Project Chess in Haskell

By: Fredrik Sandelin, Erik Sundberg and Viktor Vilmusenaho

Program Design and Data Structures, 2014/2015

Table of contents

1. Introduction

To expand our knowledge of the functional programming language Haskell, we decided to make a playable chess game with some very simple AI.

Writing games in Haskell can be quite challenging, as functional languages tend to be somewhat “anti-state”. It is of great importance to keep track of the intermediate states when making games, and we wanted to tackle this challenge, while maintaining an (almost) purely functional code.

The initial idea was to simulate some kind of chess board by creating our own data type in haskell, and "store" the chess pieces within it. These pieces will have to have different restrictions to how and where they can move, just like in regular chess. Thus we would have to make these chess pieces and store their specific properties somehow. We thought this could become somewhat of a challenge.

Our plan involved making a computer opponent as well. However, we were aware that making an AI can be very complicated depending on how advanced you want to make it. Introducing some kind of random element was our obvious, the only question was how "smart" of an AI we would have time to make.

We also decided to make the program able to validate if any player was in check, and give that information to the user.

During the project, several challenges were encountered. The most common problem was that while changing the structures of data types during the project, the code resulted in giving errors.

The functions that used these data types that were modified had to be remade, which was quite time consuming (and confusing).

The greatest challenge with this project was trying to understand code written by others. While the code was quite easy to read, difficulties came up when functions written by others had to be modified, as the whole program must be taken into account when modifying even small parts of the code.

2. Summary

The program will let the user carry out a standard game of chess. The user will specify what moves to make, and the program will draw a chess board with the new positions of the chess pieces when a move has been made. Our code will also verify that the moves are valid, so that no cheating is possible.

Our initial thought on how to make the program user friendly, was making a chess board with ascii symbols, making it easy for the user to play the actual game. This resulted in us having to represent the chessboard as a sorted list, as opposed to a coordinate system which would mean smaller time complexity for the program. If the program encounters a piece on a specific coordinate, it will print the ascii symbol representing that piece, and an empty square otherwise.

We implemented a function that will check for valid moves for each move the user is trying to make. This will confirm that all the moves that are made are valid according to rules of chess.

As for the user interaction with the program, we made a main function that involved I/O. This function takes an user input and checks that the input is valid (through the **validInput** function). The function will then send the input to the **validMove** function, to assure that the input will result in a valid move. When all the necessary checks have been made, the program will print the new chess board, with that particular piece moved to its new position.

3. Use cases

This section describes how to use the actual program. It will also cover what software you will need to use it.

3.1 Software requirements

To use the program, you will need to install the ghc compiler for Haskell. When installed, navigate in the terminal to the place where the module “chessgame.hs” is located, type “ghci”, and press Enter. When the compiler has loaded, write “:l chessgame.hs” into the terminal.

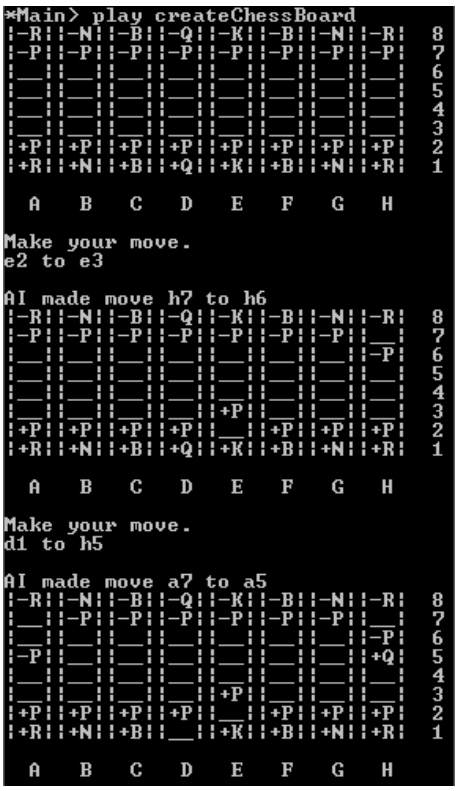
3.2 User Examples / How to play the game

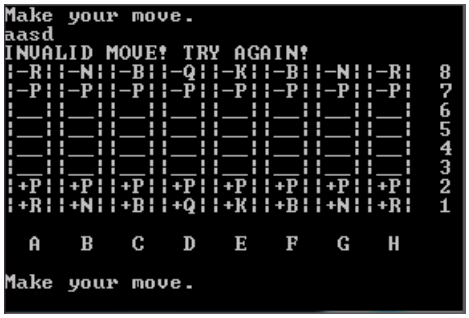
To the right is an example of how you play the game. To initiate the gameloop you write “play” and then “createChessBoard” on the same line. This will print out a new chessboard with all the pieces in their basic position.

It will ask you to make a move. By typing in the format of “a1 to b2” the game will be able to understand you and try to make your requested move. To the right you see an example move of “e2 to e3” which moved the white pawn up one space.

The game loop then tells you what move the computer made, prints out the new chessboard and asks you to make a new move.

The separate picture is an example of an invalid move. When the text “aasd” is used, the game loop replies with “INVALID MOVE! TRY AGAIN!”, prints out the current chessboard without any difference and asks you to make a new move.





