Programming Languages (Project 1)

Jakob Kjær-Kammersgaard (jakkj16)

November 17, 2017

Abstract

The overarching purpose of this project has been to implement the rules of the game, Kalaha, abstracted to having a dynamic amount of pits and stones in each pit. Additionally, it has been about implementing a rudimentary AI in the form of minimax with and without alpha-beta pruning. The tasks have been fulfilled to the tertiary goal of having not only a working implementation, but one that takes advantage of functional programming principles on top of being generally high quality code.

Contents

1 Preface

In general I have strived towards having the code be as easily humanly parseable and self-documenting as possible; namely, by favoring longer, more descriptive symbol names over those of the one- or two-letter variety perhaps more canonical to Haskell. Exceptions are common abbreviations or instances where a value needs to be bound to an intermediate symbol before being used in another expression.

2 The Kalaha game with parameters (n, m)

```haskell
module Kalaha where
import Data.List
import Debug.Trace

mtrace :: Show a => String -> a -> a
mtrace str arg = trace ('<':str ++ ": " ++ show arg ++ ">\n") arg

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

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

I feel sorry for Alice and Bob, they never get to just sit down together and play a nice game of Kalaha. Instead, they're fighting to make their insecure long-distance relationship work, all while being eavesdropped upon or having someone tamper with their messages.

Well not this time!

```haskell
(alice, bob) = (False, True)
```

2.1 The function `startStateImpl`

A simple function that sets up the initial state of a game of Kalaha given how many pits it has, and how many stones are to be placed in each pit.

Since both sides of a kalaha game is initially the same, a shorthand, `side`, is used to duplicate the pattern of all the pits on a side filled with the given amount of stones followed by the initially empty store on that side.

```haskell
startStateImpl :: Kalaha -> KState
startStateImpl (Kalaha pitCount stoneCount) = side ++ side
  where side = (replicate pitCount stoneCount) ++ [0]
```

2.2 The function `movesImpl`

Given a player and kalaha board state, returns a list of valid indexes on the board for that player to start a move from. Since it is the indexes we are interested in, we associate each position on the board with an index by zipping the board state with `[0..]`. Then the pits belonging to the given player are extracted from the association list by first dropping the offset to that player's side, second, taking the amount of elements that there is pits on one side. Finally, a list comprehension is used to filter by non-empty pits.

```haskell
movesImpl :: Kalaha -> Player -> KState -> [KPos]
movesImpl (Kalaha pitCount _) player state =
  [index | (index, value) <- indexedPlayerSide, value > 0]
  where
    indexedPlayerSide = take pitCount . drop playerOffset $ zip [0..] state
    playerOffset = if player then pitCount+1 else 0
```

2.3 The function `valueImpl`

The difference in score between alice and bob. Alice wants to minimize this value while bob wants to maximize it. This function is realized with a supporting local function, `score` that just returns the score of the given player. `fromIntegral` is used to convert the result from an `Int` to a `Double`.

```haskell
valueImpl :: Kalaha -> KState -> Double
valueImpl (Kalaha pitCount _) state = fromIntegral $ score bob - score alice
  where score player = state !! ( pitCount + if player then pitCount+1 else 0 )
```

2.4 The function `moveImpl`

Takes all the necessary inputs for making a move in a game of Kalaha, the board setup, the player making the move, the game state before the move, and the index of the position at which the player would like to grab the initial stones from.

Returns the player's who gets to make the following move in addition to the state of the board after the move.

This function has several individual concerns and have thus been composed by smaller internal functions that each perform part of the move.

The important steps of a move in kalaha are: grabbing the stones from the starting index, sowing the grabbed stones around the board, skipping the opponent's store, and handling the different possible outcomes of the last dropped stone.

These steps are roughly represented in a clear fashion in the code lines before the `where` clause, at the very top of `moveImpl`.

It is worthy of note that unlike a real game of Kalaha, this implementation does not iterate through each state that the board is going to be in from beginning to end of the move, like if the game was strictly simulated.

For instance, in an arbitrary Kalaha game, the amount of stones grabbed initially could be enough to make several full round trips on the board.

Instead of sewing through all the stones in linear time with respect to this potentially high amount of stones, this implementation calculates both the quotient and the remainder of dividing the amount of drop spots on the board (all spots but the opponent's store); and because the quotient signifies the amount of full round trips that have to be made, time and memory can be saved by simply adding the quotient to each of the drop spots in linear time with respect to the board size instead of the stone count.

Now only the remaining stones need to be sown, which appropriately the remainder holds the amount of. These are all trivially distributed to where they need to go, all except the last stone which is special-cased by the helping function, `dropLastStone` which handles all rules for the last stone.

Finally, if the move resulted in the game ending, another helping function, `cleanupIfGameOver` is applied to the returned state, putting all stones in the store of the player to whom they belong.

