atrasoPonto jogo > 0
    then return $ over atrasoPonto (+(-dt)) jogo
    else (detectColisao jogo)
        >>= (moverJogadores passoJogador)
        >>= moverBola
        >>= bolaFora
```

5 Renderizando o jogo

Para cada jogador, usa-se `renderJogador`, que admite o uma dupla contendo um lado da tela e o próprio jogador, o qual carrega consigo sua posição vertical. O desenho é um retângulo branco, cujo tamanho segue as constantes globais.

```
renderJogador :: (Lado, Jogador) → Picture
renderJogador (l,j) =
    let p = _posY j
        b = color white $ rectangleSolid ladoJogador compJogador
        x = (/2) · fromIntegral $ snd tamJanela
    in case l of Dir → translate (x - (ladoJogador/2)) p b
               Esq → translate (-x + (ladoJogador/2)) p b
```

Um círculo branco representa a bola. `renderBola` admite apenas `Bola`, que já carrega em si mesma sua posição (x, y) na tela.

```
renderBola :: Bola → Picture
renderBola b =
    let (x,y) = _posXY b
    in color white $ translate x y
        $ circleSolid raioBola
```

6 Entrada das teclas e movimento

Seguindo a documentação de `gloss`, e de acordo com o que definimos na Seção 4.1.2 pode-se escrever a função que irá receber os eventos enviados à `eventoTecla`, que vêm das funções internas de `playIO`.

```
lerTecla (EventKey (Char c) e _)
    | c == 'w' ∧ e == Down = Just (player1, Cima)
    | c == 'w' ∧ e == Up   = Just (player1, Parado)
    | c == 's' ∧ e == Down = Just (player1, Baixo)
    | c == 's' ∧ e == Up   = Just (player1, Parado)
lerTecla (EventKey (SpecialKey s) e _)
```

```

| s ≡ KeyUp    ∧ e ≡ Down = Just (player2, Cima)
| s ≡ KeyUp    ∧ e ≡ Up   = Just (player2, Parado)
| s ≡ KeyDown  ∧ e ≡ Down = Just (player2, Baixo)
| s ≡ KeyDown  ∧ e ≡ Up   = Just (player2, Parado)
lerTecla _ = Nothing

```

Nota-se que não há anotação de tipo para `lerTecla`. Isso acontece porque ela retorna `player1` e `player2` e, sendo assim, seu tipo deveria ser

```
lerTecla :: Event → Maybe (Lens' Jogo Jogador, Direcao)
```

Porém, `Lens'` dentro de `Maybe` exige polimorfismo impredicativo, o que ainda não é suportado pelo GHC, mesmo se usássemos a extensão `RankedNTypes`. Portanto, convém deixar que o tipo seja automaticamente inferido pelo compilador. Isso resulta em:

```

lerTecla :: ∀ {f :: * → *}
          · Functor f
          => Event
          → Maybe ((Jogador → f Jogador) → Jogo → f Jogo, Direcao)

```

Por este tipo não ser muito “legível”, convém não ser anotado.

6.1 Movimentação

O movimento dos desenhos dos jogadores na tela é realizado usando-se o valor `_dirMov` presente no registro do `Jogador`, dentro de `_palyer1` e `_player2` do registro `Jogo`. Ele é restringido por `limJanela` e por `compJogador`. Apesar de o passo do movimento ser também uma constante global, ele é passado para a função como um argumento para facilitar *debugging*.

```

moverJogadores :: Float → Jogo → IO Jogo
moverJogadores h jogo = return $ over player1 (moverJ h)
                        $ over player2 (moverJ h) jogo
  where moverJ h j = case (_dirMov j) of
    Cima → if (_posY j) + compJogador / 2 ≥ view _3 limJanela
           then set posY
                ((view _3 limJanela) - compJogador/2) j
           else over posY (+h) j
    Baixo → if (_posY j) - compJogador / 2 ≤ view _4 limJanela
            then set posY
                 ((view _4 limJanela) + compJogador/2) j
            else over posY (+(-h)) j
    _     → j

```

Quanto à bola, os dados necessários para sua movimentação também estão contidos nela mesma. Não há ação alguma dos jogadores que a faça mudar de direção ou sentido: isto deve acontecer apenas em caso de colisão. Portanto, basta multiplicar `_veloc` pela variação do tempo, que vem de `gloss` e considerar o ângulo do movimento contido em `_angulo`.


