Programming Languages (Project 1)

Kristian Hartmann Hoeybye (krhoe16)

November 17, 2017

Abstract

The goal of this project is to create the basis for being able to play a game of kalaha in the ghci. The game is repressented as a list of integers where the integers are the amount of stone in the pits. The game can be initialised, the score can be seen and moves can be made to infulence the current game state. It's also possible to view the game state in a more pleasent way than just a list.

The possibility to create trees containing the outcomes of different moves will also be implemented so the optimal play which optimizes your points.

Contents

1 The Kalaha game with parameters (n, m)

```haskell
module Kalaha where

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**

startStateImpl creates the list holding the Kalaha game when it begins. The stone amount m is replicated n times to create player False's pits. Players begin with 0 points so using ++ [0], 0 is added to the list representing False's points. The same idea is then used to create player True's side of the board.

Test: startStateImpl (Kalaha 3 4) = [4,4,4,0,4,4,4,0]

```haskell
startStateImpl :: Kalaha -> KState
startStateImpl (Kalaha n m) = replicate n m ++ [0] ++ replicate n m ++ [0]
```

1.2 The function **movesImpl**

movesImpl goes through one side of the playing board and recursively checks 1 pit at a time if it contains any stones. If the pit contains stones then the KState at the index is >=1. If this is the case the index repressented as c is added to the list and if not the index is ignored. Using patternmatching the loop stops when the KState list is empty. With the boolean p it's decided which players pits will be look through because a player can only use his own pits.

Test: movesImpl (Kalaha 3 4) True [4,4,4,0,4,0,4,0] = [4,6] movesImpl (Kalaha 3 4) False [4,4,4,0,4,0,4,0] = [0,1,2]

```haskell
movesImpl :: Kalaha -> Player -> KState -> [KPos]
movesImpl (Kalaha n m) p s | p==False = notEmpty (take n (fst (split' n s))) 0
                           | p==True = notEmpty (take n (snd (split' n s))) (n+1)
      where
      split' n s = splitAt (n+1) s
      notEmpty [] _ = []
      notEmpty (x:xs) c | x>=1 = c: notEmpty xs (c+1)
                        | otherwise = notEmpty xs (c+1)
```

1.3 The function **valueImpl**

valueImpl extracts the values from the two indexes in the list repressenting the two players. True=2*n+1, False=n. False's points is then subtracted from True's points and the integer value is converted to a Double using fromIntegral.

Test: valueImpl (Kalaha 3 4) [0,5,5,2,3,2,1,5] = 3.0 valueImpl (Kalaha 3 4) [0,5,5,10,3,2,1,3] = -7.0

```haskell
valueImpl :: Kalaha -> KState -> Double
valueImpl (Kalaha n m) s = fromIntegral((((!!) s (2*n+1)) - ((!!) s n))
```

1.4 The function `moveImpl`

To manage the base scenario of a move in Kalaha where a pit is emptied and the stones are handed out to all the pits one by one, moveImpl creates multiple lists addStones takes the value of the chosen pit(amount of stones in it) and creates a list with the amount of times the stones can go all around the board, which is 2*n+1. This is replicated 2n+2 times - ex. addStones 20 3 =[2,2,2,2,2,2,2,2] extraStones then creates a list with 1's and 0's where the remaining stones are repressented as 1's. The list is then shifted using makeList, which uses the original index so pits getting an extra stone have a 1 and those not getting an extra has 0. - ex makeList 2 ((extraStones 2 ((!!) [1,3,2,0,2,1,5,0] 2)==[1,1,0,0,0,0,0,0]) 3 False)==[0,0,0,1,1,0,0,0] The lists generated by extraStones and addStones are then zipped with (+) so the corresponding indexes are added together. extraStones and addStones correctly gets the amount of stones each pit will recieve except for the opponents pit which should always recieve 0. Using opponentZero depending on whose turn it is the value at the index of the opponent is set to 0 in the zipped list. Then the list is (+) zipped with the original KState where value at the chosen pits index is set to 0.

There are 3 special rules which apply: 1) If the sowing ends in your own kalaha it is your turn to move again 2) if the sowing ends in an empty pit on your own side: All stones in the opposite pit (on theopponents side) along with the last stone of the sowing are placed into your kalaha and yourturn is over. 3) If a turn ends with all pits on one side being empty the opponent collects all the remaining stones in his pits

moveImpl returns a tuple (Player,KState) and to consider 1) nextTurn checks if the turn ends in the players own pit by comparing the, amount of stones modulo 2n+1, to the index of the players pit minus the index of the emptied one. If they're equal moveImpl returns p as the boolean since it's the same player to take another turn. Otherwise it return the opponents boolean.

It takes a KState, a Player, a KPos, the value of stones at the KPos and board length.

The function stoneThief handles the situations where 2) applies and the sowing ends in an empty pit on your own side. The conditions for this to happen is the pit where the sowing ends equal to 1, because it would then have been empty when recieving the last stone, and the end pit is on your own side of the board. The pit where the sowing ends is found using the helper function endPit. If these conditions are uphold stoneThief calls stealStones and returns the list which it creates. Depending on the boolean sets the value at the current players point pit index, to the sum of the values at the indexes i, 2*n-i and the index for the point pit itself. This is done with the helper function recieveStones. After doing this the values at index i and 2*n-1 are set to 0 with the helper functions ownPitZero and opponentPitZero.

gameEnd takes 3) into account and after a players turn the sum of each side of the playing board is individually checked. If the sum of either side is =0 the function creates a new list by replicating n 0's for each side and the value at the opposite players point pit is set to the sum of his side of the game with his own point pit included. THe list returned then contains all 0's except for the two point pits.

The implementation creates lists of length 2n+2 multiple times depending on which special rules are invoked for the current move. This in asymtotic notation is O(n)

Test: moveImpl (Kalaha 3 4) False [0,1,2,3,4,0,2,2] 2 = (True, [0,1,0,4,5,0,2,2]) moveImpl (Kalaha 3 4) False [1,0,2,3,4,5,2,2] 2 = (True, [0,0,2,9,4,0,2,2]) moveImpl (Kalaha 3 4) True [1,2,1,4,0,4,1,3] 6 = (True, [1,2,1,4,0,4,0,4]) moveImpl (Kalaha 3 4) True [1,1,1,5,0,0,1,9] 6 = (True, [0,0,0,8,0,0,0,10])