4. Program documentation

4.1 Description of data structures

To make a chess game we obviously need some sort of data representation to be able to track the current state of a game, if a move is valid, if a king is checked and similar chess related things.

The first challenge was:
How do we represent the chessboard, chess pieces and where on the chessboard the chess pieces are standing. We started by creating a data types:
data ChessPiece = Piece PieceRank Colour | NoPiece
data PieceRank = King | Queen | Rook | Bishop | Knight | Pawn
data Colour = Black | White

Now every chess piece would have a rank and a colour and and an empty chess piece would be represented by NoPiece. We then thought it would be a good idea to make the board a list of chess piece (**[ChessPiece]**) where the list would always be of length 64 and the first element would represent the upper left square, the second element the square to the right of it and so on. Empty squares would be represented by a **NoPiece**. We thought this would make printing easy and we thought that using modular arithmetic to solve if a move is valid wouldn't be too hard.

After talking to supervisor:
Our supervisor told us that it would probably be easier to use some sort of coordinate system since validating moves would be harder than displaying the chessboard. So we changed the data representations:
data ChessPiece = Piece PieceRank Colour Integer Integer
data PieceRank = King | Queen | Rook | Bishop | Knight | Pawn
data Colour = Black | White
data ChessBoard = [ChessPiece]

Now the chessboard would only contain living pieces and every piece would have 2 **Integer** which would represent the coordinates (x,y) of which square the **ChessPiece** is standing on. To make printing easy we decided to always have the **ChessBoard** sorted from upper-left to lower right. This would essentially only mean that when we add and remove chess pieces from the chessboard, we will have to make sure that the list is sorted afterwards.

4.2 Description of important algorithms and functions

play

```
play :: ChessBoard -> IO()
play chessboard = do
    printChessBoard chessboard
    if getOutOfCheck White chessboard == Nothing && checkCheckKing White chessboard == Just False -- Not check but can't move = stalemate
        then putStrLn "\nIt is a stalemate."
        else do
    if getOutOfCheck White chessboard == Nothing && checkCheckKing White chessboard == Just True -- In check and can't move == Checkmate
        then putStrLn "\nYou lost."
        else do
    printOutCheckedKing chessboard
    putStrLn "\nMake your move."
    input <- getLine
    let newinput = integerFromInput input
    if input == "Save Game" || input == "save game" || input ==  "quit game" || input == "Quit Game"
        then if input == "quit game" || input == "Quit Game"
            then putStr "\nThanks for playing, Bye!\n"
            else print chessboard
        else do
    if inputValid input == True && (playChess chessboard newinput) /= chessboard && moveingWhite newinput chessboard
        then
            aiMove (playChess chessboard newinput) 4000
        else do
            putStr "INVALID MOVE! TRY AGAIN!\n"
            play chessboard
```

Basic description: The game loop takes one **ChessBoard** as an argument, which represents the current state of the game.

Step by step description:

1) The game-loop starts with displaying the current state of the game by printing the chessboard with **printChessBoard**.
2) The 2 following if statements checks if you are in a checkmate or stalemate with the help of **checkCheckKing** and **getOutOfCheck**.
3) **printOutCheckedKing** chessboard will print if either of the kings are in check.
4) The function will now ask you to make your move. It will then take input from the user.
5) The function now calls **integerFromInput** to make the user input meaningful to the program, by converting the input strings into integers.
6) If the input was either “Save Game”, “save game”, “Quit Game” or “quit game” it will either quit the game or print out the current chessboard so you can continue playing later by entering play *chessboard.*
7) The function checks if the input was a valid input with **inputValid**, if the move made a change to the board with **playChess** and if the piece that you tried to move was white with **moveingWhite**,if all checks holds it will call **aiMove** with the new chessboard together with an Integer.
8) If the input didn’t pass the checks at 7), then it was in some way invalid or illegal for the player to make and “INVALID MOVE! TRY AGAIN!” is displayed on the screen and the game loop is called again

.

aiMove

```
aiMove :: ChessBoard -> Integer -> IO()
aiMove chessboard n = do
        if n == 0 && (getOutOfCheck Black chessboard == Nothing)
            then if checkCheckKing Black chessboard == Just False
                then do
                    putStrLn "\nIt's a stalemate"
                    printChessBoard chessboard
                else do
                    putStrLn "\nYou won"
                    printChessBoard chessboard
            else do
        if n == 0
            then do
                let Just ((Piece arank acolour ax ay), (Piece brank bcolour bx by)) = getOutOfCheck Black chessboard
                putStr ("\nChuck made move " ++ [charFromIntgerLetter ax, charFromIntgerNumber ay, ' ', 't', 'o', ' ', charFromIntgerLetter bx, charFromIntgerNumber by] ++ "\n")
                printChessBoard (playChess chessboard (ax,ay,bx,by))
                play (playChess chessboard (ax,ay,bx,by))
            else do
        g <- newStdGen
        let (a, newGen) = randomR (1,8) g :: (Integer, StdGen)
        let (b, newGen2) = randomR (1,8) newGen :: (Integer, StdGen)
        let (c, newGen3) = randomR (1,8) newGen2 :: (Integer,StdGen)
        let (d, newGen4) = randomR (1,8) newGen3 :: (Integer,StdGen)
        if (colourOf(convertMaybe(pieceAtXY a b chessboard))) == Black && (playChess chessboard (a,b,c,d) /= chessboard)
            then do
                putStr ("\nChuck made move " ++ [charFromIntgerLetter a, charFromIntgerNumber b, ' ', 't', 'o', ' ', charFromIntgerLetter c, charFromIntgerNumber d] ++ "\n")
                play (playChess chessboard (a,b,c,d))
            else aiMove chessboard (n-1)
```

