Sudoku Solver in Haskell

Programkonstruktion och datastrukturer
(1DL201)

Marcus Vesterlund, Theo Meier Ström, Simon Damberg

Project report for the course Program Design and Data Structures (1DL201) at Uppsala University, March 2020.

UPPSALA
UNIVERSITET

Uppsala University
Sweden
March 4, 2020

Abstract

The group approached this project with a challenge in mind, while learning by doing was a large part of the brainstorming process. The group wanted something that would give each member enough of a challenge while also learning about new packages and methods of programming. We later decided to make a program that solves Sudoku.

There are a few ways of approaching the solving algorithm, and this program is built on a Brute-force algorithm using backtracking. The program implements an algorithm called a Depth-First Search solution. Depth-First Search first searches the board to see if every cell is of a fixed cell, if not choose a non-fixed cell and replace it with one of the possible values, then repeat the process. In cases where a non-fixed cell has no possible numbers available, the solver then backtracks where it fixed a non-fixed cell.

This program reaches out to a text file with more than 49000 standard 9x9 Sudoku boards of various difficulty. The user is able to randomize the shown board and solves the Sudoku board by pressing a single key.

The program will also translate the solving of the board into an animation. When the program is done solving a board, the user can with ease fetch a new board, and solve it without restarting the program.

Contents

1 Introduction

1.1 Background

We wanted to approach this project with the intention of learning by doing. Something that would be complex enough for us to try out functions and libraries we have not touched before. The idea of doing a Sudoku Solver originated early in the brainstorming process, while we also had other thoughts of doing a game like Tetris or Snake. Sudoku was something that had an interesting algorithm with much room for improvements and experimenting.

Sudoku is a puzzle game with the objective of solving a 9x9 grid of a total of 81 cells. The grid is also built upon 3x3 sub grids, which are sometimes called "regions". Each row, column, and region should contain the digits 1 to 9. (1)

In preparation for the project, we set up a Trello board and a Git to simplify the work for everyone in the group. These tools also made it easier to share thoughts, code, and the working process with each other.

When presenting the Sudoku Solver idea to our supervisor and got the suggestion accepted, the project as a whole took off.

The group encountered a few difficulties during the project, which resulted in some research and helping each other out. Most of these happened during the work on the code in SudokuGloss.hs which implemented the animation and user input using the Gloss module.

1.2 Inspiration

Whilst working on the project the group discovered a blog covering the same topic (2), something very helpful in situations were the group was stuck. One particular problem where the group drew inspiration from the blog was the newBoard function, specifically on how to grab the element from the board with the least amount of possible numbers using the minimumBy and 'compare on' functions.

1.3 Usage

There are not any real use cases for the program at the moment other than Solving random Sudoku boards. If the user runs the program in the terminal/GHCi, it is possible to use the functions in the Sudoku Solver to test specific Sudoku boards. However, it is not possible to do this while running the executable file. This is planned as an implementation for future developments and improvements for the program.

2 Instructions

To be able to run the program, see the instructions below. If the Glasgow Haskell Compiler and the required packages are already installed, you should be able to skip to step 3.

2.1 Packages

1. Install the Glasgow Haskell Compiler available here.
2. Run the following command to install the required packages with cabal:

   ```
   cabal install gloss random List HUnit split
   ```

3. Navigate to the folder containing the files in the terminal and run:

   ```
   ghc –make SudokuGloss.hs
   ```

4. Run SudokuGloss.exe

2.2 Controls

When the program is started, the user will be greeted with a random unsolved Sudoku. Use the following controls to use the program.

- Esc: Quits the program
- R: Randomizes the board into a new Sudoku
- Space: Solves the Sudoku

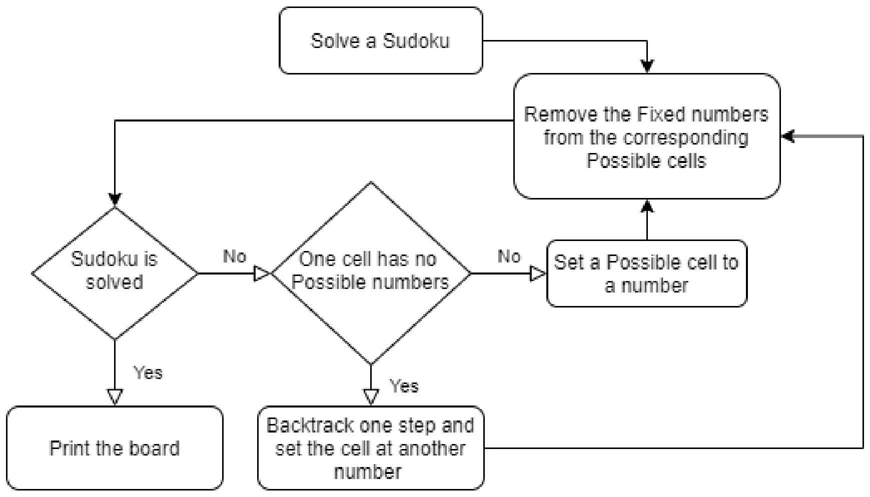
3 Documentation

3.1 Data Structures

The program consists of three fundamental data structures: Cell, Row, and Board. Cell can either be a Fixed number or a set of Possible numbers, where $0 < \text{number} < 10$. A Row is a set of 9 Cells, and a Board is a set of 9 Rows. Together, these data structures represent a Sudoku board.

