Programming Languages (Project 1)

Jeff Gyldenbrand (jegyl16)

November 13, 2017

Abstract

The goal of this project is . . . to eat cake

Contents

1 The Kalaha game with parameters (n, m)

```haskell
module Kalaha where

import Data.List

type PitCount   = Int
type StoneCount = Int
data Kalaha     = Kalaha PitCount StoneCount deriving (Show, Read, Eq)

type KPos       = Int
type KState     = [Int]
type Player     = Bool
```

1.1 The function `startStateImpl`

We are giving the declaretion `startStateImpl` with a 'Kalaha'-game: Kalaha, which takes a pit-count and stone-count as parameters and returns the initial state of the game in a list: KState.

By replicating the pit-count and stone-count, and then append a zero, we get half the list, so we simply do this twice to return the complete state.

```haskell
startStateImpl :: Kalaha -> KState
startStateImpl (Kalaha pitC stoneC) = replicate' ++ replicate'
  where
    replicate' = replicate pitC stoneC ++ [0]
```

1.2 The function `movesImpl`

In `movesImpl` which, besides the 'Kalaha'-game parameters, now take as parameters a player; False or True, and a state for the game. Then returns the pits of given player's which has a positive number of elements.

This is done by making two guards: one for each player. If it is player False we simply use the 'findIndices' function which, in our case, returns all indices greater than zero. This we can do because we split the game state into a tuble, and define player False to be the first element in the tuble. We use init to exclude player False's own kalaha pit.

To find same indices only for player True, we define our own findIndices' functions, that recursively searches the other element in the before mentioned tuble.

```haskell
movesImpl :: Kalaha -> Player -> KState -> [KPos]
movesImpl (Kalaha pitC stoneC) p gState
  | p == False = findIndices (>0) (pFalse)
  | p == True = findIndices' (pitC+1) (pTrue)
    where
      pFalse = init(fst(splitAt (pitC+1) gState))
      pTrue = init(snd(splitAt (pitC+1) gState))
      findIndices' _ [] = []
      findIndices' pitC (x:gState)
        | (x>0) = pitC : findIndices' (pitC+1) gState
        | otherwise = findIndices' (pitC+1) gState
```

1.3 The function `valueImpl`

In `valueImpl` which, besides the 'Kalaha'-game parameters, now takes as parameters the game state. Then returns True's kalaha pit subtracted from False's kalaha pit as a double.

As in `movesImpl` this is done by splitting the game state into two elements in a tuble. Then assigning True's kalaha pit to the last element in the first element of the tuble, and False's kalaha pit to the last element in the second element of the tuble. Then we simply subtract the two kalaha pits.

```
valueImpl :: Kalaha -> KState -> Double
valueImpl (Kalaha pitC stoneC) gameState = fromIntegral (pitTrue - pitFalse)
  where
    pitFalse = last(fst(splitAt(pitC+1) gameState))
    pitTrue = last(snd(splitAt(pitC+1) gameState))
```

1.4 The function `moveImpl`

In `moveImpl` we take all the same parameters as in `movesImpl` only now we want to return the the logic of a move, meaning the next player and new state, in a tuble.

To accomplish this, several help functions is developed: `letsMove`, `initMove`, `incrementMove`, `emptyPit`, `emptySpecificPit`, `lastMove`, `endCheck`, `sweapBoard`.

All these help functions are needed in order to define a ruleset of the move, and will be explained in more details in the beginning of each. As for the main function it will read the given parameters, and call our first help- function `letsMove`.

```
moveImpl :: Kalaha -> Player -> KState -> KPos -> (Player,KState)
moveImpl (Kalaha pitC stoneC) p gState pitIndex = nextTurn
 where
   nextTurn = letsMove (Kalaha pitC stoneC) p newGameState (pitIndex+1) pitVal
   pitVal = gState!!pitIndex
   newGameState = initMove pitIndex gState
```

1.5 The function `letsMove`

`letsMove` is where the primary action is happening. As parameters it takes a kalaha game, a player, a state, the current index and its value. With guards we check for some condition, that will determine the next legal move:

