Programming Languages (Project 1)

Jeff Gyldenbrand (jegyl16)

November 17, 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`

This function is giving with a 'Kalaha'-game: Kalaha, which takes a pit-count and stone-count as parameters.

By replicating the pit-count and stone-count, and then append a zero, we get half the list, so we simply add the list to itself to return the initial state of the Kalaha-game.

```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 which has a positive count of stones of a given player.

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'** function, 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.

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

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 the new state, in a tuble.

To accomplish this, several help functions is developed: **letsMove**, **pickUpStones**, **incVal**, **emptyPit**, **emptySpecificPit**, **lastMove**, **endCheck**, and **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 will call the function **pickUpStones** to pick up the stones at giving index, before making the first move called by the function **letsMove**

```haskell
moveImpl :: Kalaha -> Player -> KState -> KPos -> (Player,KState)
moveImpl (Kalaha pitC stoneC) p gState pitIndex = nextTurn
 where
   nextTurn = letsMove (Kalaha pitC stoneC) p updatedState (pitIndex+1) pitVal
   pitVal = gState!!pitIndex
   updatedState = pickUpStones 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 newGameState, the current index and its value. With guards we check for some condition, that will determine the next legal move:

1. if we have zero stones left in the hand, we check for the other rules. This is explained in more detailts in the help-function **endCheck**.
2. As long as we have stones in the hand we move.. and to prevent a move to reach past the last index in the list, we jump back at the beginning (index zero), and drop a stone.
3. If player false lands in player trues kalaha, we skip it, and land at player falses start pit, and drop a stone.
4. if player true lands in player falses kalaha, 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.

```haskell
letsMove (Kalaha pitC stoneC) p gState pIndex stones
 | (stones == 0) = emptyCheck (Kalaha pitC stoneC) p (pIndex-1) gState     -- 1
 | (stones > 0) && (pIndex > pitC*2+1) =                                   -- 2
   letsMove (Kalaha pitC stoneC) p beyond 1 (stones-1)
 | (p == False) && (pIndex == 2*pitC+1) =                                  -- 3
   letsMove (Kalaha pitC stoneC) p skipTrue 1 (stones-1)
 | (p == True) && (pIndex == pitC) =                                       -- 4
   letsMove (Kalaha pitC stoneC) p skipFalse (pIndex+2) (stones-1)
 | otherwise = letsMove (Kalaha pitC stoneC) p nextMove (pIndex+1) (stones-1)--5
 where
  nextMove = incVal pIndex gState 1
  beyond = incVal 0 gState 1
  skipTrue = incVal 0 gState 1
  skipFalse = incVal (pIndex + 1) gState 1
```

1.6 The help-function `pickUpStones`

We start a move by picking up all the stones from a chosen pit. We look at the current game state and changes the giving index's value to zero.

We can't just change it directly, so with a little work-a-round by splitting the game state into a tuble, appending a zero (which is the replacement for the actual value) and creating a new list, we get the new game state which is returned to

```haskell
pickUpStones pitIndex gState = newgState
 where
   removeIndexValue = init(fst(splitAt(pitIndex + 1) gState))
   restOfList = snd(splitAt(pitIndex + 1) gState)
   newgState = (removeIndexValue ++ 0 : restOfList)
```

1.7 The help-function `incVal`

This function takes as argument a pit index, game state and a value n. When ever this function is invoked, the value at giving index is incremented by n, and returns the new game state to the function it is invoked by.

```haskell
incVal pitIndex gState n = updateState
 where
   pitValue = gState!!pitIndex
   pitsFalse = init(fst(splitAt(pitIndex + 1) gState))
   pitsTrue = snd(splitAt (pitIndex + 1) gState)
   updateState = (pitsFalse ++ (pitValue + n) : pitsTrue)
```

1.8 The help-function `stealOpposite`

In the case where a player drops hes last stone in one of hes own empty pits, he will steal all the stones from the opposite pit (from the opponent) plus one (that last stone he dropped in hes own pit). All these stones will go into the players kalaha.