3.2 Algorithm

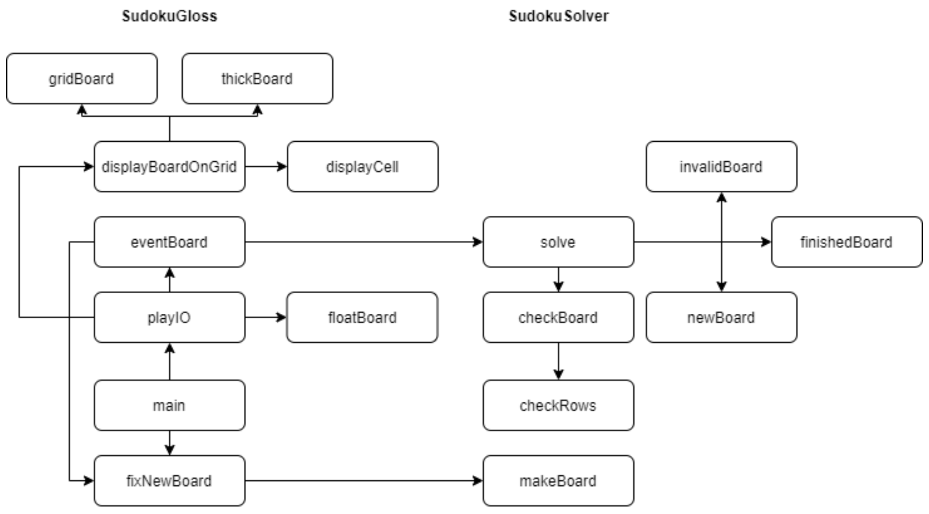
There exist multiple different algorithms for how to solve a Sudoku. The two most common are Stochastic search, randomly assigning numbers and shuffling until the errors are zero, and a Brute-force search using backtracking. This project focuses on the backtracking algorithm since the Stochastic search can take an unlimited amount of time and does not require any logic to the solution for the problem. The implemented algorithm is a Depth-First Search since it traverses completely down an entire path until it encounters a problem and then turns to the next legal path. It follows the following flowchart.


Solve a Sudoku
Remove the Fixed numbers from the corresponding Possible cells
Sudoku is solved
No
One cell has no Possible numbers
No
Set a Possible cell to a number
Yes
Yes
Print the board
Backtrack one step and set the cell at another number

The first step is to check if a Cell contains a fixed number, and remove that number from the set of possible numbers in each cell in the same row, column, and 3x3 square. After this step has been repeated for all fixed cells, check if every cell on the board is fixed. In that case, the Sudoku is solved! Otherwise, choose a non-fixed cell and fix that cell to one of the possible values. Go back to the first step and repeat. Since the algorithm is assuming numbers, it may result in a situation where one non-fixed cell does not have any possible numbers left. In this case, the program discards that branch of the solution and backtracks to the previous step where it fixed a non-fixed cell. This time, it chooses another number and fixes the cell to that number instead.

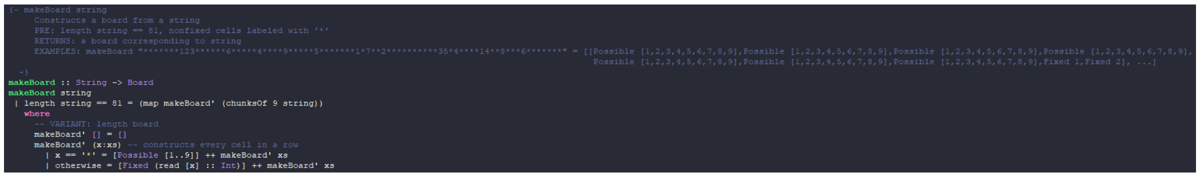
3.3 Program structure

The program consists of two files, SudokuSolver and SudokuGloss, where SudokuSolver handles the algorithm and solving of a Sudoku whilst SudokuGloss handles the IO and graphical parts. To achieve this, SudokuGloss uses the Gloss library and its built in function playIO to create a game loop and handle input from the user whilst also being able to display the Sudoku graphically. The program structure and how all the functions interact with each other is visible in the flowchart below.



