Programming Languages (Project 1)

David Aron Hartig Kaan (dakaa16)

November 13, 2017

Abstract

The goal of this project is to implement the rules of a game of Kalaha, as described in the project description. A series of functions that have to be implemented have been given. Succesfully implementing all of the functions, will result in an environment where one can play a game of kalaha.

Additionally a function to cut off all the leaves of a tree from a given depth is to be implemented, as well as the Minimax algorithm. All of this is to be implemented in haskell.

Due to time constraints, the Minimax algorithm is not implemented, and therefore not part of this report.

Contents

1 The Kalaha game with parameters (n, m)

In order to implement a game of Kalaha, a number of types are declared. All of these are documented in the project description. Together they make up everything needed to implement a game of Kalaha.

The code of the declarations can be seen below.

```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`

The function startStateImpl, outputs a KState, containing the starting point of a given game of Kalaha. The parameters of a Kalaha - namely PitCount and StoneCount - are used as arguments in replicate. PitCount is given as the number of replicates while StoneCount is the number being replicated. This gives out a list of the number StoneCount with PitCount as length. Here the concatenation operator is used, to append a list containing one element whch is 0. This is the Kalaha of Player false. The same process is done for the pits and Kalaha of Player True.

The implementation of `startStateImpl` can be seen below.

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

1.2 The function `movesImpl`

The function `movesImpl` outputs a list of KPos-es, that represents the indexes of the pits with more than 0 stones in them for a given player. In this implementation pattern matching have been used to distinguish the two players from each other.

The function findIndexes is from the 'Data.List' library. It returns a list containing the indexes of the elements in a given list, that complies with a given condition. The condition given, is that the element is to be bigger than zero, and the list is the pits of the player.

For Player false, the list of pits is simply gotten by splitting the list at the first Kalaha, and using the first element in the tuple it creates.

For Player true, the approach is almost the same, only a list of 0's are concatenated at the front of the list, as to get the right indexes. Also the function init is used, to remove the last Kalaha from the list. In order to avoid getting the first Kalaha, the lists are splitted at PitCount + 1.

The implementation of `movesImpl` can be seen below.

```haskell
movesImpl :: Kalaha -> Player -> KState -> [KPos]
movesImpl (Kalaha n m) False s = findIndices (>0) (fst(splitAt n s))
movesImpl (Kalaha n m) True s = findIndices (>0) (replicate (n+1) 0 ++
          (init (snd(splitAt (n + 1) s))))
```

1.3 The function **valueImpl**

This function takes a KState and returns the difference from Player true to Player false as the data type double. The list index operator (!!) is used, in order to get the number of stones of each players Kalaha. The function fromIntegral turns the Integral Int gotten from the list index operater into any numeric type. Double is a numeric type. From there, it is possible to just subtract Player false's Kalaha StoneCount from Player True's.

The implementation of `valueImpl` can be seen below.

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

1.4 The function **moveImpl**

The function `moveImpl` is to make a move on the Kalaha board, given a a Kalaha, a Player, a KState and a KPos which is the index of where the move starts. The function returns a tuple made out of a Player and a KState affected of the move. Three additional functions have been made in order to achieve this:

`replaceAt` replaces an element in KState, with a given element, at a given index.

The function `replaceAt` can be seen below.

```haskell
replaceAt :: Kalaha -> KState -> Int -> Int -> KState
replaceAt g s x v = (take x s ++ [v] ++ drop (x + 1) s)
```

`holesEmpty` calls `movesImpl` on both players and a given KState, and checks if the lists are empty - signifying that one or both of the players' side of pits are completely empty. The function uses guards to distinguish which of the player's (if any) side of pits are empty.

The function returns a tuple of a Player and a KState, following the rules given for this instance. If none of the lists are empty, the output will not be changed from the input.

The function `holesEmpty` can be seen below.