So based on whether we are player False or True, we invoke function **stealFromTrue** or **stealFromFalse** which invokes the functions **incVal** and **pickUpStones** which updates the state of the game with help from:

1. **totalStones** which adds the opposit pits stones with one (hence stealing the stones from the opponent plus our last dropped stone)

2. **emptyLast** which make sure to empty the pit, where we dropped the last stone.
3. **emptyOpposite** which empty the opposite pit for stones.

```
stealOpposite (Kalaha pitC _) player index gState
 | player == False = stealFromTrue
 | player == True = stealFromFalse
 where
   stealFromTrue = incVal pitC emptyOpposite totalStones
   stealFromFalse = incVal ((pitC*2)+1) emptyOpposite totalStones
   totalStones = (gState!!(index+((2*pitC)-(index*2)))) + 1
   emptyLast = pickUpStones index gState
   emptyOpposite = pickUpStones (index+((2*pitC)-(index*2))) emptyLast
```

1.9 The help-function `emptySpecificPit`

This function is called by sweapBoard to empty a players remaining stones one pit at a time, and add it to the this players kalaha.

```
emptySpecificPit (Kalaha pitC _) player index gState
 | player == False = nextFalse
 | otherwise = nextTrue
 where
  stones = gState!!index
  newState = pickUpStones index gState
  nextFalse = incVal pitC newState stones
  nextTrue = incVal (pitC*2+1) newState stones
```

1.10 The help-function `lastMove`

In this function we check for the other rules:

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 own kalaha
4. if player True lands in player own kalaha
5. nothing happens, and returns state and its next players turn

```
lastMove (Kalaha pitC stoneC) p pitIndex gState
 | (p == False) && (gState!!pitIndex == 1) && (pitIndex < pitC) = case1
 | (p == True) && (gState!!pitIndex == 1) && (pitIndex > pitC) && (pitIndex < ((pitC*2)
    +1)) = case2
 | (p == False) && (pitIndex == pitC) = case3
 | (p == True) && (pitIndex == pitC*2+1) = case4
 | otherwise = (not p, gState)
  where
  emptyFalseOpposite = (stealOpposite (Kalaha pitC stoneC) False pitIndex gState)
  emptyTrueOpposite = (stealOpposite (Kalaha pitC stoneC) True pitIndex gState)
  case1 = (True, emptyFalseOpposite)
  case2 = (False, emptyTrueOpposite)
  case3 = (False, gState)
  case4 = (True, gState)
```

1.11 The help-function `emptyCheck`

In this function we check if the pits of a player are empty. In that case the oppenent will collect all hes own stones to hes kalaha, and the game will end.

We do this by looking for the case where we get the value 'Nothing' from finding indices greater than zero in both player true and false's pits. So if a players pits are empty, we invoke the `sweapBoard` function for the opposite player which then collects hes own stones.

```
emptyCheck (Kalaha pitC stoneC) p pitIndex gState
 | (findIndex (>0) (fst(splitAt pitC newState)) == Nothing) = swap'
 | (findIndex (>0) (init(snd(splitAt (pitC+1) newState))) == Nothing) = swap'
 | otherwise = lastMove (Kalaha pitC stoneC) p pitIndex gState
  where
    swap = not p
    swap' = (swap, collect)
    newState = snd(lastMove (Kalaha pitC stoneC) p pitIndex gState)
    collect
     | p == False = sweapBoard (Kalaha pitC stoneC) True newState indexList l
     | p == True = sweapBoard (Kalaha pitC stoneC) False newState indexList l
     where
      l = ((length indexList) -1)
      indexList = findIndices (>0) newState
```

1.12 The help-function `sweapBoard`

So the sweapBoard function is invoked by emptyCheck, and recursively extracts the values in the pits into the correct players kalaha, by invoking the function emptySpecificPit. Ultimately it will return the final game state.

```
sweapBoard (Kalaha pitC stoneC) p gState indexList pitIndex
 | (pitIndex<0) = gState
 | (extractVal == pitC) = sweapBoard (Kalaha pitC stoneC) p gState indexList (pitIndex
    -1)
 | (extractVal == pitC*2+1) = sweapBoard (Kalaha pitC stoneC) p gState indexList (
    pitIndex-1)
 | otherwise = sweapBoard (Kalaha pitC stoneC) p execute indexList (pitIndex-1)
 where
    extractVal = indexList!!pitIndex
    execute = emptySpecificPit (Kalaha pitC stoneC) p extractVal gState
```

1.13 The function `showGameImpl`

For a given game and state we want a pretty output of the game state. For this we define tree lines, one for player true, one for the two kalahas, and one for player false. We use unlines to join the three lines with newlines appended, and also we map unwords to the three lines to seperate them by spaces.