3.4 SudokuSolver

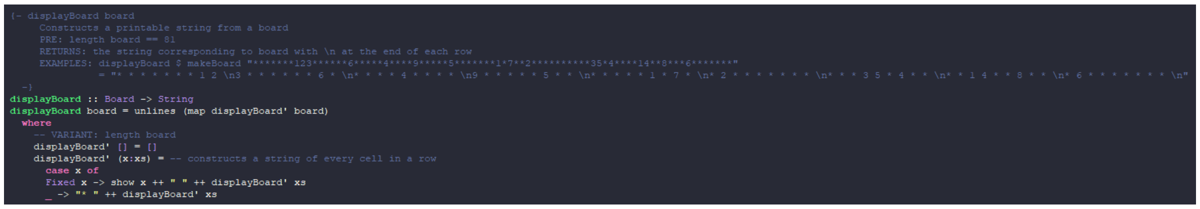
3.4.1 makeBoard



```haskell
{- makeBoard string
     Constructs a board from a string
     PRE: length string == 81, nonfixed cells labeled with '*'
     RETURNS: a board corresponding to string
     EXAMPLES: makeBoard "*******123******6*****4****9*****5*******1*7**2**********35*4****14**8***6*******" = [[Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],
                                                                                                                  Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Possible [1,2,3,4,5,6,7,8,9],Fixed 1,Fixed 2], ...]
  -}
makeBoard :: String -> Board
makeBoard string
 | length string == 81 = (map makeBoard' (chunksOf 9 string))
   where
     -- VARIANT: length board
     makeBoard' [] = []
     makeBoard' (x:xs) -- constructs every cell in a row
       | x == '*' = [Possible [1..9]] ++ makeBoard' xs
       | otherwise = [Fixed (read [x] :: Int)] ++ makeBoard' xs
```

makeBoard is used at the start of the main-function or, more specifically, in the fixNewBoard-function. It gets a random string from our sudoku-textfile and then converts it into a Board. In makeBoard, we are using the function chunksOf from Data.List.Split. It converts a string into a list of strings with the specified length. This function is handy since we want to split the string into nine character long strings, which are then converted into the actual Board. The conversion is pretty simple. Every star becomes a Possible list of numbers, and every number becomes a Fixed number.

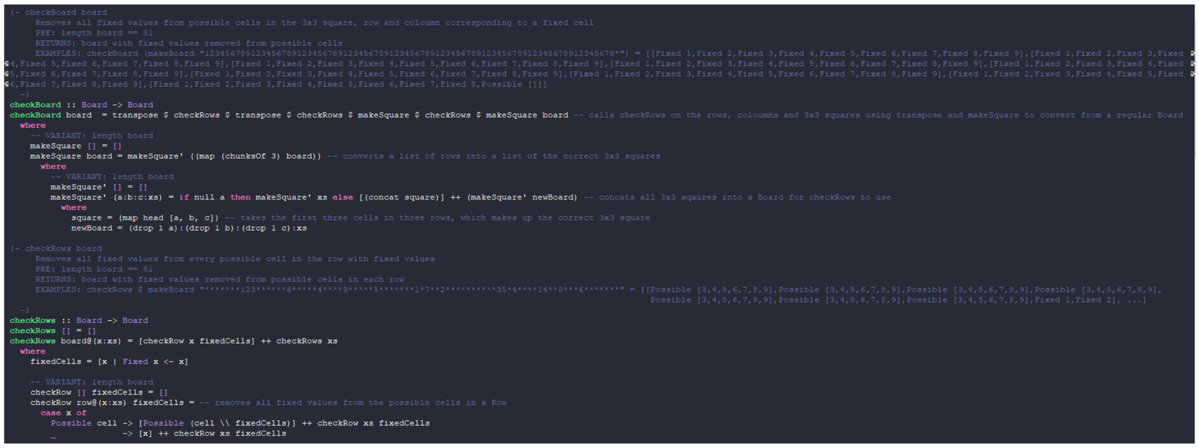
3.4.2 displayBoard



```haskell
{- displayBoard board
     Constructs a printable string from a board
     PRE: length board == 81
     RETURNS: the string corresponding to board with \n at the end of each row
     EXAMPLES: displayBoard $ makeBoard "*******123******6*****4****9*****5*******1*7**2**********35*4****14**8***6*******"
                 = "* * * * * * * 1 2 \n3 * * * * * * 6 * \n* * * * 4 * * * * \n9 * * * * * 5 * * \n* * * * * 1 * 7 * \n* 2 * * * * * * * * \n* * * 3 5 * 4 * * \n* * 1 4 * * 8 * * \n* 6 * * * * * * * * \n"
  -}
displayBoard :: Board -> String
displayBoard board = unlines (map displayBoard' board)
  where
    -- VARIANT: length board
    displayBoard' [] = []
    displayBoard' (x:xs) = -- constructs a string of every cell in a row
      case x of
      Fixed x -> show x ++ " " ++ displayBoard' xs
      _ -> "* " ++ displayBoard' xs
```

The function displayBoard is a function that is not used when the main-function runs. It was instead used in the early stages of development to display the board in the terminal. The problem that the function solves is that boards are shown as lists of Possible and Fixed, these lists are hard to comprehend. The function is straight-forward as it converts every element into either a number or a star. Unlines is used in this function, and what it does is concatenate multiple strings into one single string with newline characters between the original strings.