Basic description: The AI loop takes a **ChessBoard** which represents the current state of the game and an **Integer**. When the AI tries to make a move, it randomizes 4 Integer values between 1 and 8. If the 4 integers result in a valid move for Black on the Board, then that move will be made. The AI will try to do this the same number of times as the **Integer** argument. If the computer didn't find a random valid move when the limit is reached, then it checks if it is in checkmate or stalemate. If so, the game ends, otherwise he finds the quickest move so that he is not in check after he made that move.

Step by step Description:

1) The AI loop starts by checking if he is done trying to make the number of random moves that he was said out to make
    a) If he isn't done he will make a random seed and use it to randomize a number between 1 and 8 and a new random seed. 4 random numbers are generated, the numbers are called a,b,c and d.
    b) He will then check if the move those 4 number represents is made to a black piece by checking if **colourOf (convertMaybe (pieceAtXY** a b chessboard**)))** is **Black**. If it is **Black** and **playChess (** chessboard (a,b,c,d)**)** made a change to the board, then the AI has found a random move. (By checking is **playChess** made a change to the board, it is automatically checking if the move was valid). The random move is displayed by the help of **charFromIntgerLetter** & **charFromIntgerNumber** and the main game loop **play** is called with the new chessboard.

c) If the random move didn't pass the check, **aiMove** is called again with the same chessboard and with the a lesser **Integer** input (input - 1) than previously to make sure the function will eventually terminate.

2) If the **Integer** input == 0, which means the AI has tried the number of random moves we programmed it to try, he will check if **getOutOfCheck Black** chessboard == Nothing, which would mean that there is no possible move that the computer can make to not be in chess after the move is made. If that is the case, he will check if he is standing in check with **checkCheckKing Black** chessboard. If he is in check, he is checkmate and has lost, which is displayed and aiMove stops. If he isn't in check, it is a stalemate, which is displayed and aiMove stops.
3) **getOutOfCheck Black** chessboard does not return Nothing, it means that there for sure is a move that can be made to not be in check after the move. In that case the AI will use the two chesspieces returned by **getOutOfCheck Black** chessboard to make a move, and call the **play** function with the new chessboard.

validMove

```
validMove :: ChessPiece -> ChessPiece -> ChessBoard -> Bool
validMove (Piece rank colour x y) (Piece brank bcolour bx by) chessboard
    |x == bx && y == by = False
    |(bx > 8 || bx < 1) || (by > 8 || by < 1) = False
    |checkCheckKing colour (moveChessPiece (Piece rank colour x y) (Piece brank bcolour bx by) chessboard) == Just True = False
    |rank == King = kingCheck (Piece rank colour x y) (Piece brank bcolour bx by) chessboard
    |rank == Queen = queenCheck (Piece rank colour x y) (Piece brank bcolour bx by) chessboard
    |rank == Rook = rookCheck (Piece rank colour x y) (Piece brank bcolour bx by) chessboard
    |rank == Bishop = bishopCheck (Piece rank colour x y) (Piece brank bcolour bx by) chessboard
    |rank == Knight = knightCheck (Piece rank colour x y) (Piece brank bcolour bx by) chessboard
    |rank == Pawn = pawnCheck (Piece rank colour x y) (Piece brank bcolour bx by) chessboard
```

Basic description: The **validMove** function takes two **ChessPieces** and a **ChessBoard**. The first **ChessPiece** is the piece to be moved and the second piece is the new spot for the piece. This function will reuturn True if the move is approved by the corresponding ***"Piece"*Check** function.

Step by step description:

1) The first guard of the function compares the coordinates of the C**hessPieces**. If the coordinates are the same it means that the player tried to move a piece to the same spot, which is not allowed and the function returns False.
2) The second guard checks the coordinates of the target destination. If they are below 1 or above 8 the move is invalid, and the function returns False.
3) The third guard runs the function **checkCheckKing** with the colour of the piece to be moved and the result of the function **"moveChessPiece"** with the 2 pieces and the **ChessBoard** as arguments. This is our check to make sure our player doesn't stand in check after his move.