With some help function **pad** and **part**, we align numbers perfectly. And by defining some empty space from the length of our list, we push line1 and line2 to their proper place.

```
showGameImpl :: Kalaha -> KState -> String
showGameImpl g@(Kalaha pitC stoneC) gameState =
  unlines $ map unwords [line1, line2, line3]
  where
```

```
    maxLen = length $ show $ 2*pitC*stoneC
    empty = replicate maxLen ' '

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

pad :: Int -> String -> String
pad pitC s = replicate (pitC - 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

```
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**

This function takes as parameters a depth, where we want to cut the children of the tree, a kalaha-game (number of pits and stones), a player and a game state.

Base case zero is the empty list, otherwise we recursively map the tree to the list.

```
takeTree :: Int -> Tree m v -> Tree m v
takeTree n (Node v list)
 | (n==0) = (Node v [])
 | otherwise = (Node v (map tree' list))
  where
   tree' (m,t) = (m, takeTree (n-1) t)
```

3 The Minimax algorithm

```
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
```

```haskell
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**

```haskell
type AlphaBeta = (Double,Double)

maxValue :: (Double,Double) -> (Maybe m, Double) ->  [(m, Tree m (Player, Double))] -> (Maybe m, Double)
maxValue (a,b) (m0,v0) [] = (m0,v0)
maxValue (a,b) (m0,v0) ((m1,c) : ch)
 | (v >= b) = (Just m1, v)
 | otherwise = maxValue (maxAlpha,b) (m,v) (ch)
  where
```

```haskell
   v1       = snd $ minimaxAlphaBeta (a,b) c
   (m,v)    = maxSnd (m0,v0) (Just m1,v1)
   maxAlpha = (max a v)



minValue :: (Double,Double) -> (Maybe m, Double) ->  [(m, Tree m (Player, Double))] ->
    (Maybe m, Double)
minValue (a,b) (m0,v0) [] = (m0,v0)
minValue (a,b) (m0,v0) ((m1,c) : ch)
 | (v <= a) = (Just m1, v)
 | otherwise = minValue (a,minBeta) (m,v) (ch)
  where
   v1      = snd $ minimaxAlphaBeta (a,b) c
   (m,v)   = minSnd (m0,v0) (Just m1,v1)
   minBeta = (min b v)



minimaxAlphaBeta :: AlphaBeta -> Tree m (Player, Double) -> (Maybe m, Double)
minimaxAlphaBeta (a,b) (Node (_,treeValue) []) =  (Nothing, treeValue)
minimaxAlphaBeta (a,b) (Node (p, v) ch)
 | p == True       = maxValue (a,b) (Nothing, (-1/0)) (ch)
 | otherwise       = minValue (a,b) (Nothing, (1/0)) (ch)
```

4 Testing and sample executions

As seen in picture X, we test the function **valueImpl** with two different cases: one for a kalaha game with six pits, and one with only two pits. In both cases we see that player True's kalaha is subtracted from player False's kalaha, with the output 7.0 and -1.0

```
*Kalaha> valueImpl (Kalaha 6 6) [4,0,2,0,0,1,15,2,10,0,2,0,13,22]
7.0
*Kalaha>
*Kalaha> valueImpl (Kalaha 2 2) [0,1,3,0,2,2]
-1.0
*Kalaha>
```

As seen in picture X, we test the function **movesImpl** with three different cases: one for player False, where we return hes pits which has positive elements, we test the same case only for player True, and one case where player False has zero positive elements, hence returning an empty list.