```haskell
moveImpl :: Kalaha -> Player -> KState -> KPos -> (Player, KState)
moveImpl (Kalaha pitCount stoneCount) player startState startPosition =
  let
    (grabbedStones, stateAfterGrab) = grabStones startPosition startState
    (numRoundtrips, stonesToSowe)   = (grabbedStones-1) `quotRem` (boardSize-1)
    stateAfterRoundtrips            = addRoundtrips numRoundtrips stateAfterGrab
    sowes = iterate sowe (stateAfterRoundtrips, positionAfter startPosition)
  in
    applyToSnd cleanupIfGameOver $ dropLastStone $ sowes !! stonesToSowe
  where
    -- Common expressions and/or for conveying intent
    boardSize     = 2*pitCount+2
    dropStone     = modifyAtIndex (+1)
    grabStones    = getAndModifyAtIndex (const 0)

    (ownStore, opponentStore) =
      onCondition (player == bob) swapTuple (pitCount, 2*pitCount+1)

    addRoundtrips :: Int -> KState -> KState
    addRoundtrips n = modifyAtIndex (subtract n) opponentStore . map (+n)

    positionAfter :: KPos -> KPos
    positionAfter = head . filter (/= opponentStore)
                  . tail . iterate ((`mod` boardSize).(+1))

    sowe :: (KState, KPos) -> (KState, KPos)
    sowe (state, position) = (dropStone position state, positionAfter position)

    cleanupIfGameOver :: KState -> KState
    cleanupIfGameOver state
      | not gameOver = state
      |     gameOver = emptyPits ++ aliceFinalScore : emptyPits ++ [bobFinalScore]
      where -- It was the last move if a whole side of pits got empty
        gameOver = or $ map (all (==0) . init) [aliceSide, bobSide]
        (aliceSide, bobSide) = splitAt (pitCount+1) state
        aliceFinalScore = sum $ take (pitCount+1) state
        bobFinalScore   = (2*pitCount*stoneCount - aliceFinalScore)
        emptyPits = replicate pitCount 0

    dropLastStone :: (KState, KPos) -> (Player, KState)
    dropLastStone (state, position)
      | position == ownStore               = (player, dropStone position state)
      | state !! position == 0 && onOwnSide = (not player, stateOppositeStones)
      | otherwise                          = (not player, dropStone position state)
      where
        statePutDownLastStone = dropStone position state
        onOwnSide = position < ownStore && position >= (ownStore - pitCount)
        stateOppositeStones = modifyAtIndex (+(stones+1)) ownStore stateGrab
          where
            (stones, stateGrab) = grabStones otherSidePit state
            otherSidePit = boardSize - 2 - position --, 'position' /= <a store>
```

2.5 The function `showGameImpl`

Creates a string representation of a Kalaha game board given initial board settings and the the state of the board to be represented.

This function makes heavy use of abstract helper functions that do very specific but commonly applicable things, like the library function **unwords**, which takes a list of strings and joins them with a linebreak.

Also in use is such a function, not from the standard library, but defined in this source text; it is, `applyToBothInPair` which handily applies a function to both values in a tuple.

