DM552: Programming Languages

Project 2 Report

Functional Programming

Torvald Johnson

Table of Contents

Implementation:

The main purpose of this project, was to utilize the functional programming paradigm and the Haskell language to build a working implementation of the Siege Game as outlined in the project description. Furthermore, the second half of the project required the implementation of functions to create a "game tree" to represent all possible moves taken by players in any game, and a "minimax" algorithm which would decide on the next move after inspecting the game tree.

In part one, the implementation of the Siege Game, task 1.1 was to create a placeDefenders function. This was quite simple, the function just inserts a given key (the coordinates) and a value (the player) into a map (the game board) of key-value pairs, using the M.insert function of the Data.Map package.

Task 1.2 was very similar to task 1.1. To move a player from a given vertex to another vertex, all I had to do was lookup what player was on the original vertex, delete them from the original vertex, and insert them into the new vertex.

To complete task 1.3, I made an auxiliary function "captureHelper" that accepts a vertex for the defender's current location and a vertex for the attacker's current location, and then returns a vertex representing the location the defender will hop to after taking the attacker. Using this helper function, the captureMove function is able to recursively loop through the list of attackers to be taken, deleting the attackers and the original defender location from the map, and inserting the defender at the new location. I was also able to reuse the "captureHelper" function to help complete task 1.8.

Task 1.4 simply required me to make the connections between the algebraic data types and their matching functions.

For task 1.5 I created 2 auxiliary functions to help out. The function "getGameVertices" creates a list of all useable coordinates in the game board, and "inhabitedVertices" accepts this list of coordinates, and produces a list of vertices to be inhabited by attackers at the start of the game. Using this, it was a simple matter to replicate a list of "True" the same length as the list of inhabited vertices, and then zip the two lists together to create the game board at a game's initialization.

For task 1.6 I created the function "placeFirstDefender" to first find an empty vertex on the game board, and then call my other function "placeSecondDefender". This function finds another empty vertex, and then adds both vertexes to a list of "PlaceDefenders" SiegeMoves. It then recursively loops to find the next empty vertex for the second defender piece, until all possibilities are exhausted. The function then loops through the next empty vertex for the "placeFirstDefender"function, and continues on, until it has produced a list of all possible "PlaceDefenders" moves.

To accomplish task 1.7, I first created a helper function "findPlayerPiece" which recursively loops through every vertex on the board, until it finds a coordinate inhabited by the specified player. It then calls the "findEmptyNeighbor" function I created, to loop through the neighbors of that vertex, checking each one to see if it is empty. If it finds an empty neighbor vertex, it will then if the specified player is an attacker, check if attackers are allowed to move to that vertex. Then, it will add a "SimpleMove" and the

player's current vertex, and the empty neighbor vertex, to a list of "SimpleMove"s, and continue checking until all possibilities of player locations and their neighbors are exhausted.

Task 1.8 was pretty tricky. First I created a function "defenderCaptureMovesFindDefender" to recursively loop through every location on the board until it finds a vertex with a defender piece. It then calls the "defenderCaptureMovesFindAttackerNeighbor" function I created, to recursively loop through the neighbors of the defender piece, until it finds one containing an attacker. Then, using the "captureHelper" function I created for task 1.3, it checks to see if the vertex the defender would move to is empty. It then also checks to see if that vertex is, in fact, a neighbor of the attacker, ensuring that this piece is on the board. If all of these are true, it adds a "CaptureMove" to a list of CaptureMoves, along with the vertex containing the defender, and another list containing the attacker vertex, plus the result of my helper function "defenderCaptureMovesFindNextAttacker". This function recursively loops through the neighbors of the new defender location (on a board where the old defender location, and the attacker who was taken, have both been deleted) to see if there are any more attacker neighbors. If there are, it will once again use the "captureHelper" function to check if the vertex the defender would move to is empty, and if that vertex is a neighbor of the attacker. If all of these are true, the attacker vertex is added to a list of vertexes, and defenderCaptureMovesFindNextAttacker is called again to check the neighbors of the new defender location. This continues until all possible options are exhausted, and returns a list of possible sequences the defender player could choose for capturing attackers.