```

moverBola :: Jogo → IO Jogo
moverBola jogo = return $ over (pong · posXY) f jogo
  where v = view (pong · veloc) jogo
        a = view (pong · angulo) jogo
        f (x,y) = ( x + v * (cos (a * π / 180) )
                    , y + v * (sin (a * π / 180) ) )

```

7 Pontuação

A condição para que haja ponto é que a bola toque o limite de ponto definido nas constantes globais, na esquerda ou na direita. Se a bola sair pelo lado l , o procedimento seguido é o seguinte:

1. Definir que ultimo ponto foi no lado l ;
2. Definir um ângulo alet'ório para lançamento da bola:
 - (a) Se $l == \text{Esq}$, entre -45° e $+45^\circ$;
 - (b) Se $l == \text{Dir}$, entre 135° e 225° .
3. Definir que o último toque foi numa posição “nula” (Topo);
4. Posicionar ambos os jogadores na posição central;
5. Incrementar a pontuação do jogador do lado oposto:
 - (a) Se $l == \text{Esq}$, incrementa em $_player2$;
 - (b) Se $l == \text{Dir}$, incrementa em $_player1$.
6. Resetar o contador de atraso para delayInit ;
7. Posicionar a bola na origem da tela, $(0,0)$.

```

bolaFora :: Jogo → IO Jogo
bolaFora jogo =
  let (x,y) = view (pong · posXY) jogo
      f d x = if d == Dir
                then if odd x then x           else (-x)
                else if odd x then (180 - x)    else (180 + x)
      pontoReset l a j = set (pong · posXY) (0,0)
                        $ set atrasoPonto delayInit
                        $ over (l · pts) (+1)
                        $ set (player1 · posY) 0
                        $ set (player2 · posY) 0
                        $ set ultimoToque Topo
                        $ (λj' → if _ultimoPonto j' == Dir
                                then set (pong · angulo)
                                    (fromIntegral · (f Dir) $ a)
                                j')
                        else set (pong · angulo)

```

```

                                (fromIntegral · (f Esq) $ a)
                                j' )
    $ (λj' → if x > 0
        then set ultimoPonto Dir j'
        else set ultimoPonto Esq j' ) j
in do s ← randomRIO (5, 45) :: IO Int
    if x > view _3 limJanela
    then return $ pontoReset player1 s jogo
    else if x < view _4 limJanela
        then return $ pontoReset player2 s jogo
        else return $ jogo

detectColisao :: Jogo → IO Jogo
detectColisao jogo
    | (y - raioBola) < view (_4) limJanela
    = atualiza Base jogo
    | (y + raioBola) > view (_3) limJanela
    = atualiza Topo jogo
    | (x + raioBola + ladoJogador) > view (_3) limJanela ∧ c
    = atualiza Dir jogo
    | (x - raioBola - ladoJogador) < view (_4) limJanela ∧ c
    = atualiza Esq jogo
    | otherwise = return $ jogo
    where c = contatoJB jogo
          (x,y) = view (pong · posXY) jogo
          atualiza d j = if d == _ultimoToque j then return j
                        else return
                            $ set ultimoToque d
                            $ over (pong · angulo) (refletir d) j

-- Contato entre jogador e bola
contatoJB :: Jogo → Bool
contatoJB jogo =
    if ((xB < 0) ∧ (abs (yB - yJ1)) < (compJogador/2)) ∨
       ((xB > 0) ∧ (abs (yB - yJ2)) < (compJogador/2))
    then True else False
    where yJ1 = view (player1 · posY) jogo
          yJ2 = view (player2 · posY) jogo
          lEsq = view _4 limJanela
          lDir = view _3 limJanela
          (xB,yB) = view (pong · posXY) jogo

-- Mudana de ngulo de uma bola devido a coliso
refletir :: Lado → Float → Float
refletir l a
    | l == Topo = worker a 270
    | l == Base = worker a 90
    | l == Esq = worker a 0
    | l == Dir = worker a 180

```

where worker $t_n = (n + 90) - (t - (n + 90))$