```
showGameImpl :: Kalaha -> KState -> String
showGameImpl (Kalaha pitCount stoneCount) state =
  unlines $ map unwords [line1, line2, line3]
  where
    ((aliceSide, aliceKalaha), (bobSide, bobKalaha)) =
          applyToBothInPair (splitAt pitCount)
        $ splitAt (pitCount+1)
        $ map (leftPad ' ' maxLength . show) state

    line1 = emptySlot : (reverse $ bobSide)
    line2 = bobKalaha ++ (replicate pitCount emptySlot) ++ aliceKalaha
    line3 = emptySlot : (aliceSide)

    maxLength = length . show $ 2*pitCount*stoneCount
    emptySlot = replicate maxLength ' '
```

3 Trees

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

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

Show instance for Tree derived from similar implementation in Data.Tree module. It is used solely for debugging. NB: I have shared this implementation with some of my peers, so it might show up elsewhere.

```
instance (Show m, Show v) => Show (Tree m v) where
  show t = unlines $ draw t
    where
      draw :: (Show m, Show v) => Tree m v -> [String]
      draw (Node value children) = show value : drawSubTrees children
        where
          drawSubTrees [] = []
          drawSubTrees [(branchID, t)] = let (x:xs) = draw t in
              "|" : shift "`- " "    " (('{': show branchID ++ "} => " ++ x) : xs)
          drawSubTrees ((branchID, t):ts) = let (x:xs) = draw t in
              "|" : shift "+- " "|   " (('{': show branchID ++ "} => " ++ x) : xs) ++
    drawSubTrees ts
```

```
12
13          shift first other = zipWith (++) (first : repeat other)
```

3.1 The function `takeTree`

Cuts off a tree at a given depth.

The helping function applyToSnd enables a neat map over the branches in the tree, by recursively applying `takeTree` to the second element of the tuples that represent the branches, leaving the `m`'s in the tree alone.

```
1 takeTree :: Int -> Tree m v -> Tree m v
2 takeTree 0 (Node v _)  = Node v []
3 takeTree _ (Node v []) = Node v []
4 takeTree d (Node v ts) = Node v $ map (applyToSnd $ takeTree (d - 1)) ts
```

4 The Minimax algorithm

```
1  data Game s m = Game {
2      startState    :: s,
3      showGame      :: s -> String,
4      move          :: Player -> s -> m -> (Player,s),
5      moves         :: Player -> s -> [m],
6      value         :: Player -> s -> Double}
7
8  kalahaGame :: Kalaha -> Game KState KPos
9  kalahaGame k = Game {
10     startState = startStateImpl k,
11     showGame   = showGameImpl k,
12     move       = moveImpl k,
13     moves      = movesImpl k,
14     value      = const (valueImpl k)}
15
16 startTree :: Game s m -> Player -> Tree m (Player,Double)
17 startTree game player = tree game (player, startState game)
```

4.1 The function `tree`

Lazily generates the full game tree of a game of Kalaha.

The function gets the moves possible for a given player from `movesImpl` and then maps `nodeForMove` over all of those possible moves getting the resulting next player's and next game states to recursively continue generating from, down the tree.

```
1 tree :: Game s m -> (Player, s) -> Tree m (Player, Double)
2 tree game (player, state) = Node (player, theValue) (map nodeForMove theMoves)
3   where
4     theValue = value game player state
5     theMoves = moves game player state
6     nodeForMove aMove = (aMove, tree game outcomeOfMove)
7       where
8         outcomeOfMove = move game player state aMove
```

4.2 The function `minimax`

Algorithm that tries to minimize the other player's ability to force bad moves on you.

Essentially it works by playing all possible games ahead of the current game state, before returning the initial move that, given optimal, adversarial play by both player's, result in the best guaranteed score for the player it was trying to optimize for.

This implementation tries to be minimally redundant, making use of the custom helper function `bestBy` which is comparable to the standard library functions **`minimumBy`** and **`maximumBy`** apart from the criterium for what is best when comparing two elements being a parameter to the function, making it more general.

Also, `applyToPair` is used to apply two different functions to the first and second element the tuples representing the initial move and the best guaranteed value found in one of the leaves of the game tree.