1. When last move in game is made, we check for the other move-rules
2. To prevent a move to reach past the last index in the list, we jump back at the beginning, and drop a stone.
3. If player false lands in player trues pit, we skip it, and land at player falses start pit, and drop a stone.
4. if player true lands in player falses pit, we skip it, and land at player trues start pit, and drop a stone.
5. if none of above rules are violated, we move to next pit and drop a stone.

```
letsMove (Kalaha pitC stoneC) p gState pitIndex pitVal
-- Guard 1:
 | (pitVal == 0) = endCheck (Kalaha pitC stoneC) p (pitIndex-1) gState
-- Guard 2:
 | (pitVal > 0) && (pitIndex > pitC*2+1) =
   letsMove (Kalaha pitC stoneC) p beyond 1 (pitVal-1)
```

```
-- Guard 3:
 | (p == False) && (pitIndex == 2*pitC+1) =
   letsMove (Kalaha pitC stoneC) p skipTrue 1 (pitVal-1)
-- Guard 4:
 | (p == True) && (pitIndex == pitC) =
   letsMove (Kalaha pitC stoneC) p skipFalse (pitIndex+2) (pitVal-1)
-- Guard 5:
 | otherwise = letsMove (Kalaha pitC stoneC) p nextMove (pitIndex+1) (pitVal-1)
 where
  nextMove = modify pitIndex gState 1
  beyond = modify 0 gState 1
  skipTrue = modify 0 gState 1
  skipFalse = modify (pitIndex + 1) gState 1
```

1.6 The help-function `initMove`

This function assist the main-function `moveImpl`, and ...

```
initMove pitIndex gameState = newGameState
 where
   pitsFalse = init(fst(splitAt(pitIndex + 1) gameState))
   pitsTrue = snd(splitAt(pitIndex + 1) gameState)
   newGameState = (pitsFalse ++ 0 : pitsTrue)
```

1.7 The help-function `incrementMove`

```
modify pitIndex gameState incrementValue = newGameState
 where
   pitValue = gameState!!pitIndex
   pitsFalse = init(fst(splitAt(pitIndex + 1) gameState))
   pitsTrue = snd(splitAt (pitIndex + 1) gameState)
   newGameState = (pitsFalse ++ (pitValue + incrementValue) : pitsTrue)
```

1.8 The help-function `emptyPit`

```
emptyPit (Kalaha pitCount stoneCount) player q s
 | player == False = emptyF'
 | player == True = emptyT'
 where
   -- Modsat pit + 1
   op = (s!!(q+((2*pitCount)-(q*2)))) + 1
   -- tmmer index for tomt slut pit
   k2 = initMove q s
   -- tmmer modsat pit
   k3 = initMove (q+((2*pitCount)-(q*2))) k2
   -- false
   emptyF' = modify pitCount k3 op
   -- true
   emptyT' = modify ((pitCount*2)+1) k3 op
```

1.9 The help-function `emptySpecificPit`

```haskell
emptySpecificPit (Kalaha n m) p o s
 | p == False = allEmptyFalse'
 | otherwise = allEmptyTrue'
 where
  val = s!!o
  k = initMove o s
  allEmptyFalse' = modify n k val
  allEmptyTrue' = modify (n*2+1) k val
```

The help-function `lastMove` 1. if player false lands in an empty pit 2. if player true lands in an empty pit 3. if player false lands in player trues kalaha 4. if player true lands in player falses kalaha 5. nothing happens, and returns state and its next players turn —-

```haskell
lastMove (Kalaha pitC stoneC) p pitIndex gState
 | (p == False) && (gState!!pitIndex == 1) && (pitIndex < pitC) = lastEF'
 | (p == True) && (gState!!pitIndex == 1) && (pitIndex > pitC) && (pitIndex < ((pitC*2)+1)) = lastET'
 | (p == False) && (pitIndex == pitC) = lastKF'
 | (p == True) && (pitIndex == pitC*2+1) = lastKT'
 | otherwise = (not p, gState)
  where
  bo = (emptyPit (Kalaha pitC stoneC) False pitIndex gState)
  biver = (emptyPit (Kalaha pitC stoneC) True pitIndex gState)
  lastEF' = (True, bo)
  lastET' = (False, biver)
  lastKF' = (False, gState)
  lastKT' = (True, gState)
```

1.10 The help-function `endCheck`

```haskell
endCheck (Kalaha pitCount stoneCount) player pitIndex gameState
-- if all player False's pits are zero, and we are player True -> True collects the rest
 | (findIndex (>0) (fst(splitAt pitCount gState)) == Nothing) = (swap, tCollect)
-- if all player True's pits are zero, and we are player False -> False collects the rest
 | (findIndex (>0) (init(snd(splitAt (pitCount+1) gState))) == Nothing) = (swap, fCollect)
 | otherwise = lastMove (Kalaha pitCount stoneCount) player pitIndex gameState
  where
   swap = not player
   gState = snd(lastMove (Kalaha pitCount stoneCount) player pitIndex gameState)
-- creates a list of the indexes of the remaining stones still in play
   listOfindexes = findIndices (>0) gState
-- sweaps the board with the remaining stones
   tCollect = sweapBoard (Kalaha pitCount stoneCount) True gState listOfindexes ((length listOfindexes) -1)
   fCollect = sweapBoard (Kalaha pitCount stoneCount) False gState listOfindexes ((length listOfindexes) -1)
```