4) Next is six similar guards. Only one will be run each call of the function depending on the rank of the piece to be moved. The function for the corresponding piece will be run with the two **chessPieces** and the **ChessBoard**. If this function returns **True** then the **validMove** function will return **True**.

queenCheck

```
queenCheck :: ChessPiece -> ChessPiece -> ChessBoard -> Bool
queenCheck (Piece rank colour x y) (Piece brank bcolour bx by) chessboard
    |(pieceAtXY bx by chessboard /= Nothing) && colour == ccolour = False
    |x == bx = straight (Piece rank colour x y) (Piece brank bcolour bx by) chessboard -- updown
    |y == by = straight (Piece rank colour x y) (Piece brank bcolour bx by) chessboard -- rightleft
    |(bx > x && by > y) && ((by - y) == (bx - x)) = diagonal (Piece rank colour x y) (Piece brank bcolour bx by) chessboard -- upright
    |(bx < x && by > y) && ((by - y) == (x - bx)) = diagonal (Piece rank colour x y) (Piece brank bcolour bx by) chessboard  -- upleft
    |(bx < x && by < y) && ((y - by) == (x - bx)) = diagonal (Piece rank colour x y) (Piece brank bcolour bx by) chessboard  -- downleft
    |(bx > x && by < y) && ((y - by) == (bx - x)) = diagonal (Piece rank colour x y) (Piece brank bcolour bx by) chessboard  -- downright
    |otherwise = False
```

Basic description: The **validMove** function will, for each unique **ChessPiece**, call a function that will determine if the desired move is valid for that particular chess piece. As illustrated above, the function **queenCheck** will return a boolean value depending on if the queen can be moved from one square to the other on the chessboard (according to normal chess rules, of course). Similar functions exist for each of the six unique ranks of chess pieces.

Step by step description:

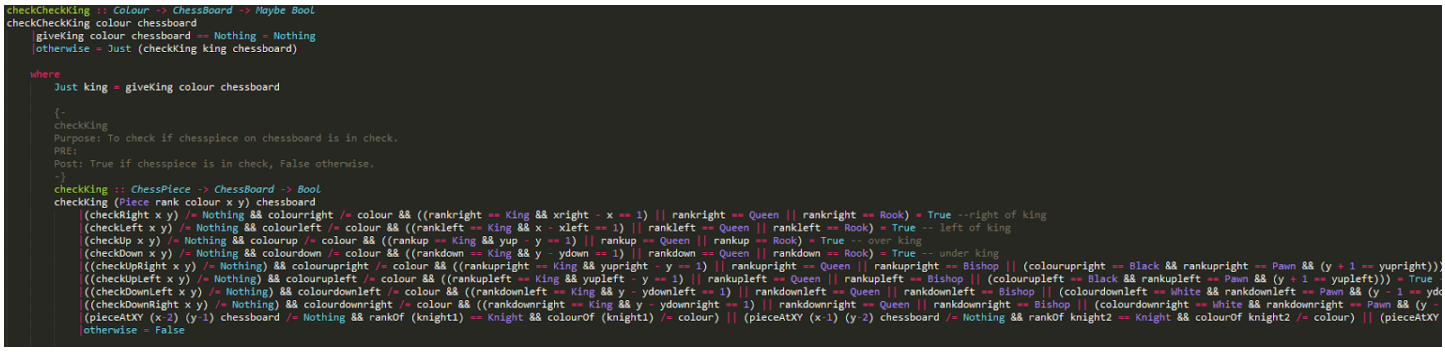
1) If the player already has one of its chess pieces at the desired target destination, the function will return False - you cannot move your queen there.
2) If the initial and target x coordinates are equal, or if the same applies to the y-coordinates, the function **straight** will be called, checking if the move is possible.
3) If none of the initial x and y coordinates are equal to the destination coordinates, the function **diagonal** will be called, checking if the diagonal move is possible.

inputValid

```
inputValid :: String -> Bool
inputValid [a,b,' ','t','o',' ',c,d]
    |integerFromChar(a) /= 0 && integerFromChar(b) /= 0 && integerFromChar(c) /= 0 && integerFromChar(d) /= 0 = True
    |otherwise = False
inputValid s = False
```

Basic description: This function is simple but important, because it refuses to take any input that the program cannot handle. The function **integerFromChar** will give the user input values of range 1 to 8. If the input is incorrect (does not make a valid chess move) **integerFromChar** will return 0, which is interpreted by **inputValid** as an invalid input.

checkCheckKing



```haskell
checkCheckKing :: Colour -> ChessBoard -> Maybe Bool
checkCheckKing colour chessboard
    |giveKing colour chessboard == Nothing = Nothing
    |otherwise = Just (checkKing king chessboard)

    where
        Just king = giveKing colour chessboard

        {-
        checkKing
        Purpose: To check if chesspiece on chessboard is in check.
        PRE:
        Post: True if chesspiece is in check, False otherwise.
        -}
        checkKing :: ChessPiece -> ChessBoard -> Bool
        checkKing (Piece rank colour x y) chessboard
            |(checkRight x y) /= Nothing && colourright /= colour && ((rankright == King && xright - x == 1) || rankright == Queen || rankright == Rook) = True --right of king
            |(checkLeft x y) /= Nothing && colourleft /= colour && ((rankleft == King && x - xleft == 1) || rankleft == Queen || rankleft == Rook) = True -- left of king
            |(checkUp x y) /= Nothing && colourup /= colour && ((rankup == King && yup - y == 1) || rankup == Queen || rankup == Rook) = True -- over king
            |(checkDown x y) /= Nothing && colourdown /= colour && ((rankdown == King && y - ydown == 1) || rankdown == Queen || rankdown == Rook) = True -- under king
            |((checkUpRight x y) /= Nothing) && colourupright /= colour && ((rankupright == King && yupright - y == 1) || rankupright == Queen || rankupright == Bishop || (colourupright == Black && rankupright == Pawn && (y + 1 == yupright)))
            |((checkUpLeft x y) /= Nothing) && colourupleft /= colour && ((rankupleft == King && yupleft - y == 1) || rankupleft == Queen || rankupleft == Bishop || (colourupleft == Black && rankupleft == Pawn && (y + 1 == yupleft))) = True
            |((checkDownLeft x y) /= Nothing) && colourdownleft /= colour && ((rankdownleft == King && y - ydownleft == 1) || rankdownleft == Queen || rankdownleft == Bishop || (colourdownleft == White && rankdownleft == Pawn && (y - 1 == ydo
            |((checkDownRight x y) /= Nothing) && colourdownright /= colour && ((rankdownright == King && y - ydownright == 1) || rankdownright == Queen || rankdownright == Bishop || (colourdownright == White && rankdownright == Pawn && (y -
            |(pieceAtXY (x-2) (y-1) chessboard /= Nothing && rankOf (knight1) == Knight && colourOf (knight1) /= colour) || (pieceAtXY (x-1) (y-2) chessboard /= Nothing && rankOf knight2 == Knight && colourOf knight2 /= colour) || (pieceAtXY
            |otherwise = False
```