Task 1.9 recursively loops through a given game board with the "boardString" function, checking every vertex and building a string to represent the board and the locations of pieces. If the vertex is not within the specified game board, an empty space is added to the string. If a vertex is within the "goal" area of the board, "goalString" is called, which adds a "*" symbol to the string if the vertex is empty, an "A" to the string if the vertex contains an attacker, and a "D" to the string if the vertex contains a defender. If the vertex is not in the goal area of the board and is not outside of the board, "restString" is called, which adds a "." symbol to the string if the vertex is empty, an "a" to the string if the vertex contains an attacker, and a "d" to the string if the vertex contains a defender. "formatHelper" is called to check every vertex, adding a "\n" to create a new line, every time the board reaches a 3 in the x-value.

Task 2.1 to create a game tree was pretty tricky.

Unfortunately I don't have time to write about part 2, and I did not have time to finish the minimax functions. I did finish part 2.1 and 2.2. I worked as hard as I could but ran out of time, and now I have to upload my project.

Some example executions:

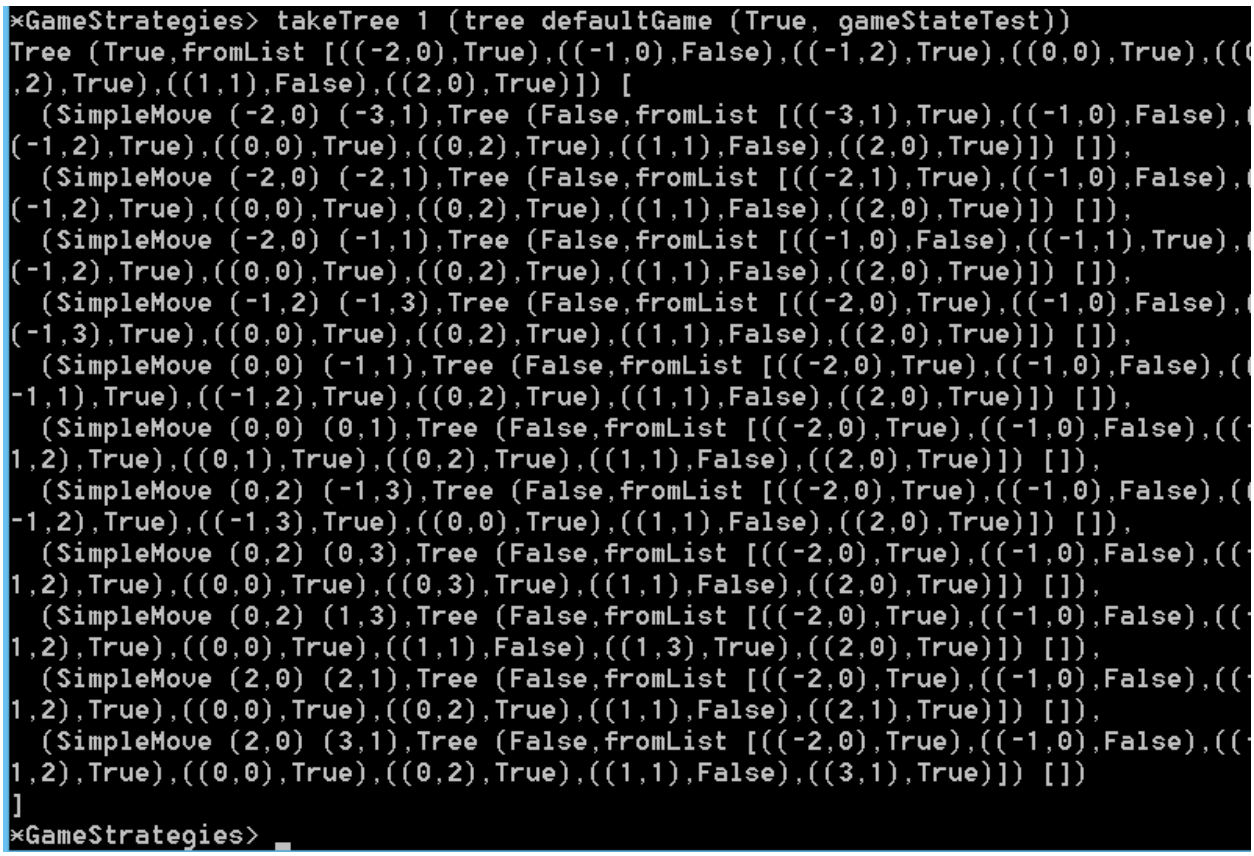
```
*SiegeGameImpl> defenderCaptureMoves defaultGame gameStateTest
[CaptureMove (1,1) [(0,0)],CaptureMove (1,1) [(0,2),(-1,2),(-2,0)],CaptureMove (
1,1) [(2,0)],CaptureMove (-1,0) [(-2,0)],CaptureMove (-1,0) [(0,0)]]
*SiegeGameImpl>
```