3.4.3 checkBoard

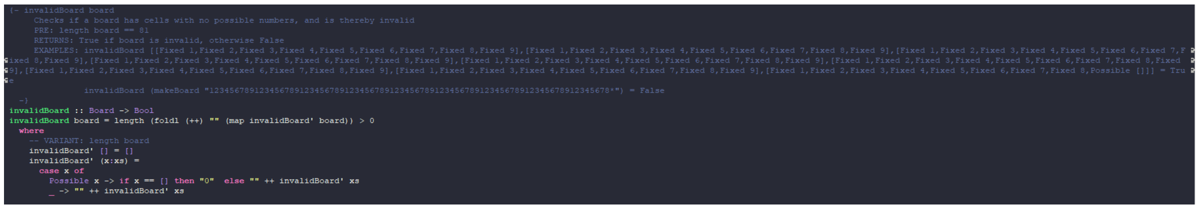


```haskell
{- checkBoard board
     Removes all fixed values from possible cells in the 3x3 square, row and coloumn corresponding to a fixed cell
     PRE: length board == 81
     RETURNS: board with fixed values removed from possible cells
     EXAMPLES: checkBoard (makeBoard "123456789123456789123456789123456789123456789123456789123456789123456789123456 78*") = [[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed
4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed
5,Fixed 6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed
6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Possible []]]
  -}
checkBoard :: Board -> Board
checkBoard board  = transpose $ checkRows $ transpose $ checkRows $ makeSquare $ checkRows $ makeSquare board -- calls checkRows on the rows, coloumns and 3x3 squares using transpose and makeSquare to convert from a regular Board
  where
    -- VARIANT: length board
    makeSquare [] = []
    makeSquare board = makeSquare' ((map (chunksOf 3) board)) -- converts a list of rows into a list of the correct 3x3 squares
      where
        -- VARIANT: length board
        makeSquare' [] = []
        makeSquare' (a:b:c:xs) = if null a then makeSquare' xs else [(concat square)] ++ (makeSquare' newBoard) -- concats all 3x3 sqaures into a Board for checkRows to use
          where
            square = (map head [a, b, c]) -- takes the first three cells in three rows, which makes up the correct 3x3 square
            newBoard = (drop 1 a):(drop 1 b):(drop 1 c):xs

{- checkRows board
     Removes all fixed values from every possible cell in the row with fixed values
     PRE: length board == 81
     RETURNS: board with fixed values removed from possible cells in each row
     EXAMPLES: checkRows $ makeBoard "*******123******6*****4****9*****5*******1*7**2**********35*4****14**8***6*******" = [[Possible [3,4,5,6,7,8,9],Possible [3,4,5,6,7,8,9],Possible [3,4,5,6,7,8,9],Possible [3,4,5,6,7,8,9],
                                                                                                                          Possible [3,4,5,6,7,8,9],Possible [3,4,5,6,7,8,9],Possible [3,4,5,6,7,8,9],Fixed 1,Fixed 2], ...]

  -}
checkRows :: Board -> Board
checkRows [] = []
checkRows board@(x:xs) = [checkRow x fixedCells] ++ checkRows xs
  where
    fixedCells = [x | Fixed x <- x]

    -- VARIANT: length board
    checkRow [] fixedCells = []
    checkRow row@(x:xs) fixedCells = -- removes all fixed values from the possible cells in a Row
      case x of
        Possible cell -> [Possible (cell \\ fixedCells)] ++ checkRow xs fixedCells
        _             -> [x] ++ checkRow xs fixedCells
```

The function checkBoard is one of the key functions in the program. Its purpose is to remove all fixed numbers from its corresponding row, column, and 3x3 square. checkRows is the function that does the actual work, whereas checkBoard does the converting. Since the board is built upon a list of rows, the transpose function from Data.List is perfect for converting the rows into columns. This is because it creates a new list with the first element from each row, which is the correct column. checkBoard is also able to convert the rows into 3x3 squares, by separating each row into chunks of 3 cells and concatenate the first chunk in three rows to create the correct 3x3 square. checkBoard then calls the function checkRows on each of the three types of boards.

checkRows is the function that actually removes the fixed numbers from a given list of rows. It does this by calling an auxiliary function checkRow on each row with a list of the fixed numbers as an argument. checkRow takes a row and checks one cell at a time recursively. If the cell is non-fixed, it uses the \\function from Data.List to filter out the already fixed numbers from the list of possible numbers in the cell.