```
moveImpl :: Kalaha -> Player -> KState -> KPos -> (Player,KState)
moveImpl (Kalaha n m) p s i = (nextTurn p n i ((!!) s i),
                               gameEnd(stoneThief(zipWith (+) (opponentZero(zipWith (+)
    (addStones ((!!) s i) n) (makeList i (extraStones i ((!!) s i) n p))) p)
                               (init(fst(splitAt (i+1) s)) ++ [0] ++ snd(splitAt (i+1) s
    ))) p i ((!!) s i) n) n)

addStones:: Int->Int->[Int]
addStones v n = replicate (2*n+2) (v `div` (2*n+1))
```

```haskell
{- Calculates the amount of times you can go around the board putting a stone in each pit
(opponent point pit excluded) and then creates a list with length 2n+2 with this number.
-}

extraStones:: Int->Int->Int->Bool->[Int]
extraStones i v n b =replicate (leftovers v n) 1 ++ replicate ((2*n+2)-(leftovers v n)) 0
      where
      leftovers:: Int->Int->Int
      leftovers v n | b==True && (v `mod` (2*n+1)) > 2*n+1+n-i = 1+(v `mod` (2*n+1))
                    | b==False && (v `mod` (2*n+1))> 2*n-i = 1+(v `mod` (2*n+1))
                    | otherwise =(v `mod` (2*n+1))
      split l = splitAt i l
{- deals out the remaining stones when it's not possible to put a stone in each pit and creates a list with 1's and 0's.
   This list will be corrected so the pits that gets another stone will be correct and zipped with the list from addStones-}

makeList:: Int->[Int]->[Int]
makeList i l = snd(split i l) ++ fst(split i l)
        where
        split i l = splitAt (length(l)-(i+1)) l
{- Corrects the list from extraStones by shifting the list fitting to the index of the chosen pit. -}

opponentZero:: [Int]->Bool->[Int]
opponentZero l p | p==False = init(l) ++ [0]
                 | p==True = init(fst(splitAt (length(l) `div` 2) l)) ++ [0] ++ snd(splitAt (length(l) `div` 2) l)
{- Evaluates the boolean to check which player pit index number to set to zero
   in the list made by makeList so opponent doesn't get points in your turn -}
nextTurn:: Bool->Int->Int->Int->Bool
nextTurn p n i v | p==False && (v `mod` (2*n+1))==n-i = False
                 | p==False = True
                 | p==True && (v `mod` (2*n+1))==2*n+1-i = True
                 | p==True = False
{- Decides which player gets the next turn by checking if the current player ended in their own point pit. -}
stoneThief:: KState->Bool->KPos->Int->Int->KState
stoneThief s p i v n | p==False && ((!!) s endPit)==1 && endPit < n = stealStones
                    | p==True && ((!!) s endPit)==1 && endPit>n && endPit < (2*n+1) = stealStones
                    | otherwise = s
      where
      endPit | p == False = mod (i+(mod v (2*n+1))) (2*n+1)
             | p == True = mod ((mod (i-(n+1)+(mod v (2*n+1))) (2*n+1))+(n+1)) (2*n+2)