Execution of DefenderCaptureMoves on a test game state I created

```
    Tab character found here, and in 22 further locations.
    Please use spaces instead.
Ok, modules loaded: SiegeGameImpl, Game (Game.o), SiegeGame (SiegeGame.o), Siege
GameScore1 (SiegeGameScore1.o).
*SiegeGameImpl> putStrLn (showGameImpl defaultGame gameStateTest)
  ***
  AA*
..**D..
.ada.a.
.......
  ...
  ...

*SiegeGameImpl>
```

Execution of showGameImpl on a test game state I created



Execution of takeTree 1 on a test game state I created

Full program listing:

SiegeGameImpl.hs

{-# LANGUAGE TypeFamilies,FlexibleContexts #-}

module SiegeGameImpl where

import Game

import SiegeGame

import SiegeGameScore1

import Data.Map (Map)

import qualified Data.Map as M

import Data.Maybe

import Data.List (tails, partition)

```
data ScoreAlg = Score0 | Score1 deriving (Show,Read)

getGame :: ScoreAlg -> SiegeGame

getGame Score0 = defaultGame

getGame Score1 = case defaultGame of SiegeGame g _ -> SiegeGame g (score1 g)


---


-- Task 1.1)

placeDefenders :: [V] -> Map V Player -> Map V Player

placeDefenders [] s = s

placeDefenders (v:vs) s = placeDefenders vs (M.insert v False s)


-- Task 1.2)

simpleMove :: V -> V -> Map V Player -> Map V Player

simpleMove v0 v1 s

   | M.lookup v0 s == Nothing = s

         | otherwise = M.insert v1 p (M.delete v0 s) where Just p = M.lookup v0 s



-- Task 1.3)

captureMove :: V -> [V] -> Map V Player -> Map V Player

captureMove v [] s = s

captureMove v (x:xs) s

   | M.lookup v s == Just False = captureMove v (x:xs) (M.delete v s)

   | otherwise = captureMove c xs (M.insert c False (M.delete x s)) where c = captureHelper v x


captureHelper :: (V) -> (V) -> (V)

captureHelper (x,y) (a,b) = (2*a - x, 2*b - y)
```

```
-- Task 1.4)
moveImpl :: SiegeMove -> Map V Player -> Map V Player
moveImpl (PlaceDefenders vs) s = placeDefenders vs s
moveImpl (SimpleMove v0 v1) s = simpleMove v0 v1 s
moveImpl (CaptureMove v0 v1) s = captureMove v0 v1 s

-- Task 1.5)
startStateImpl :: SiegeGame -> Map V Player
startStateImpl sg = M.fromList (zip vs ps)
  where
    vs = inhabitedVertices(getGameVertices sg)
    ps = replicate (length (getGameVertices defaultGame)) True

inhabitedVertices :: [V] -> [V]
inhabitedVertices [] = []
inhabitedVertices (x:xs)
  | inG x = inhabitedVertices xs
        | otherwise = (x) : inhabitedVertices xs

getGameVertices :: SiegeGame -> [V]
getGameVertices (SiegeGame g _) = [ v | (v, _) <- M.toList g ]

-- Task 1.6)
placeDefenderMoves :: SiegeGame -> [SiegeMove]
placeDefenderMoves sg = placeFirstDefender (startStateImpl sg) (getGameVertices sg) []

placeFirstDefender :: Map V Player -> [V] -> [SiegeMove] -> [SiegeMove]
placeFirstDefender s [] ml = ml
```

```
placeFirstDefender s (v:vs) ml
   | M.lookup v s /= Just True = placeFirstDefender s vs (placeSecondDefender v vs s ml)
   | otherwise = placeFirstDefender s vs ml

placeSecondDefender :: V -> [V] -> Map V Player -> [SiegeMove] -> [SiegeMove]
placeSecondDefender v [] s ml = ml
placeSecondDefender v (x:xs) s ml
   | M.lookup x s /= Just True && x /= v = placeSecondDefender v xs s (ml ++ [PlaceDefenders[v,x]])
   | otherwise = placeSecondDefender v xs s ml

-- Task 1.7)
simpleMoves :: SiegeGame -> Player -> Map V Player -> [SiegeMove]
simpleMoves sg@(SiegeGame g _) p s = findPlayerPiece (getGameVertices sg) p g s []

findPlayerPiece :: [V] -> Player -> Map V VertexInfo -> Map V Player -> [SiegeMove] -> [SiegeMove]
findPlayerPiece [] p g s sm = sm
findPlayerPiece (v:vs) p g s sm
   | M.lookup v s == Just p = findPlayerPiece vs p g s (findEmptyNeighbor v p (neighbors (fromJust
(M.lookup v g))) s sm)
   | otherwise = findPlayerPiece vs p g s sm

findEmptyNeighbor :: V -> Player -> [NeighborInfo] -> Map V Player -> [SiegeMove] -> [SiegeMove]
findEmptyNeighbor v p [] s sm = sm
findEmptyNeighbor v p (i:is) s sm
   | p == True && a == True && M.lookup n s == Nothing = findEmptyNeighbor v p is s (sm ++
[SimpleMove v n])
        | p /= True && M.lookup n s == Nothing = findEmptyNeighbor v p is s (sm ++ [SimpleMove v