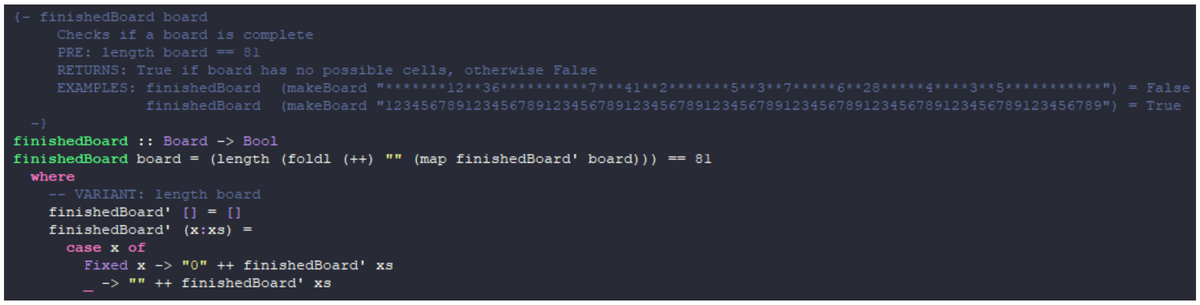
3.4.4 invalidBoard



```
{- invalidBoard board
     Checks if a board has cells with no possible numbers, and is thereby invalid
     PRE: length board == 81
     RETURNS: True if board is invalid, otherwise False
     EXAMPLES: invalidBoard [[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,F
ixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Fixed 
9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Fixed 9],[Fixed 1,Fixed 2,Fixed 3,Fixed 4,Fixed 5,Fixed 6,Fixed 7,Fixed 8,Possible []]] = Tru
e
                invalidBoard (makeBoard "12345678912345678912345678912345678912345678912345678912345678912345678912345678*") = False
  -}
invalidBoard :: Board -> Bool
invalidBoard board = length (foldl (++) "" (map invalidBoard' board)) > 0
  where
    -- VARIANT: length board
    invalidBoard' [] = []
    invalidBoard' (x:xs) =
      case x of
        Possible x -> if x == [] then "0"  else "" ++ invalidBoard' xs
        _ -> "" ++ invalidBoard' xs
```

The function invalidBoard checks if the board is invalid. By invalid, we mean that one of the lists of possible numbers is empty. If a list is empty, that means that something has gone wrong, and we need to backtrack a step. It does all this by checking every cell on the board. If a cell has a Possible list that is empty, it returns a 0. Otherwise, it returns an empty string. It then adds all of these strings together. If the length of this string is over 0, the board is invalid.

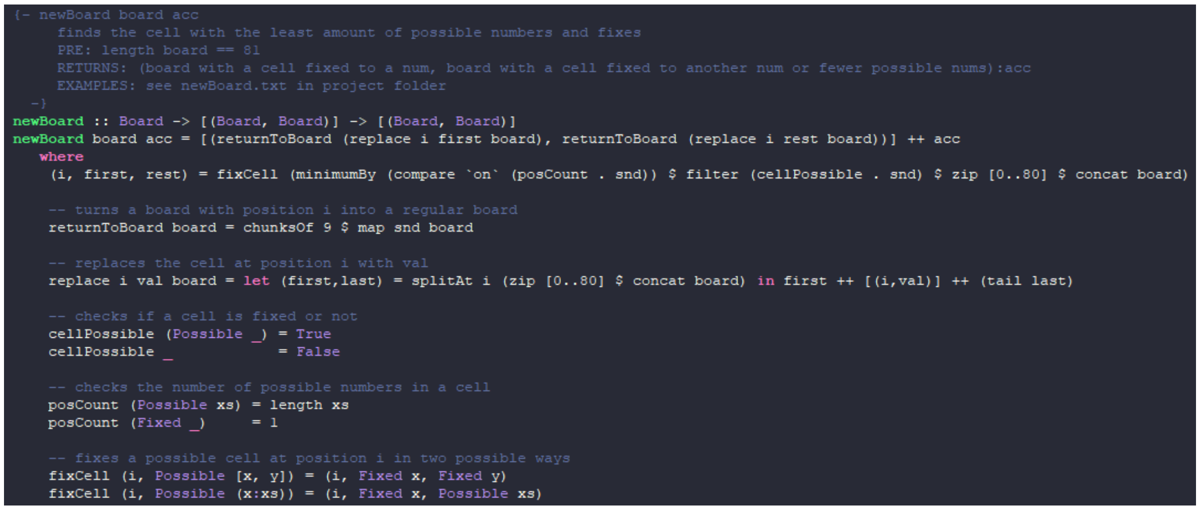
3.4.5 finishedBoard



```
{- finishedBoard board
     Checks if a board is complete
     PRE: length board == 81
     RETURNS: True if board has no possible cells, otherwise False
     EXAMPLES: finishedBoard  (makeBoard "*******12**36*********7***41**2*******5**3**7*****6**28*****4****3**5**********") = False
               finishedBoard  (makeBoard "123456789123456789123456789123456789123456789123456789123456789123456789123456789") = True
  -}
finishedBoard :: Board -> Bool
finishedBoard board = (length (foldl (++) "" (map finishedBoard' board))) == 81
  where
    -- VARIANT: length board
    finishedBoard' [] = []
    finishedBoard' (x:xs) =
      case x of
        Fixed x -> "0" ++ finishedBoard' xs
        _ -> "" ++ finishedBoard' xs
```