```
minimax :: Tree m (Player, Double) -> (Maybe m, Double)
minimax (Node (_, value) []) = (Nothing, value)
minimax (Node (maximizer, _) branches) =
    applyToPair (Just, snd)
  $ bestBy (preference `on` (snd.snd))
  $ map (applyToSnd minimax) branches
  where preference = if maximizer then (>) else (<)
```

4.3 The function `minimaxAlphaBeta`

Minimax again this time with alpha-beta pruning.

It relies on the invariant that alpha and beta always represent the best already explored values for the maximizing and minimizing player's respectively along the path to the root.

When minimax gets to a new subtree it always needs to check the first branch, regardless of alpha-beta pruning; this gives a lower bound on that entire subtree saying that, it can be no worse than that.

The pruning step can happen right before the next branch in the same subtree is checked, now the algorithm can look at the best value for the previous player, higher up in the tree, by looking at either alpha or beta, depending on whether it was the maximizing or the minimizing player before, and compare that value to the newly found lower bound in this subtree; if this subtree already guarantees an outcome better for you, but worse for the previous player, the previous player will never let the game go down into this subtree, thus, no further branches need checking, they can be pruned.

This implementation varies quite a bit from the straight-forward `minimax` implementation. Since, by definition, not all subtrees are visited, a simple **`map`** over the branches with recursive calls for each item, is not an option. Instead, this implementation has two base cases. The first is for when the algorithm hits a leaf node, which is the only time it gets passed a node with an empty list of branches; in this case it simply returns the value in the leaf node. The second base case is when only a single branch is in the node. It gets to this case even without it occurring naturally in the game tree because the the algorithm calls itself recursively both vertically, like in minimax, but also horizontally, for each branch in the current subtree, passing on only the tail of the list of branches, excluding the one just checked.

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

-- Alpha: best already explored option along the path to the root for the maximizer
-- Beta: best already explored option along the path to the root for the minimizer
minimaxAlphaBeta :: AlphaBeta -> Tree m (Player, Double) -> (Maybe m, Double)
minimaxAlphaBeta (a, b) (Node nodeValue@(maximizer, gameValue) branches) =
  case branches of
    [] -> (Nothing, gameValue)
    [(childMove, childTree)] -> (Just childMove, snd $ minimaxAlphaBeta (a, b)
    childTree)
    ((childMove, childTree):restOfBranches) ->
      let childNodeValue@(_, childGameValue) = (Just childMove, snd $ minimaxAlphaBeta
    (a,b) childTree)
          (ab', prune, preference)
            | maximizer = ( (max a childGameValue, b), b <= childGameValue, (>) )
            | otherwise = ( (a, min b childGameValue), a >= childGameValue, (<) )
          bestOfRestOfChildren = minimaxAlphaBeta ab' (Node nodeValue restOfBranches)
      in if prune
        then childNodeValue
        else bestOfTwoBy (preference `on` snd) childNodeValue bestOfRestOfChildren
```

5 Helper Functions

Simple function that does a uniform transformation on both arguments (in the same typeclass) of a binary operator.

```haskell
on :: (a -> a -> b) -> (c -> a) -> c -> c -> b
on f transformation = \x y -> f (transformation x) (transformation y)
```

Replaces the element at index 'index' with the function 'op' applied to that element returning a tuple of the original element and the modified list

```haskell
getAndModifyAtIndex :: (a -> a) -> Int -> [a] -> (a, [a])
getAndModifyAtIndex op index list =
  case splitAt index list of
      (list, []) -> error "Index out of range"
      (ys, x:xs) -> (x, ys ++ (op x : xs))
```

Same as 'getAndModifyAtIndex' but only returns the modified list.

```haskell
modifyAtIndex :: (a -> a) -> Int -> [a] -> [a]
modifyAtIndex o i l = snd $ getAndModifyAtIndex o i l
```

Pads a string on the left side with 'c' if it is shorter than 'toLength'.

```haskell
leftPad :: Char -> Int -> String -> String
leftPad c toLength str = replicate padLength c ++ str
  where padLength = max 0 (toLength - (length str))
```

Makes a function only do something when a condition holds.

```haskell
onCondition :: Bool -> (a -> a) -> (a -> a)
onCondition False _ = id
onCondition _ f = f
```

Miscellaneous functions facilitating function application in tuples.

```
applyToPair :: ((a -> b), (c -> d)) -> (a, c) -> (b, d)
applyToPair (f1, f2) (l, r) = (f1 l, f2 r)
applyToBothInPair f = applyToPair (f, f)
applyToFst f = applyToPair (f, id)
applyToSnd f = applyToPair (id, f)
swapTuple (a,b) = (b,a)
```