1.11 The help-function `emptyAll`

```haskell
sweapBoard (Kalaha pitCount stoneCount) player gameState listOfindexes indexOfPit
-- to prevent negative index
 | (indexOfPit<0) = gameState
```

```
-- recursively extract player Falses pits and update gameState
 | (extractValue == pitCount) = sweapBoard (Kalaha pitCount stoneCount) player
   gameState listOfindexes (indexOfPit-1)
-- recursively extract player Trues pits and update gameState
 | (extractValue == pitCount*2+1) = sweapBoard (Kalaha pitCount stoneCount) player
   gameState listOfindexes (indexOfPit-1)
--
 | otherwise = sweapBoard (Kalaha pitCount stoneCount) player k listOfindexes (
   indexOfPit-1)
 where
    extractValue = listOfindexes!!indexOfPit
    k = emptySpecificPit (Kalaha pitCount stoneCount) player extractValue gameState
```

1.12 The function **showGameImpl**

```
showGameImpl :: Kalaha -> KState -> String
showGameImpl g@(Kalaha pitCount stoneCount) gameState =
  unlines $ map unwords [line1, line2, line3]
  where
    maxLen = length $ show $ 2*pitCount*stoneCount
    empty = replicate maxLen ' '

    gameState' = map (pad maxLen) $ map show gameState
    (line1, line3) = (empty : (reverse $ part(pitCount+1, 2*pitCount+1) gameState'),
   empty : (part(0,pitCount) gameState'))
    line2 = last gameState' : (replicate pitCount empty ++ [gameState'!!pitCount])

pad :: Int -> String -> String
pad pitCount s = replicate (pitCount - length s) ' ' ++ s

part :: (Int, Int) -> [a] -> [a]
part (x,y) l = drop x $ take y l
```

2 Trees

```
data Tree m v  = Node v [(m,Tree m v)] deriving (Eq, Show)
```

2.1 Test tree hjelpefunktion

```
testTree :: Tree Int Int
testTree = Node 3 [(0, Node 4[(0, Node 5 []),(1, Node 6 []), (2, Node 7 [])]), (1, Node
    9[(0, Node 10[])])]
```

2.2 The function **takeTree**

```
takeTree :: Int -> Tree m v -> Tree m v
takeTree n (Node v list)
 | (n==0) = (Node v [])
 | otherwise = (Node v (map theTree list))
  where
   theTree (m,t) = (m, takeTree (n-1) t)
```

3 The Minimax algorithm

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

kalahaGame :: Kalaha -> Game KState KPos
kalahaGame k = Game {
    startState = startStateImpl k,
    showGame   = showGameImpl k,
    move       = moveImpl k,
    moves      = movesImpl k,
    value      = const (valueImpl k)}

startTree :: Game s m -> Player -> Tree m (Player,Double)
startTree g p = tree g (p, startState g)
```

3.1 The function **tree**

```haskell
tree :: Game s m -> (Player, s) -> Tree m (Player, Double)
tree game (player, state) = Node (player, treeValue) treeMove
    where
    treeValue = value game player state
    treeMoves = moves game player state
    treeMove = [(m, tree game (move game player state m)) | m <- treeMoves]
```

3.2 The function **minimax**

```haskell
minimax   :: Tree m (Player, Double) -> (Maybe m, Double)
minimax (Node (_,treeValue) []) =  (Nothing, treeValue)
minimax (Node (p, v) ch)
  | p == True = maximumSnd [ (Just path, snd $ minimax subtree) | (path, subtree) <- ch ]
  | otherwise = minimumSnd [ (Just path, snd $ minimax subtree) | (path, subtree) <- ch ]

maxSnd :: (a,Double) -> (a, Double) -> (a, Double)
maxSnd a@(_,v1) b@(_,v2) | v1 >= v2 = a| otherwise = b

minSnd :: (a, Double) -> (a, Double) -> (a, Double)
minSnd a@(_,v1) b@(_,v2) | v1 < v2 = a| otherwise = b

maximumSnd :: [(a, Double)] -> (a, Double)
maximumSnd [] = error "undefined for empty list"
maximumSnd (x:xs) = foldl maxSnd x xs

minimumSnd :: [(a, Double)] -> (a, Double)
minimumSnd [] = error "undefined for empty list"
minimumSnd (x:xs) = foldl minSnd x xs
```

3.3 The function **minimaxAlphaBeta**

```
type AlphaBeta = (Double,Double)

minimaxAlphaBeta :: AlphaBeta -> Tree m (Player, Double) -> (Maybe m, Double)
minimaxAlphaBeta = undefined
```

4 Testing and sample executions