Haskell exercise: Games

Department of Mathematics and Computer Science
University of Southern Denmark

October 6, 2017

In this Exercise session, you are going to implement a simple game and a few functions related to games.

In the file `gameslab.hs` we have provided the following data type *Game*.

```
type Player = Bool
data Game s m = Game {
  startState :: s,
  showGame   :: s -> String,
  move       :: Player -> s -> m -> (Player, s),
  moves      :: Player -> s -> [m],
  value      :: Player -> s -> Double
  }
```

1 The Add Game

The Add Game is a two-player game with parameters (n, k).

Two players start from state $s = 0$ and alternatively add a number $1 \leqslant m \leqslant k$ to the sum s. The player who reaches n wins.

Here is an example game with $n = 23$ and $k = 10$:

- Starting state $s = 0$.
- Player *True*'s move: 1 (s is now $= 1$)
- Player *False*'s move: 2 (s is now $= 3$)
- Player *True*'s move: 9 (s is now $= 12$)
- Player *False*'s move: 3 (s is now $= 15$)

- Player *True*'s move: 8 (s is now $= 23$)
- Player *True* won!

In the `gameslab.hs` file, we have provided the following implementation stub:

$$
\begin{array}{l}
\textbf{data}\ \mathit{AddGame} = \mathit{AddGame}\ \mathit{Int}\ \mathit{Int} \\[4pt]
\mathit{startStateImpl} :: \mathit{AddGame} \to \mathit{Int} \\
\mathit{startStateImpl}\ \_ = \bot \\[4pt]
\mathit{moveImpl} :: \mathit{AddGame} \to \mathit{Player} \to \mathit{Int} \to \mathit{Int} \to (\mathit{Player}, \mathit{Int}) \\
\mathit{moveImpl}\ (\mathit{AddGame}\ n\ k)\ p\ s\ m = \bot \\[4pt]
\mathit{movesImpl} :: \mathit{AddGame} \to \mathit{Player} \to \mathit{Int} \to [\mathit{Int}] \\
\mathit{movesImpl}\ (\mathit{AddGame}\ n\ k)\ p\ s = \bot \\[4pt]
\mathit{valueImpl} :: \mathit{AddGame} \to \mathit{Player} \to \mathit{Int} \to \mathit{Double} \\
\mathit{valueImpl}\ (\mathit{AddGame}\ n\ k)\ p\ s = \bot \\[4pt]
\mathit{addGame} :: \mathit{AddGame} \to \mathit{Game}\ \mathit{Int}\ \mathit{Int} \\
\mathit{addGame}\ g = \mathit{Game}\ \{ \\
\quad \mathit{startState} = \mathit{startStateImpl}\ g, \\
\quad \mathit{showGame} = \mathit{show}, \\
\quad \mathit{move} \qquad\;\; = \mathit{moveImpl}\ g, \\
\quad \mathit{moves} \qquad\; = \mathit{movesImpl}\ g, \\
\quad \mathit{value} \qquad\;\, = \mathit{valueImpl}\ g \\
\quad \}
\end{array}
$$

You should define appropriate function bodies instead of the $\bot$ symbols.

1. *startStateImpl* is the starting state of the game, i.e. 0.
2. The *valueImpl* function returns 0 if further moves are possible, 1 if Player *True* has won, and -1 if player *False* has won.
3. The *movesImpl* function returns the list of numbers from 1 to k which can be added to the game state s without exceeding n.
4. The *moveImpl* function adds a number to the current game state s.

2 Game paths

You are now going to define two functions which works for *any* game, which provides an implementation of the *Game* type.

$$leftmostGame :: Game\ s\ m \to Player \to s \to [s]$$
$$rightmostGame :: Game\ s\ m \to Player \to s \to [s]$$

leftmostGame g p s will output the game states visited if starting in state s with player p, and each player picks the first move available in his list of moves.

rightmostGame g p s will output the game states visited if starting in state s with player p, and each player picks the last move available in his list of moves.

For example, for the Adding Games defined above, the correct outputs should be:

$> leftmostGame\ (addGame\ (AddGame\ 10\ 3))\ True\ 0$
$[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$
$> rightmostGame\ (addGame\ (AddGame\ 10\ 3))\ True\ 0$
$[0, 3, 6, 9, 10]$

3 The one-dimensional 2048-game

You might have heard of the game 2048 (`http://gabrielecirulli.github.io/2048/`). Check out the website, and make a few moves with the arrow keys, to get an idea of how the game works. This exercise is about implementing a simple variant of this game, which is only in 1 dimension.

We introduce the data type **data** *Game1D* = *Game1D Int* to describe a game.

The state of the game *Game1D* n is a list of integers [*Int*] of length n.

The starting state of the game is the list of all zeroes of length n.

The Computer is player *True* and the Human is player *False*.

The Computer's move is to put a value of either 2 or 4 at an empty position (indexed from 0 to $n-1$) in the list.

The Human's move is a direction, either left or right.

- If the Human chose Left, the following happens to the list representing the game state:
  - First, all 0's are removed. Example: [2,0,2,2] becomes [2,2,2].
  - All consecutive numbers which are equal are merged by adding them, from left to right. Example: [2,2,2] becomes [4,2]. [2,2,2,2] becomes [4,4]. [2,2,4] becomes [4,4].

- The state list is padded with zeros at the end such that it again is of length n. Example: [2,2] becomes [2,2,0,0] for $n = 4$.

- The Human cannot choose left, if the move operation described above leaves the list unchanged. Example: For state [2,4,0,0], left is not available.

- If the Human chose Right: Can be defined using the left operation described above: If the left operation is given by the function $\mathit{leftOp} :: [\mathit{Int}] \rightarrow [\mathit{Int}]$, the operation of going right, is given by $\mathit{reverse} \circ \mathit{leftOp} \circ \mathit{reverse}$.

- The Human cannot choose right, if the move operation described above leaves the list unchanged. Example: For state [0,0,2,4], right is not available.

Below is written the types to describe a move, and below that, you are given an instance stub in the `Instances.hs` file, where you should replace the $\perp$-bodies.

```haskell
data Direction = L | R deriving Show
type Position = Int
type Value = Int
data Move1D = Computer Position Value | Human Direction deriving Show

game1d :: Game1D -> Game [Int] Int

game1d g = Game {
  startState = startStateImpl' g,
  showGame   = showGameImpl' g,
  move       = moveImpl' g,
  moves      = movesImpl' g,
  value      = valueImpl' g
  }
```

The value function is *maxBound* if the Human has no available moves.

Otherwise, it is given by the following formula: $-a - 3 * b$ where a is the number of 0's in the current state, and b is the maximum value in the current state (the Human, player *False*, wants to minimize this value when playing. This is just a heuristic function.)

Sample game for *Game1D* 3:

- Start state $[0, 0, 0]$

- Player *True* chooses move *Computer* 0 2 (state is now $[2, 0, 0]$)

- Player *False* chooses move *Human R* (state is now $[0, 0, 2]$)
- Player *True* chooses move *Computer* 0 2 (state is now $[2, 0, 2]$)
- Player *False* chooses move *Human L* (state is now $[4, 0, 0]$)
- Player *True* chooses move *Computer* 1 2 (state is now $[4, 2, 0]$)
- Player *False* chooses move *Human R* (state is now $[0, 4, 2]$)
- Player *True* chooses move *Computer* 0 2 (state is now $[2, 4, 2]$)
- No moves available!