Function that takes the *first*, best element in a list by comparing each element to the best found-so-far element according to a passed binary comparison operator that defines the criterium for what is the better of two elements.

```
bestBy :: (a -> a -> Bool) -> [a] -> a
bestBy _ [] = error "Cannot get element from empty list"
bestBy cmp (x:xs) = foldl (bestOfTwoBy cmp) x xs

bestOfTwoBy :: (a -> a -> Bool) -> a -> a -> a
bestOfTwoBy cmp a b | b `cmp` a = b
                    | otherwise = a
```

6 Testing and sample executions

Mostly, testing has been making shure the code passes the provided QuickCheck-based tests and trying to quickly get them working when they occationally broke.

Other than that, using the custom Show instance for trees, I have inspected the output of "spoon-fed" input to mostly the functions `startTree`, `minimax` and `minimaxAlphaBeta`.

`showGameImpl` has just been giving the expected output every time since it was completely implemented, and it has been but thoroughly (not exhaustivly) tested.

6.1 Working `showGameImpl`

```
*Kalaha> let g = Kalaha 5 0 in putStrLn $ showGameImpl g $ startStateImpl g
  0 0 0 0 0
0           0
  0 0 0 0 0

*Kalaha> let g = Kalaha 5 1 in putStrLn $ showGameImpl g $ startStateImpl g
    1  1  1  1  1
 0                 0
    1  1  1  1  1

*Kalaha> let g = Kalaha 5 10 in putStrLn $ showGameImpl g $ startStateImpl g
     10  10  10  10  10
  0                       0
     10  10  10  10  10

*Kalaha> let g = Kalaha 5 100 in putStrLn $ showGameImpl g $ startStateImpl g
      100  100  100  100  100
   0                             0
      100  100  100  100  100
```

```
*Kalaha> let g = Kalaha 6 6 in (...) $ [3,0,1,0,0,0,27,4,1,0,0,6,10,20]
   10  6  0  0  1  4
20                   27
    3  0  1  0  0  0
```

6.2 Working QuickCheck-based test

```
Prelude> :l KalahaTest
[1 of 2] Compiling Kalaha           ( Kalaha.lhs, interpreted )
[2 of 2] Compiling Main             ( KalahaTest.hs, interpreted )
Ok, modules loaded: Main, Kalaha.
*Main> main
=== prop_startStateLen from KalahaTest.hs:33 ===
+++ OK, passed 100 tests.
=== prop_startStateSum from KalahaTest.hs:37 ===
+++ OK, passed 100 tests.
=== prop_startStateValue from KalahaTest.hs:41 ===
+++ OK, passed 100 tests.
=== prop_startStateSymmetric from KalahaTest.hs:47 ===
+++ OK, passed 100 tests.
=== prop_valueSymmetric from KalahaTest.hs:51 ===
+++ OK, passed 100 tests.
=== prop_movesSymmetric from KalahaTest.hs:58 ===
+++ OK, passed 100 tests.
=== prop_moveSymmetric from KalahaTest.hs:63 ===
+++ OK, passed 100 tests.
=== prop_moveDoesntChangeSum from KalahaTest.hs:74 ===
+++ OK, passed 100 tests.
=== prop_specificMoves from KalahaTest.hs:83 ===
+++ OK, passed 1 tests.
True

*Main> :l GameStrategiesTest
[1 of 2] Compiling Kalaha           ( Kalaha.lhs, interpreted )
[2 of 2] Compiling Main             ( GameStrategiesTest.hs, interpreted )
Ok, modules loaded: Main, Kalaha.
*Main> main
=== prop_minimaxPicksWinningStrategyForNim from GameStrategiesTest.hs:42 ===
+++ OK, passed 100 tests.
=== prop_alphabetaSolutionShouldEqualMinimaxSolution from GameStrategiesTest.hs:46 ===
+++ OK, passed 100 tests.
=== prop_boundedAlphabetaIsOptimalForNim from GameStrategiesTest.hs:50 ===
+++ OK, passed 100 tests.
=== prop_pruningShouldBeDoneCorrectlyForStaticTestTrees from GameStrategiesTest.hs:56
    ===
+++ OK, passed 1 tests.
True
```