      stealStones = opponentPitZero (ownPitZero (recieveStones s))

      ownPitZero s = fst(splitAt endPit s) ++ [0] ++ tail(snd(splitAt endPit s))
      opponentPitZero s = fst(splitAt (2*n-endPit) s) ++ [0] ++ tail(snd(splitAt (2*n-endPit) s))
```

```
52
53      recieveStones s | p==False = init(fst(splitAt (n+1) s)) ++ [1+((!!) s (2*n-endPit
    ))+((!!) s n)] ++ snd(splitAt (n+1) s)
54                      | p==True = init s ++ [1+((!!) s (2*n-endPit))+((!!) s (2*n+1))
    ]
55
56 {- After sowing if the pit you end in only contains 1 stone these two pits in the list
    are set =0
57    and ther stone value is added to your point pool -}
58
59 gameEnd:: KState->Int->KState
60 gameEnd s n | sum(fst(splitAt n s))==0 = replicate n 0 ++ [(!!) s n] ++ replicate n 0
    ++ [sum(snd(splitAt (n+1) s))]
61           | sum(init(snd(splitAt (n+1) s)))==0 = replicate n 0 ++ [sum(fst(splitAt (n
    +1) s))] ++ replicate n 0 ++ [(!!) s (2*n+1)]
62           | otherwise = s
63 {- each side if all the pits are empty. If so the sum of all the pits on the opponents
    side goes to the opponent. -}
```

1.5 The function **showGameImpl**

showGameImpl respective generate the three lines top, middle and bottom seperately. Top is player True's side of the board, middle is the two Players point pits and bottom is player Fale's side of the board. Each pit is repressented by a digit amount of the function maxLen. For each pit, there will be printed ((maxLen)-the amount of digits for the value) amount of empty spaces, ''. This is then followed up by the value in the pit. for top and bottom maxLen+1 empty spaces is replicated before putting this so it's placed accordingly to player True's point pit appearing left most. the middle line is created by modelling the two point pits the same as the other pits. Between the two point pits empty spaces equal to (maxLen*n)+2 is put so player False's point pit is the right most and is 1 space further than the last side pit. these strings appears as lists of lists and by mapping unwords to the lines the result is only lists of strings. To finish up unlines is used to just get the separate strings.

Test: putStrLn(showGameImpl (Kalaha 3 4) [4,4,10,0,22,3,10,10]) = 10 3 22 10 0 4 4 10

```
1  showGameImpl :: Kalaha -> KState -> String
2  showGameImpl (Kalaha n m) s = unlines(map unwords [top, middle, bottom])
3          where
4          top = [replicate (maxLen+1) ' '] ++ nextNumber (reverse(init(snd(splitAt (n+1)
       s))))
5          middle = [replicate (maxLen-length(show (last s))) ' '] ++ [show (last s)] ++
       ([replicate (((maxLen)*n)+2) ' ']) ++ [show(head(snd(splitAt n s)))]
6          bottom = [replicate (maxLen+1) ' '] ++ nextNumber (fst(splitAt n s))
7          nextNumber [] = []
8          nextNumber (x:xs) = replicate (maxLen-(length(show x))) ""  ++ [show x] ++
       nextNumber xs
9
10         maxLen=length(show(2*n*m))
```

2 Trees

```
1  data Tree m v  = Node v [(m,Tree m v)] deriving (Eq, Show)
2  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 [])])]
```

2.1 The function takeTree

takeTree takes an integer n, which is the amount of layers in the tree to cover, and a Tree consisting of Nodes. Only the n-highest layers of the given tree is returned and the root is considered layer 0. This is done by recursively adding all the nodes in the layer below, by mapping the subfunction nextLevel to c which is the nodes' subtree. nextLevel calls takeTree recursively with n-1. So the amount of layers in the tree which are called will be n. Using pattern matching the loop breaks after takeTree is called with n=0 and where it no longer call itself.

Test: takeTree 0 testTree = Node 3 [] takeTree 1 testTree = Node 3 [(0, Node 4 []),(1, Node 9 [])] As supposed to in the description of the assignment.