finishedBoard is a simple function that checks if a board is solved. It checks this by creating a string where all fixed cells are represented with a "0", and all non-fixed cells with an empty string. If the length of this created string is 81, that means that each cell is Fixed and that the board is solved.

3.4.6 newBoard



```haskell
{- newBoard board acc
     finds the cell with the least amount of possible numbers and fixes
     PRE: length board == 81
     RETURNS: (board with a cell fixed to a num, board with a cell fixed to another num or fewer possible nums):acc
     EXAMPLES: see newBoard.txt in project folder
  -}
newBoard :: Board -> [(Board, Board)] -> [(Board, Board)]
newBoard board acc = [(returnToBoard (replace i first board), returnToBoard (replace i rest board))] ++ acc
   where
    (i, first, rest) = fixCell (minimumBy (compare `on` (posCount . snd)) $ filter (cellPossible . snd) $ zip [0..80] $ concat board)

    -- turns a board with position i into a regular board
    returnToBoard board = chunksOf 9 $ map snd board

    -- replaces the cell at position i with val
    replace i val board = let (first,last) = splitAt i (zip [0..80] $ concat board) in first ++ [(i,val)] ++ (tail last)

    -- checks if a cell is fixed or not
    cellPossible (Possible _) = True
    cellPossible _            = False

    -- checks the number of possible numbers in a cell
    posCount (Possible xs) = length xs
    posCount (Fixed _)     = 1

    -- fixes a possible cell at position i in two possible ways
    fixCell (i, Possible [x, y]) = (i, Fixed x, Fixed y)
    fixCell (i, Possible (x:xs)) = (i, Fixed x, Possible xs)
```

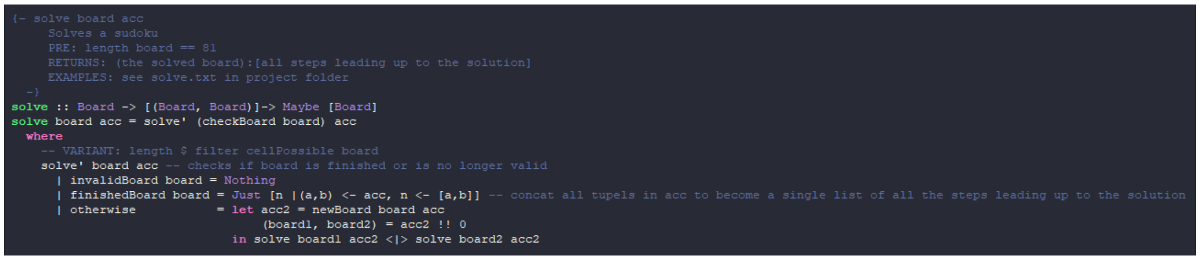
The function newBoard is called from the solve-function. The function searches for the cell with the least amount of possible number. The reason that we search for the lowest one is that the possibility of guessing right is a lot higher there than in any other Possible-cell. The function has multiple auxiliary functions.

The most important of these is replace. It gets a position, a value, and a board. It then finds that position on the board and changes the value of that cell to the given one. The next auxiliary function is fixCell. It splits a Possible list into two separate ones. If the possible list only has two elements, then it is split into two Fixed ones. If the Possible list has more than two elements, it is split into one that is the first element fixed, and one thought is the rest of the Possible list.

Another auxiliary is posCount, which returns the number of possibilities the cell has. If it is fixed, then the answer is 1. Otherwise, it is the length of the list. Another auxiliary is returnToBoard, which removes the index-tuples and makes it into a Board again. The last auxiliary is cellPossible; it just checks if the cell is Fixed or not.