Basic description: **checkCheckKing** takes a **Colour** and a **ChessBoard** and returns Just True if the coloured king stands in check, Just False if the coloured king isn't standing in check and Nothing is there is no king of the specified colour.

checkCheckKing was previously a short function that took a long time to process, so we rewrote it, resulting in a lot more lines of code, but also a function that would make our AI run 100 times faster. This was important since our AI works by randomizing a move up to 4000 times and every time he randomizes a move, he has to run **checkCheckKing**.
Basically the previously short function tried to move every single piece on the board to the kings location and if one of those moves was a valid move according to **validMove** then he would be standing in check. The new function works by scanning the surroundings of the king and checking if there is a **ChessPiece** that can kill the king on those squares.

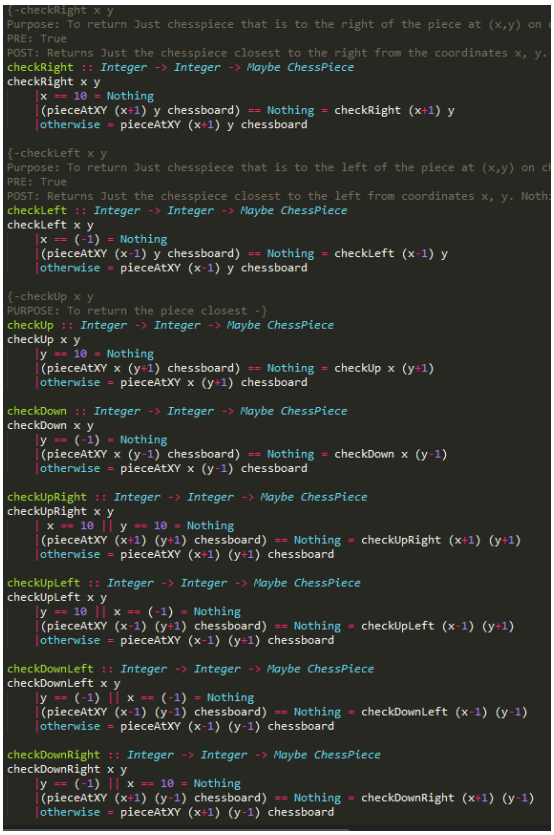
```haskell
where

    rankright = rankOf (convertMaybe (checkRight x y))
    colourright = colourOf (convertMaybe (checkRight x y))
    xright = xOf (convertMaybe (checkRight x y))

    rankleft = rankOf (convertMaybe (checkLeft x y))
    colourleft = colourOf (convertMaybe (checkLeft x y))
    xleft = xOf (convertMaybe (checkLeft x y))

    rankup = rankOf (convertMaybe (checkUp x y))
    colourup = colourOf (convertMaybe (checkUp x y))
    yup = yOf (convertMaybe (checkUp x y))

    rankdown = rankOf (convertMaybe (checkDown x y))
    colourdown = colourOf (convertMaybe (checkDown x y))
    ydown = yOf (convertMaybe (checkDown x y))

    rankupright = rankOf (convertMaybe (checkUpRight x y))
    colourupright = colourOf (convertMaybe (checkUpRight x y))
    yupright = yOf (convertMaybe (checkUpRight x y))
    xupright = xOf (convertMaybe (checkUpRight x y))

    rankupleft = rankOf (convertMaybe (checkUpLeft x y))
    colourupleft = colourOf (convertMaybe (checkUpLeft x y))
    yupleft = yOf (convertMaybe (checkUpLeft x y))

    rankdownleft = rankOf (convertMaybe (checkDownLeft x y))
    colourdownleft = colourOf (convertMaybe (checkDownLeft x y))
    ydownleft = yOf (convertMaybe (checkDownLeft x y))

    rankdownright = rankOf (convertMaybe (checkDownRight x y))
    colourdownright = colourOf (convertMaybe (checkDownRight x y))
    ydownright = yOf (convertMaybe (checkDownRight x y))

    knight1 = convertMaybe (pieceAtXY (x-2) (y-1) chessboard)
    knight2 = convertMaybe (pieceAtXY (x-1) (y-2) chessboard)
    knight3 = convertMaybe (pieceAtXY (x-2) (y+1) chessboard)
    knight4 = convertMaybe (pieceAtXY (x-1) (y+2) chessboard)
    knight5 = convertMaybe (pieceAtXY (x+1) (y+2) chessboard)
    knight6 = convertMaybe (pieceAtXY (x+2) (y+1) chessboard)
    knight7 = convertMaybe (pieceAtXY (x+2) (y-1) chessboard)
    knight8 = convertMaybe (pieceAtXY (x+1) (y-2) chessboard)
```