```
*Kalaha> movesImpl (Kalaha 6 6) False [4,0,2,0,0,1,15,2,10,0,2,0,13,22]
[0,2,5]
*Kalaha>
*Kalaha> movesImpl (Kalaha 6 6) True [4,0,2,0,0,1,15,2,10,0,2,0,13,22]
[7,8,10,12]
*Kalaha>
*Kalaha> movesImpl (Kalaha 6 6) False [0,0,0,0,0,0,50,10,5,0,0,0,7]
```

For the function **moveImpl** we look at three cases, one for a regular move for player True, as shown in picture X, one for the case where all player False's pits become empty, so player True collects the remaining stones to hes kalaha, as shown in picture X, and lastly, the case where player False lands in one of hes own empty pits, and steals the stones from the opposite side, as shown in picture X.

```
*Kalaha> moveImpl (Kalaha 6 6) True [0,5,2,2,1,0,31,0,0,7,0,4,0,16] 9
(False,[1,6,3,2,1,0,31,0,0,0,1,5,1,17])
```

```
*Kalaha> moveImpl (Kalaha 6 6) False [0,0,0,0,0,1,32,5,0,6,2,4,3,19] 5
(True,[0,0,0,0,0,0,33,0,0,0,0,0,0,39])
```

```
*Kalaha> moveImpl (Kalaha 6 6) False [4,0,2,0,0,1,15,2,10,0,2,0,13,22] 2
(True,[4,0,0,1,0,1,26,2,0,0,2,0,13,22])
```

In the function **showGameImpl** we test for two cases to make sure the output is pretty and aligned, so case one is for a kalaha with six pits, and second case for a kalaha with 4 pits. Both cases with number of one and two digits.

```
*Kalaha> putStrLn (showGameImpl (Kalaha 6 6) [0,5,2,2,1,0,31,0,0,11,0,4,0,16])
    0  4  0 11  0  0
16                    31
    0  5  2  2  1  0

*Kalaha>
*Kalaha> putStrLn (showGameImpl (Kalaha 4 4) [3,1,2,5,5,0,11,0,4])
    4  0 11  0
 4              5
    3  1  2  5
```

In the **treeImpl** we get the complete tree for all possible moves for a given player and kalaha state. We only test for a tree with two pits and two stones in each, otherwise the output tree would be too big.

```
*Kalaha> tree (kalahaGame (Kalaha 2 2)) (False, [2,2,0,2,2,0])
Node (False,0.0) [(0,Node (False,-1.0) [(1,Node (True,4.0) [])]),(1,Node (True,-1.0) [(3,Node (False
,0.0) [(0,Node (True,-1.0) [(3,Node (False,-1.0) [(1,Node (True,2.0) [])]),(4,Node (False,0.0) [(0,N
ode (True,0.0) [(3,Node (False,-2.0) [])]),(1,Node (True,-1.0) [(3,Node (True,0.0) [(4,Node (False,0
.0) [])])])])])]),(4,Node (False,0.0) [(0,Node (True,-1.0) [(3,Node (False,0.0) [(0,Node (True,0.0)
[(4,Node (False,-2.0) [])]),(1,Node (True,-1.0) [(3,Node (False,-1.0) [(0,Node (True,0.0) [])]),(4,N
ode (True,0.0) [(3,Node (False,2.0) [])])])])])])])]
```

takeTree of tree false 2 2

```
*Kalaha> takeTree 2 (tree (kalahaGame (Kalaha 2 2)) (False, [2,2,0,2,2,0]))
Node (False,0.0) [(0,Node (False,-1.0) [(1,Node (True,4.0) [])]),(1,Node (True,-1.0) [(3,Node (False
,0.0) []),(4,Node (False,0.0) [])])]
```

takeTree

```
*Kalaha> takeTree 0 testTree
Node 3 []
*Kalaha>
*Kalaha> takeTree 1 testTree
Node 3 [(0,Node 4 []),(1,Node 9 [])]
```

minimax

```
*Kalaha> minimax(takeTree 5 (tree (kalahaGame (Kalaha 6 6)) (True, [0,5,2,2,1,0,31,0,0,11,0,4,0,16])))
(Just 11,-12.0)
```

alfa beta

```
*Kalaha> minimaxAlphaBeta (-1/0,1/0) (takeTree 5 (tree (kalahaGame(Kalaha 6 6))
(True, [0,5,2,2,1,0,31,0,0,11,0,4,0,16])))
(Just 11,-12.0)
```