Then all these auxiliaries are used to change a board into two new boards, which both then can be tested in solve. These new boards have two different possibilities to test which one is correct. This line (minimumBy (compare ‘on‘ (posCount . snd)) ”list” is interesting. What happens here is that instead of just using minimum, minimumBy is used. So it does look for a minimum, but it does it using the compare on posCount.snd of the list. This is used since we have a list of tuples and want to compare the second element in them but still keep the first one around.

3.4.7 solve



```haskell
{- solve board acc
     Solves a sudoku
     PRE: length board == 81
     RETURNS: (the solved board):[all steps leading up to the solution]
     EXAMPLES: see solve.txt in project folder
  -}
solve :: Board -> [(Board, Board)]-> Maybe [Board]
solve board acc = solve' (checkBoard board) acc
  where
    -- VARIANT: length $ filter cellPossible board
    solve' board acc -- checks if board is finished or is no longer valid
      | invalidBoard board = Nothing
      | finishedBoard board = Just [n |(a,b) <- acc, n <- [a,b]] -- concat all tupels in acc to become a single list of all the steps leading up to the solution
      | otherwise           = let acc2 = newBoard board acc
                                  (board1, board2) = acc2 !! 0
                              in solve board1 acc2 <|> solve board2 acc2
```

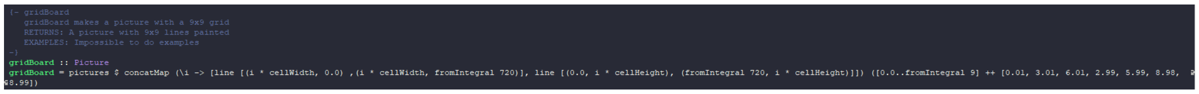
Solve is the function that combines all the previous functions into a loop that solves the Sudoku. It takes two arguments, a board and an accumulator, which, in this case, will be a list of all the previous steps in the solving process. The accumulator is not needed for the function to work and was implemented later for the Gloss part of the program to be able to keep track of all the steps and animate the solution. The first step is to call on an auxiliary function with the fixed numbers removed from the possible cells using the function checkBoard.

The auxiliary function has three guards that check different scenarios. The first guard checks if the board is invalid using the invalidBoard function; in that case, it returns Nothing. The second guard checks if the board is solved using the finishedBoard function. If the board is solved, it returns the accumulator but modified using list comprehension to turn the list of tuples into a single list of all the solutions. The list comprehension part was taken from stackoverflow (3).

If none of the scenarios above are met, it calls on newBoard to create two new possible boards. To be able to check both branches of the solution, the function uses an implementation of the Maybe type called Alternative (<|>), which works perfectly in this scenario. This function first runs solve on the first board recursively as its first branch, and if that returns Nothing, it will instead run solve with the second board as the second branch. This is why the solve function returns Nothing if the board is invalid since this will lead to the function backtracking one step and continue with the other possible board.

3.5 SudokuGloss

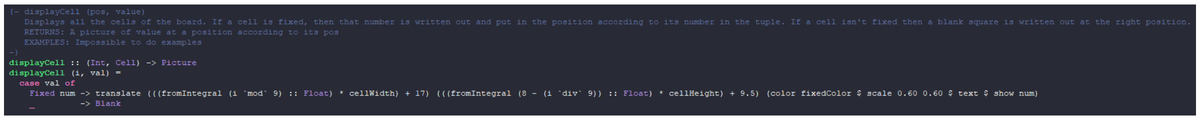
3.5.1 gridBoard



```
{- gridBoard
   gridBoard makes a picture with a 9x9 grid
   RETURNS: A picture with 9x9 lines painted
   EXAMPLES: Impossible to do examples
-}
gridBoard :: Picture
gridBoard = pictures $ concatMap (\i -> [line [(i * cellWidth, 0.0) ,(i * cellWidth, fromIntegral 720)], line [(0.0, i * cellHeight), (fromIntegral 720, i * cellHeight)]]) ([0.0..fromIntegral 9] ++ [0.01, 3.01, 6.01, 2.99, 5.99, 8.98,
8.99])
```

The function gridBoard makes the original grid. It is called in the displayBoardOnGrid-function. It uses a function called concatMap; this function concats the vertical and horizontal lines into one big grid using the pictures function. The function gridBoard is originally made by Alexey Kutepov. (4)