Step by step description:

1. **checkCheckKing** uses **giveKing** to get the specified king and sends the king and the chessboard to **checkKing**
2. **checkKing** has 8 guards that checks if the piece that is closest to the king in a specific direction is one that can kill him. The directions are right, left,up, down, up-right, up-left, down-left and up-right. The 9th guard checks the squares that

there might be a **Knight** and the 10th simple returns False if all previous guards fail which means that the king is safe.

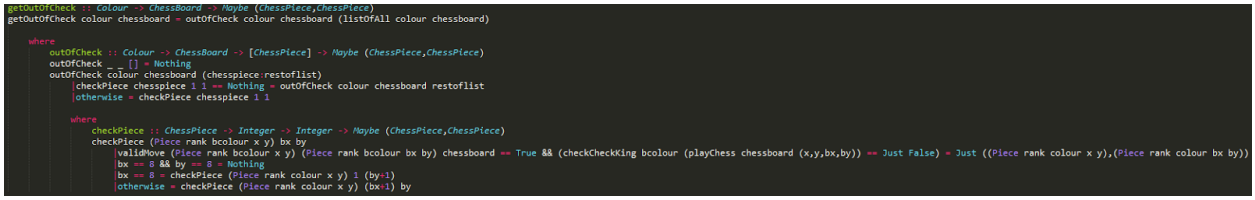
3. In the guards, a lot of the checks use variables with names that are declared in a where block to make the code more easy to read. For example, rankright is the rank of the closest **ChessPiece** to the right of the king. This is obtained from using simple functions **rankOf** & **convertOf** and a more complex function **checkRight**. **checkRight** is defined in a where block. The ranks, colours and coordinates for directions up, left, down, up-right, up-left, down-left and down-right are defined similarly to how rank colour and coordinated for direction right is defined.



```
{-checkRight x y
Purpose: To return Just chesspiece that is to the right of the piece at (x,y) on
PRE: True
POST: Returns Just the chesspiece closest to the right from the coordinates x, y.
checkRight :: Integer -> Integer -> Maybe ChessPiece
checkRight x y
    |x == 10 = Nothing
    |(pieceAtXY (x+1) y chessboard) == Nothing = checkRight (x+1) y
    |otherwise = pieceAtXY (x+1) y chessboard

{-checkLeft x y
Purpose: To return Just chesspiece that is to the left of the piece at (x,y) on c
PRE: True
POST: Returns Just the chesspiece closest to the left from coordinates x, y. Noth
checkLeft :: Integer -> Integer -> Maybe ChessPiece
checkLeft x y
    |x == (-1) = Nothing
    |(pieceAtXY (x-1) y chessboard) == Nothing = checkLeft (x-1) y
    |otherwise = pieceAtXY (x-1) y chessboard

{-checkUp x y
PURPOSE: To return the piece closest -}
checkUp :: Integer -> Integer -> Maybe ChessPiece
checkUp x y
    |y == 10 = Nothing
    |(pieceAtXY x (y+1) chessboard) == Nothing = checkUp x (y+1)
    |otherwise = pieceAtXY x (y+1) chessboard

checkDown :: Integer -> Integer -> Maybe ChessPiece
checkDown x y
    |y == (-1) = Nothing
    |(pieceAtXY x (y-1) chessboard) == Nothing = checkDown x (y-1)
    |otherwise = pieceAtXY x (y-1) chessboard

checkUpRight :: Integer -> Integer -> Maybe ChessPiece
checkUpRight x y
    | x == 10 || y == 10 = Nothing
    |(pieceAtXY (x+1) (y+1) chessboard) == Nothing = checkUpRight (x+1) (y+1)
    |otherwise = pieceAtXY (x+1) (y+1) chessboard

checkUpLeft :: Integer -> Integer -> Maybe ChessPiece
checkUpLeft x y
    |y == 10 || x == (-1) = Nothing
    |(pieceAtXY (x-1) (y+1) chessboard) == Nothing = checkUpLeft (x-1) (y+1)
    |otherwise = pieceAtXY (x-1) (y+1) chessboard

checkDownLeft :: Integer -> Integer -> Maybe ChessPiece
checkDownLeft x y
    |y == (-1) || x == (-1) = Nothing
    |(pieceAtXY (x-1) (y-1) chessboard) == Nothing = checkDownLeft (x-1) (y-1)
    |otherwise = pieceAtXY (x-1) (y-1) chessboard

checkDownRight :: Integer -> Integer -> Maybe ChessPiece
checkDownRight x y
    |y == (-1) || x == 10 = Nothing
    |(pieceAtXY (x+1) (y-1) chessboard) == Nothing = checkDownRight (x+1) (y-1)
    |otherwise = pieceAtXY (x+1) (y-1) chessboard
```