```haskell
holesEmpty :: Kalaha -> (Player, KState) -> (Player, KState)
holesEmpty g@(Kalaha n m) (p, s)
    |(movesImpl g True s)  == [] = (p, (replicate n 0 ++ [((s!!n) + sum (fst(splitAt n s
     )))]++ replicate n 0 ++ [(s!!(2*n+1))]))
    |(movesImpl g False s) == [] = (p, (replicate n 0 ++ [s!!n] ++ replicate n 0 ++[((s
     !!(2*n+1)) + sum (snd(splitAt (n+1) (init s))))]))
    |otherwise = (p, s)
```

`sow` is a recursive function, sowing from a given index, until there are no more stones to sow. The function uses guards in order to know when to return, when to go from max index back to index 0, when to skip a pit (opponents' Kalaha) and when to sow normally.

The function returns a tuple consisting of an Int describing the index after the index of which the sowing ended and a KState that has been changed by the sowing.

The function `sow` can be seen below.

```haskell
sow :: Kalaha -> KState -> Int -> Int -> Int -> (Int, KState)
sow g@(Kalaha n m) s stones xs skip
    |xs == (2*n + 2) = sow g s stones 0 skip
    |xs == skip = sow g s stones (xs + 1) skip
    |stones == 0 = (xs, s)
    |otherwise = sow g (replaceAt g s xs (s !! xs + 1)) (stones - 1) (xs + 1) skip
```

The implementation of moveImpl calls the three functions mentioned above, in order to succesfully make a move following the rules of Kalaha.

Pattern matching is used to distinguish the two players from one another. In this report, the case of one player in general will be explained, as the cases for the two players are fairly similar.

A **where** is used to declare a variable t. This is a tuple, holding the values returned by calling sow. The input given for sow is the KState given to movesImpl, but with the element at KPos set to 0, as, that is where the sowing starts. Also the stones given to sow, is short 1 stone. This is done in order to check the condition of the pit, before the stone is added.

The three guards are used to distinguish whether the last index of the sowing is on the players side, Kalaha or on the opponents side. If it is on the opponents side, the stone is simply put at the given index and the function returns a tuple of the other Player and the KState. If the index is at the players Kalaha, the same happens, only the turn of the player does not change. If the index is at the players side, an if statement is raised, using movesImpl in order to determine whether the pit is empty or not. If the pit is not empty, the same happens as if it had landed at the opponents side. If the pit is empty, the stone, as well as all the stones from the pit at the opposite side goes into the players Kalaha and it is the other players turn.

At each guard holesEmpty is called on what would actually just be returned. This is done, as to check after every move made, if all the pits at one side is empty. If not, it will just return the input.

The effeciency of the algorithm was not the biggest thing in mind when implementing the function. This results in an unstructured and slow implementation, that does not take full advantage of Haskell.

The implementation of moveImpl can be seen below.

```haskell
moveImpl :: Kalaha -> Player -> KState -> KPos -> (Player,KState)
moveImpl g@(Kalaha n m) False s xs
   |fst t < n = if (fst t) `elem` (movesImpl g False (snd t))
     then holesEmpty g (True, (replaceAt g (snd t) (fst t) (snd t !! fst t + 1)))
     else holesEmpty g (True, (replaceAt g
     (replaceAt g (snd t) n ((snd t !! n) + (snd t !! (2*n - fst t)) + 1))
     (2*n - fst t) 0))
   |fst t == n = holesEmpty g (False,(replaceAt g (snd t) (fst t)(snd t !! fst t + 1)))
   |fst t > n = holesEmpty g (True, (replaceAt g (snd t) (fst t) (snd t !! fst t + 1)))
   where t = sow g (replaceAt g s xs 0) ((s!!xs)-1) (xs + 1) (2*n + 1)
moveImpl g@(Kalaha n m) True s xs
   |fst t < n = holesEmpty g (False, (replaceAt g (snd t) (fst t)(snd t !! fst t + 1)))
   |fst t == (2*n + 1) = holesEmpty g (True,(replaceAt g (snd t)(fst t)(snd t !! fst t
    + 1)))
   |fst t > n = if (fst t) `elem` (movesImpl g True (snd t))
     then holesEmpty g (False, (replaceAt g (snd t) (fst t) (snd t !! fst t + 1)))
     else holesEmpty g (False, (replaceAt g
     (replaceAt g (snd t) (2*n+1) ((snd t !! (2*n+1)) + (snd t !! (2*n - fst t)) + 1))
     (2*n - fst t) 0))
   where t = sow g (replaceAt g s xs 0) ((s!!xs)-1) (xs + 1) (n)
```

1.5 The function showGameImpl

The function showGameImpl is to give a visual representation of the Kalaha board, given a Kalaha and a KState. This is done by a **String**. Three additional functions have been made in order to achieve this:

The function padding uses the maximum length of the decimal representation for a pit on a Kalaha of a given size, to pad space(s) in front of strings that are smaller than max. The output will be a string that is one character longer than the maximum length, as each pit is to have at least one space between them.

The `middle` function will make the middle row of the Kalaha board string, namely the Kalahas of the two players. The function takes two strings, which is the two Kalahas, expected to already have padding. Thus it is only needed to make space between the two and concatenate them. The spaces are made by simply using **`replicate`** to make a space `(PitCount + PitCount * maximum length)` times. This in length is equal to one of the players' side of pits.

The function `upperUnder` takes a list of strings and expect them all to be padded. Spaces equal to an extra element is concatenated, to the front, to make space for the Kalaha on the left. **`show`** is called on the list of strings, which gives out unwanted characters, which is removed by a filter.

The functions `padding`, `middle` and `upperUnder` can be seen below.