n])
        | otherwise = findEmptyNeighbor v p is s sm
        where
```

```
      n = getNeighbor i

      a = isAttackerAllowed i


-- Task 1.8)

defenderCaptureMoves :: SiegeGame -> Map V Player -> [SiegeMove]

defenderCaptureMoves sg@(SiegeGame g _) s = defenderCaptureMovesFindDefender
(getGameVertices sg) g s []


defenderCaptureMovesFindDefender :: [V] -> Map V VertexInfo -> Map V Player -> [SiegeMove] ->
[SiegeMove]

defenderCaptureMovesFindDefender [] g s sm = sm

defenderCaptureMovesFindDefender (v:vs) g s sm

   | M.lookup v s == Just False = defenderCaptureMovesFindDefender vs g s sm ++
(defenderCaptureMovesFindAttackerNeighbor v (lookupNeighborInfoList v g) g s [])

   | otherwise = defenderCaptureMovesFindDefender vs g s sm


defenderCaptureMovesFindAttackerNeighbor ::  V -> [NeighborInfo] -> Map V VertexInfo -> Map V
Player -> [SiegeMove] -> [SiegeMove]

defenderCaptureMovesFindAttackerNeighbor v [] g s sm = sm

defenderCaptureMovesFindAttackerNeighbor v@(x,y) (i:is) g s sm

   | M.lookup n s == Just True && M.lookup e s == Nothing && elem e (map
getNeighbor(lookupNeighborInfoList n g)) = defenderCaptureMovesFindAttackerNeighbor v is g s (sm
++ [CaptureMove v ([n]++(defenderCaptureMovesFindNextAttacker e (lookupNeighborInfoList n g) g
(M.delete v (M.delete n s)) []))])

         | otherwise = defenderCaptureMovesFindAttackerNeighbor v is g s sm

         where

      n@(a,b) = getNeighbor i

      e = captureHelper v n


defenderCaptureMovesFindNextAttacker ::  V -> [NeighborInfo] -> Map V VertexInfo -> Map V Player -
> [V] -> [V]

defenderCaptureMovesFindNextAttacker v [] g s sm = sm
```

```
defenderCaptureMovesFindNextAttacker v@(x,y) (i:is) g s sm
   | M.lookup n s == Just True && M.lookup e s == Nothing && elem e (map
getNeighbor(lookupNeighborInfoList n g)) = sm ++ [n] ++ (defenderCaptureMovesFindNextAttacker e
(lookupNeighborInfoList e g) g (M.delete v (M.delete n s)) [])
          | otherwise = defenderCaptureMovesFindNextAttacker (x,y) is g s sm
          where
      n@(a,b) = getNeighbor i
      e = captureHelper v n

lookupNeighborInfoList :: V -> Map V VertexInfo -> [NeighborInfo]
lookupNeighborInfoList v g = neighbors (fromJust (M.lookup v g))

-- Task 1.9)
showGameImpl :: SiegeGame -> Map V Player -> String
showGameImpl (SiegeGame g _) m = boardString [(x, y) | y <- reverse [-3..3], x <- [-3..3]] ""
   where
      boardString :: [V] -> String -> String
      boardString [] s = s
      boardString (v:vs) s
         | not (inV v) = " " ++ formatHelper (fst v) vs s
         | isGoal (fromJust (M.lookup v g)) = goalString v ++ formatHelper (fst v) vs s
         | otherwise = restString v ++ formatHelper (fst v) vs s
      formatHelper i l s
         | i == 3 = ("\n" ++ boardString l s)
         | otherwise  = boardString l s
      goalString v
         | M.lookup v m == Nothing = "*"
         | M.lookup v m == Just True = "A"
         | otherwise = "D"
```

```
      restString v
        | M.lookup v m == Nothing = "."
        | M.lookup v m == Just True = "a"
        | otherwise = "d"

---

defenderMoves :: SiegeGame -> Map V Player -> [SiegeMove]
defenderMoves sg s = case [v | (v, False) <- M.toList s] of
  [] -> placeDefenderMoves sg
  _  -> case defenderCaptureMoves sg s of
    [] -> simpleMoves sg False s
    xs -> xs

movesImpl :: SiegeGame -> Player -> Map V Player -> [SiegeMove]
movesImpl sg True s    = simpleMoves sg True s
movesImpl sg False s   = defenderMoves sg s

valueImpl :: SiegeGame -> Player -> Map V Player -> Double
valueImpl sg p m | null $ movesImpl sg p m = if p then -infinity else infinity
valueImpl (SiegeGame g _) _ m | and [ M.lookup v m == Just True | (v,vi) <- M.toList g, isGoal vi ] =
infinity
valueImpl (SiegeGame g s) p m = s m


instance Game SiegeGame where
  type GameState SiegeGame  = Map V Player
  type Move SiegeGame       = SiegeMove
```

```
 startState    = startStateImpl
 showGame      = showGameImpl
 move _ _ s m  = moveImpl m s
 moves         = movesImpl
 value         = valueImpl
```