4. **checkRight** works by checking the square to the right of the king with **pieceAtXY** and if it finds a piece it returns it otherwise it continues to check to the right recursively until it is checking outside of the board. When it is checking outside of the board it means that it have not found any **ChessPiece** to the right.
5. When checking for possible Knights (9th guard of **checkKing**) that can kill the king it simply uses **pieceAtXY** on all 8 possible squares that can reach the king if a **Knight** stands on it. If a Knight with a different colour than the king stands on one of those, function returns True.

getOutOfCheck



```
getOutOfCheck :: Colour -> ChessBoard -> Maybe (ChessPiece,ChessPiece)
getOutOfCheck colour chessboard = outOfCheck colour chessboard (listOfAll colour chessboard)

    where
        outOfCheck :: Colour -> ChessBoard -> [ChessPiece] -> Maybe (ChessPiece,ChessPiece)
        outOfCheck _ _ [] = Nothing
        outOfCheck colour chessboard (chesspiece:restoflist)
            |checkPiece chesspiece 1 1 == Nothing = outOfCheck colour chessboard restoflist
            |otherwise = checkPiece chesspiece 1 1

            where
                checkPiece :: ChessPiece -> Integer -> Integer -> Maybe (ChessPiece,ChessPiece)
                checkPiece (Piece rank bcolour x y) bx by
                    |validMove (Piece rank bcolour x y) (Piece rank bcolour bx by) chessboard == True && (checkCheckKing bcolour (playChess chessboard (x,y,bx,by)) == Just False) = Just ((Piece rank colour x y),(Piece rank colour bx by))
                    |bx == 8 && by == 8 = Nothing
                    |bx == 8 = checkPiece (Piece rank colour x y) 1 (by+1)
                    |otherwise = checkPiece (Piece rank colour x y) (bx+1) by
```

Basic description: **getOutOfCheck** takes a **Colour**, a **ChessBoard** and gives back a **Maybe (ChessPiece,ChessPiece)**.
The function returns Just(chesspiece1,chesspiece2) where chesspiece1 and chesspiece2 represent a move that player **Colour** can make to not stand in check after the move. If there is no move that can be made, Nothing is returned, which means the game is over and ended in either checkmate or a stalemate.

Step by step description:

1. The function uses **listOfAll** with the **Colour** and **ChessBoard** argument of **getOutOfCheck** which returns a list of all the **ChessPieces** with the specified colour input on the chessboard. **outOfCheck** is called with the same colour and chessboard together with the list of chess pieces from **listOfAll**.
2. **outOfCheck** goes through the list of chess pieces recursively and sends every chesspiece together with to **Integers**, 1 and 1 to **checkPiece**. **checkPiece** will return either Nothing or Just(chesspiece1,chesspiece2). If Nothing is returned it will continue going through the list until the list is empty and return Nothing. If Just(chesspiece1,chesspiece2) is returned it will return the same Just(chesspiece1,chesspiece2).
3. **checkPiece** checks all the possible moves for the chesspiece. If that move will make the king not stand in check (this is checked by recursively going through the coordinates of all the 64 squares of a chessboard). Then it tests if moving the chesspiece there is a valid move with **validMove**. If so it will also check if that move will make the king not stand in check with the help of **playChess and checkCheckKing**.

giveKing

```haskell
{- giveKing colour chessboard
Purpose: To give Just chesspiece of the king of colour on chessboard
Pre: True
Post: Just chesspiece where chesspiece is the king of colour. Nothing if that
Examples:
    giveKing White createChessBoard == Just (Piece King White 5 1)
    giveKing Black createChessBoard == Just (Piece King Black 5 8)
    giveKing White (Board []) == Nothing
-}
giveKing :: Colour -> ChessBoard -> Maybe ChessPiece
giveKing _ (Board []) = Nothing
giveKing colour (Board ((Piece brank bcolour bx by):restofboard))
    |brank == King && colour == bcolour = Just (Piece brank bcolour bx by)
    |otherwise = giveKing colour (Board restofboard)
```

Basic description: The function **giveKing** simply determines if there is a king of the given colour on the chessboard. If there is a king of the given colour, it will return that king. This function is called by **checkCheckKing**, described above.

removeChessPiece

```haskell
removeChessPiece :: ChessPiece -> ChessBoard -> ChessBoard
removeChessPiece chesspiece board = Board (removePiece chesspiece board)

    where
        removePiece :: ChessPiece -> ChessBoard -> [ChessPiece]
        removePiece _ (Board []) = []
        removePiece rchesspiece (Board (chesspiece:restofboard))
            |rchesspiece == chesspiece = removePiece rchesspiece (Board restofboard)
            |otherwise = chesspiece : removePiece rchesspiece (Board restofboard)
```

Basic description: This function will remove a **ChessPiece** from a **ChessBoard**, using pattern matching on the sorted list that represents a **ChessBoard**.