```
takeTree :: Int -> Tree m v -> Tree m v
takeTree 0 (Node v _) = Node v []
takeTree n (Node v c) = Node v (map nextLevel c)
     where
     nextLevel (m, treeMV) = (m, takeTree (n-1) treeMV)
```

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
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

The function, tree, recieves a game, with a KState and a KPos, and a tuple containing a boolean and a KState. It then generates a Tree from the current state of the game by initially creating a Node(the root of the tree) that holds a tuple with which players turn it is and the point difference, generated by valueImpl. By recursively calling itself with the helper fucntion makeMove, which gets mapped on all the available moves(the pits that can be chosen), tree creates a list as the subtree for the current Node containing all the possible outcomes after a move was made.It's repressented as (i, Node (player, score)) where i is the index chosen for the move, player is the player who gets the next turn and score is the current score on the field with negatives repressenting False being ahead. Since this is done recursively after every move, all the outcomes of the moves that are possible for the new game state will be put in a list with the same idea. Because of this then when

movesImpl returns an empty list the game is over and the current node will have no subtree and is therefore a leaf of the tree. This means that all the leaves in the tree are the possible outcomes of a game.

Test: 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 (False,2.0) [])]),(4,Node (False,0.0) [(0,Node (True,0.0) [(3,Node (False,-2.0) [])]),(1,Node (True,-1.0) [(3,Node (True,0.0) [(4,Node (True,0.0) [])])])])])]), (4,Node (False,0.0) [(0,Node (True,-1.0) [(3,Node (False,0.0) [(0,Node (True,0.0) [(4,Node (True,-2.0) [])]),(1,Node (True,-1.0) [(3,Node (False,-1.0) [(0,Node (True,0.0) [])]),(4,Node (True,0.0) [(3,Node (False,2.0) [])])])])])])])]

```haskell
tree:: Game s m -> (Player, s) -> Tree m (Player, Double)
tree game (player, s) = Node (player, value game player s)
                             (map makeMove (moves game player s))
            where
            makeMove m = (m, tree game (move game player s m))
```

3.2 The function **minimax**

For the implementation code snippets given by our TA (henpe15) have been used. These are the functions maxSnd and maximumSnd, which are have also been used as templates when creating minimumSnd and minSnd. minimax returns the optimal move which for player True is the move where value is the highest and the smallest if you're player False.

When given a tree minimax either finds the max value or the min value depending on which player it is. If it's player False minimumSnd is used to find the min value from a list of tuples. The list of tuples is created from the function oneList which takes the subtree from a Node and recursively calls minimax to get the values at the different Nodes in the tree. Using pattern matching this stops and gives the value when a Node with no subtree(a leaf) is called to break the loop. When this list have been looked through by either maximumSnd or minimumSnd, for min or max, it returns the pit index to choose and the value given by doing so.

Test: minimax (tree (kalahaGame (Kalaha 2 2)) (False, [2,2,0,2,2,0])) = (Just 1,2.0)

```haskell
minimax:: Tree m (Player, Double) -> (Maybe m, Double)
minimax (Node (_,value) []) =  (Nothing, value)
minimax (Node (player, value) s)  | player==False =minimumSnd (oneList s)
                                  | player==True = maximumSnd (oneList s)
oneList [] = []
oneList ((path, subtree):xs) = (Just path, snd(minimax subtree)): oneList xs

maximumSnd :: [(a, Double)] -> (a, Double)
maximumSnd [] = error "undefined for empty list"
maximumSnd (x:xs) = foldl maxSnd x xs
   where
   maxSnd :: (a,Double) -> (a, Double) -> (a, Double)
   maxSnd a@(_,v1) b@(_,v2) | v1 >= v2 = a| otherwise = b

minimumSnd :: [(a, Double)] -> (a, Double)
minimumSnd [] = error "undefined for empty list"
minimumSnd (x:xs) = foldl minSnd x xs
    where
    minSnd :: (a, Double) -> (a, Double) -> (a, Double)
    minSnd a@(_,v1) b@(_,v2) | v1 <= v2 = a| otherwise = b
```

3.3 The function **minimaxAlphaBeta**

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

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

4 Testing and sample executions

one or more of the tests done for each individual function can be found at their respective function descriptions. Besides this the KalahaTest.hs has been run and returns with all 100 tests passed for each categori.

GameStrategisTest.hs has also been run and the first test called prop_minimaxPicksWinningStrategyForNim returns with passed for 100 tests. The other tests here returns Failed as they require the function minimaxAlphaBeta to be working which here is not made.