GameStrategies.hs

```
{-# LANGUAGE FlexibleContexts                #-}
module GameStrategies where
import Game
import Data.List (intercalate)

import SiegeGameImpl
import SiegeGame

data Tree m v   = Tree v [(m,Tree m v)]
type GTree g    = Tree (Move g) (Player,GameState g)

type AlphaBeta = (Value,Value)
type Depth = Int

instance (Show m, Show v) => Show (Tree m v) where
 show (Tree x xs) = "Tree " ++ show x ++ " ["++ sub' ++"]"
   where
    sub = intercalate ",\n" (map show xs)
    sub' = if null sub
           then ""
           else "\n" ++ unlines (map ("  " ++ ) $ lines sub)
```

```
startTree :: Game g => g -> Player -> GTree g
startTree g p = tree g (p, startState g)

-- Task 2.1)
tree :: Game g => g -> (Player, GameState g) -> GTree g
tree g ps@(p,s) = Tree ps (zip (moves g p s) (map (tree g) st)) where st = movedStatePlayerZip g p (generateMovedGameStateList g p s (moves g p s) [])

movedStatePlayerZip :: Game g => g -> Player -> [GameState g] -> [(Player, GameState g)]
movedStatePlayerZip g p gs = zip ps gs
  where
    ps = replicate (length gs) (not p)

generateMovedGameStateList :: Game g => g -> Player -> GameState g -> [Move g] -> [GameState g] -> [GameState g]
generateMovedGameStateList g p s [] gs = gs
generateMovedGameStateList g p s (x:xs) gs = generateMovedGameStateList g p s xs (gs ++ [(move g p s x)])

-- Task 2.2)
takeTree :: Depth -> Tree m v -> Tree m v
takeTree 0 (Tree v _) = Tree v []
takeTree _ (Tree v []) = Tree v []
takeTree n (Tree v ch) =
  Tree v (map (foo (n - 1)) ch)
 where
  foo n (m, t) = (m, takeTree n t)
```

```
-- Task 2.3)

minimax :: Game g => g -> GTree g -> (Maybe (Move g), Value)

minimax = undefined


--minimax g t =


-- Task 2.4)

minimaxAlphaBeta :: Game g => g -> AlphaBeta -> GTree g -> (Maybe (Move g), Value)

minimaxAlphaBeta = undefined
```