printChessBoard

```haskell
printChessBoard :: ChessBoard -> IO()
printChessBoard chessboard = (printBoard chessboard 1 8 8) >> putStr "\n" >> printLetters

    where
        printBoard :: ChessBoard -> Integer -> Integer -> Integer -> IO()
        printBoard (Board []) 8 1 i = printEmpty >> putStr "  " >> print i
        printBoard (Board []) currentX currentY i
            |currentX == 8 = printEmpty >> putStr "  " >> print i >> printBoard (Board []) 1 (currentY - 1) (i-1)
            |otherwise = printEmpty >> printBoard (Board []) (currentX + 1) currentY i
        printBoard (Board ((Piece rank colour x y):rest)) 8 1 i = printPiece rank colour >> putStr "  " >> print i
        printBoard (Board ((Piece rank colour x y):rest)) currentX currentY i
            |currentX == x && x == 8 && currentY == y = printPiece rank colour >> putStr "  " >> print i >> printBoard (Board rest) 1 (currentY - 1) (i-1)
            |currentX == x && currentY == y = printPiece rank colour >> printBoard (Board rest) (currentX + 1) currentY i
            |currentX == 8 = printEmpty >> putStr "  " >> print i >> printBoard (Board ((Piece rank colour x y):rest)) 1 (currentY - 1) (i-1)
            |otherwise = printEmpty >> printBoard (Board ((Piece rank colour x y):rest)) (currentX + 1) currentY i

        {-printEmpty
        PURPOSE: To print the ASCII symbol for empty space to screen
        PRE: True
        POST: Returns "|__|"-}
        printEmpty :: IO()
        printEmpty = putStr "|__|"

        {-printLetters
        PURPOSE: To print letters A to H to the screen
        PRE: True
        POST: Returns "  A  B  C  D  E  F  G  H"-}
        printLetters :: IO()
        printLetters = putStr "  A   B   C   D   E   F   G   H \n"

        {-printPiece rank colour
        PURPOSE: To print the ASCII symbol for a specific rank of chesspiece/colour
        PRE: True
        POST: -}
        printPiece :: PieceRank -> Colour -> IO()
        printPiece rank colour
            |rank == King && colour == Black = putStr "|-K|"
            |rank == Queen && colour == Black = putStr "|-Q|"
            |rank == Rook && colour == Black = putStr "|-R|"
            |rank == Bishop && colour == Black = putStr "|-B|"
            |rank == Knight && colour == Black = putStr "|-N|"
            |rank == Pawn && colour == Black = putStr "|-P|"
            |rank == King = putStr "|+K|"
            |rank == Queen = putStr "|+Q|"
            |rank == Rook = putStr "|+R|"
            |rank == Bishop = putStr "|+B|"
            |rank == Knight = putStr "|+N|"
            |rank == Pawn = putStr "|+P|"
            |otherwise = putStr "|__|"
```

Basic description: This function takes in a chessboard and performs an IO operation to print it out.

Step by step description:

1) The first and only pattern match runs the function **printBoard** with the input chessboard and the numbers 1, 8, 8.
2) The first two pattern matches in **printBoard** is for when the board is empty, when all the pieces have been printed. Depending on the integers the remaining blank squares are printed out to complete the chessboard.
3) If there is any objects left in the list of the chessboard, the function then extracts the first object from it. if this was the last piece of a row, (i.e. the first integer is 8) the function writes a blank space, the number of the row, and then a linebreak (\n).
4) If the object wasn't the last in the list, the function calls itself recursively with the remaining part of the list.

pieceAtXY

```haskell
{- pieceAtXY x y chessboard
Purpose: To return the chesspiece at coordinaters (x,y) on chessboard
PRE: True
POST: Returns Just Chesspiece if there is a Chesspiece at x y on chessboard. Nothing otherwise.
EXAMPLES: PieceAtXY 1 2 createChessBoard = Just (Pawn White 1 2)-}
pieceAtXY :: Integer -> Integer -> ChessBoard -> Maybe ChessPiece
pieceAtXY _ _ (Board []) = Nothing
pieceAtXY x y (Board ((Piece rank colour bx by):restofboard))
    |x == bx && y == by = Just (Piece rank colour bx by)
    |otherwise = pieceAtXY x y (Board restofboard)
```

Basic description: The function **pieceAtXY** takes an x- and y-coordinate and returns the **ChessPiece** that has that coordinate. If there is no **ChessPiece**, Nothing will be returned. This function is crucial to make our program work, as most of our other functions call it.

Functions that call **pieceAtXY** are **movechessPiece**, **checkCheckKing** (including all the subfunctions), **validMove** (including **pawnCheck**, **queenCheck** etc.), as well as **aiMove**.

5. Known shortcomings of the program

There is one thing we chose to not fully implement. That is, when a pawn moves all the way to the opponents side (for instance a white pawn moves to 8 8), the player should have the option to choose which type of chess piece that pawn becomes.

Our implementation of this is that you are given a queen, as opposed to being able to pick the rank of the piece yourself. This is a result of us making sure that more important chess rules are enforced at all times.

All in all, we think that this is not a problem of big magnitude, as it is a very specific event that rarely happens in a game, and when it does happen, the queen is almost always picked.

Except for this matter, we feel that we have made a fully functional chess game.