Programming and Reasoning Assignment

Stepan Sindelar

This report describes the implementation of Reversi game, also called Othello. It consists of three sections. In the first one, the general data structures and functions are discussed. The second one provides an overview of the AI algorithm used. And in the last one, the approach to build the user interface is described.

1 General Data Structures and Algorithms

1.1 Representation of the Game Board

To represent the game board, the `Array` data type from Haskell Array library is used. However, the game board is defined as a new type `GameBoard` and all the operations on it are performed through "access" functions such as `get` function. This way, the game field is encapsulated and the internal representation may be changed without affecting the other code.

The element type of the internal `GameField` array is `Maybe Piece` where `Piece` is a data type with two constructors: `Head` and `Tail` representing two different players. Some 'access' functions use `Maybe Piece` (e.g., `get`) and some use just `Piece` (e.g., `push` – because it is not needed to 'delete' pieces from the game field). The index type of the array is tuple `(Int,Int)` so it is used as a classical two dimensional array (matrix).

Two special `Array` releated operators are used in the source code. The `!` operator queries the array. For example, the following expression returns the element under the index $(2, 3)$: `array!(2,3)`.

The `//` operator creates a new array with just one cell alternated. (Because arrays, at least with the standard implementation, are immutable, it has to create a new array.) The following code snipped returns a new array with `Tail` under the index $(3, 4)$: `array // [((x,y),Tail)]`

1.2 Finding valid moves

The most complex general operation on `GameField` type is the `getPossibleMovesWithEval` function. It searches for all the possible positions given player can play at. Apart from these positions of type `(Int,Int)`, it also returns useful information about the eventual lines that would be flipped if the player actually played at some specific position: namely the length of such line and the position where this line starts.

The search is performed as follows: firstly a list of all the positions of the pieces of given color is obtained by simple search through the whole field. Then all 8 directions from each of these positions are inspected for a possible line.

2 AI Algorithm

The AI algorithm that is used is based upon the Minimax algorithm with alpha-beta prunning. This algorithm searches the game tree that is a tree with all possible game situations envolving from the current game situation. When designing a Minimax implementation one has to consider two important things: when to stop the recursion so that the algorithm does run in some sensible time and if the recursion is stopped in a situation when the game is not over yet, how to evaluate that game situation. Aside from these, it is also important to provide a goods heuristic function, which increases the chances of alpha-beta prunning, but it should run fast so it does not affect the overall performance.

2.1 Deciding When to Stop the Recursion

One of the common approaches is to limit the deepth of the searched game tree. However, in Reversi, the number of valid moves may be very different in different game situations. Sometimes there is no valid move at all, sometimes ten valid moves are available. For this reasons an integer 'time quantum' is used: the minimax function takes a 'time quantum' as an argument and it divides it by the count of valid moves (that is the number of branches in the game tree) and this value is used as a 'time quantum' for all the recursive calls (for all the branches). When the 'time quantum' reaches 0, the recursion is stopped and the current game situation is evaluated. The recursion might stop before 'time quantum' is 0 that happens when the end of the game is reached and also the allocated 'time quantum' might not be fully utilised because of alpha-beta prunning. In such case it would make sense to 'return' the unused 'time quantum' to the caller. However, for the sake of simplicity of the implementation, this enhancement was not implemented. Therefore the 'time quantum' argument is actually used as an upper bound.

2.2 Evaluation function

The evaluation function is generally based on the difference between number of pieces that each player has on the board. Corner and edge pieces are more prefered, so they are counted as 4 and 2 normal pieces respectively. Moreover, to give slightly more preference to game situations where the opponent does not have many valid moves, the difference between number of valid moves is added to the result. If the evaluated game situation captures a situation when the game is over, only the difference between number of pieces is taken into account and is multiplyed by a large constant.

2.3 Heuristic

As a heuristic, simply a sum of lengths of all the lines that would be flipped if the player played at given position is used.

2.4 Performance

The implementation has very different performance when compiled using ghc with all safe optimalisations and when interpreted using ghci. When compiled, the minimax function can run in very short time when 'time quantum' is of size 7000 which very roughly corresponds to a game tree of deepth 5–6 if we assume 5 valid moves at each level. In practise, the deepth was observed to be even greater.

3 User Interface

The user interface is built as a console application using standard IO system in Haskell, namely the IO monad type. In practise, it means that all the functions that make use of input or output must return special `IO` type and all the IO operations must be sequenced using operator `>>=` or, in a more convenient way, using the `do` notation. The `IO` type wraps actual return type, which often happens to be simply empty tuple `IO ()`. More information about IO system in Haskell can be found in TODO.

3.1 ANSI Console Commands

To provide a good looking interface, standard ASI console commands are used. These commands are just sequences of characters "printed" on the console, but the console actually treats them as special commands that can, for example, change the font color or move the cursor to a different position.

Fortunatelly it is not needed to write these special sequences by hand, because there is a haskell package called `System.Console.ANSI` which provides a wrapper around them. This package provides two types of functions: pure functions that return the control sequence as a string and functions that operate on `IO` type, in other words, they directly print the commands. I used only the pure functions. In the following example the function `show` returns a text that, if printed, is red.

```
show :: String
show =
  setSGRCode [ SetColor Foreground Vivid Red ] ++
  "This text will be red when printed" ++
  setSGRCode [ SetColor Foreground Vivid White ] -- to switch the color back
```

Because console on Microsoft Windows operating systems does not support ANSI console commands correctly, the resulting program does not work on Windows. It was, however, succesfully tested on Linux and with cygwin it should work on windows as well.

3.2 Main Control 'Loop'

The function that implemnts the game 'loop' is called `game` and it has the following type.

```
game :: GameField -> GamePlayer -> GamePlayer -> IO ()
```

It takes the current game field and two data structures of type `GamePlayer`. The most important field of this structure is a function `play` that takes `GamePlayer` as an argument and returns `IO Pos` which is a position at which the player wants to play. With this we can create a `GamePlayer` that uses minimax as its play function or we can implement another play function that asks the user. The function `game` procceds as follows: it gets the position from the first player, creates new updated game field, and recursively calls itself with the new game field and with the two players swapped.

3.3 Usage

The file `Reversi.hs` contains a parameter-less `main` function so that it can be compiled into a console program using ghc. It also provides function `playReversi` that accepts the desired size of the game field as a first argument (type `(Int,Int)`) and a type of piece which will start the game. Computer always has heads (red) and user tails (blue). The program itself then prints a textual help on how to procced.