```haskell
padding :: Kalaha -> String -> String
padding (Kalaha n m) s = replicate (((length(show(2*n*m))) - length s)+1) ' ' ++ s

middle :: Kalaha -> String -> String -> String
middle (Kalaha n m) i j = j ++ (replicate (n+(n*(length(show(2*n*m))))) ' ') ++ i ++ "\
    n"

upperUnder :: Kalaha -> [String] -> String
upperUnder (Kalaha n m) xs = replicate ((length(show(2*n*m)))+1) ' ' ++ (filter(not .
    (`elem`",[]\"")) (show xs))
```

A **`where`** is used to declare a variable `S`. This variable is a list of strings. The function **`show`** is mapped by a **`map`** to the KState, in order to make every element a **`String`**. The function `padding` is the mapped to the list of strings, in order to make them all padded.

`upperUnder` is called on `S`, splitted to only hold the pits of `Player` true and reversed. This is concatenated to a new line character, which is concatenated to the middle part, generated by `middle`. Lastly the pits of `Player` false are concatenate, these have been made into a **`String`** using `upperUnder`.

The implementation of `showGameImpl` can be seen below.

```haskell
showGameImpl :: Kalaha -> KState -> String
showGameImpl g@(Kalaha n m) xs = (upperUnder g (reverse(init(snd(splitAt (n + 1) s)))))
     ++ "\n" ++ (middle g (s !! n) (s !! (2*n+1))) ++ (upperUnder g (fst(splitAt (n) s)
    ))
   where s = map (padding g ) (map show xs)
```

2 Trees

The data type `Tree` is seen below.

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

2.1 The function `takeTree`

The function `takeTree` cuts off all the leaves of a tree from a given depth.

Pattern matching is used to catch the instance where the depth (`x` in code) is set to 0. In that instance, the function returns the top node, with an empty list, leaving it with no leaves. In all other instances, the function returns the top node with its list of leaves being changed.

`fst` is mapped to `m` in order to get a list of the `m`'s of all the node in question's children. A recursive call, with depth - 1 is then mapped to the second half of `m`, making a list of all the `Tree`'s in the node in question.

This resursive call will make sure that every node is checked for children, but the recursive call will never get further down the `Tree`, than the given depth will allow, thus recreating the `Tree` with a max depth.

```haskell
takeTree :: Int -> Tree m v -> Tree m v
takeTree 0 (Node v m) = Node v []
takeTree x (Node v m) = (Node v (zip (map (fst) m) (map (takeTree (x-1) . snd) m)))
```

3 Testing and sample executions

For the Kalaha part, the testing was mostly done, by printing values at certain points in the code. When implementing the function `moveImpl`, having a lot of different test data was important, as the function got hard to grasp. This is where the file KalahaTest.hs was helping a lot, as it brought a lot of varied testing. The implementation of Kalaha have passed all the test given by this file.

Having tested `moveImpl`, it has some shortcomings. It works fine in regard to the minimax algorithm, however not so well for human input. The function does not take mistakes into account. One might start sowing from the Kalaha or an empty pit, without the function catching the mistake. In conclusion; the function will follow the rules of Kalaha, if the player does.

Implementing `takeTree` took a lot of testing, with messy trees. A lot of helper functions was written, in the process of coming up with a simple and elegant solution. A lot of varied tree structures was tested and in the end, the function could handle them all correctly.