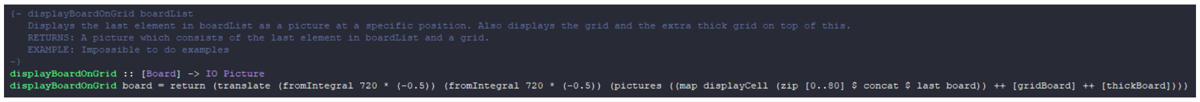
3.5.2 displayCell



```
{- displayCell (pos, value)
   Displays all the cells of the board. If a cell is fixed, then that number is written out and put in the position according to its number in the tuple. If a cell isn't fixed then a blank square is written out at the right position.
   RETURNS: A picture of value at a position according to its pos
   EXAMPLES: Impossible to do examples
-}
displayCell :: (Int, Cell) -> Picture
displayCell (i, val) =
  case val of
    Fixed num -> translate (((fromIntegral (i `mod` 9) :: Float) * cellWidth) + 17) (((fromIntegral (8 - (i `div` 9)) :: Float) * cellHeight) + 9.5) (color fixedColor $ scale 0.60 0.60 $ text $ show num)
    _         -> Blank
```

displayCell converts a tuple of a position as an int between 0 and 80 and a value into a picture. The function translate is used since that puts the picture at a specific position. To calculate the position, the int is run mod 9 and div 9. What this does is it gets the position of the cell on the row and the position of the row on the board. Then the value is displayed as a green number on that position. displayCell is also called upon from displayBoardOnGrid.

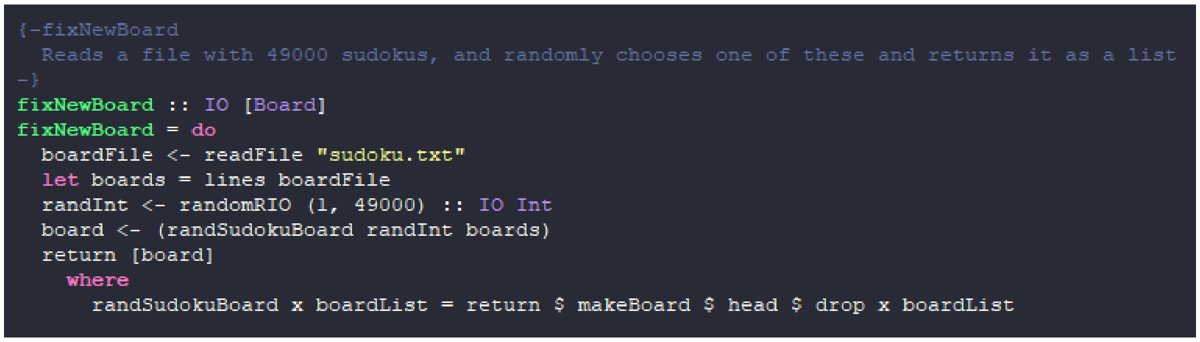
3.5.3 displayBoardOnGrid



```
{- displayBoardOnGrid boardList
   Displays the last element in boardList as a picture at a specific position. Also displays the grid and the extra thick grid on top of this.
   RETURNS: A picture which consists of the last element in boardList and a grid.
   EXAMPLE: Impossible to do examples
-}
displayBoardOnGrid :: [Board] -> IO Picture
displayBoardOnGrid board = return (translate (fromIntegral 720 * (-0.5)) (fromIntegral 720 * (-0.5)) (pictures ((map displayCell (zip [0..80] $ concat $ last board)) ++ [gridBoard] ++ [thickBoard])))
```

This function takes the board state and converts the last board in the list into a picture. To show all the values on the board, displayCell is run on every element on the board. All of the pictures from that are added with the grid and then displayed on the screen.

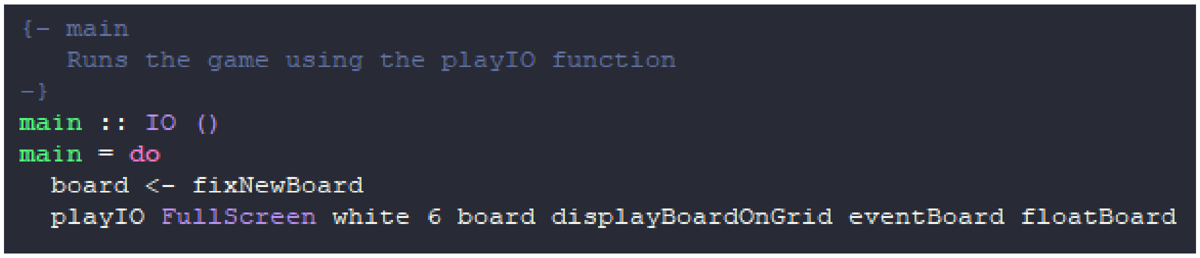
3.5.4 fixNewBoard



```haskell
{-fixNewBoard
  Reads a file with 49000 sudokus, and randomly chooses one of these and returns it as a list
-}
fixNewBoard :: IO [Board]
fixNewBoard = do
  boardFile <- readFile "sudoku.txt"
  let boards = lines boardFile
  randInt <- randomRIO (1, 49000) :: IO Int
  board <- (randSudokuBoard randInt boards)
  return [board]
    where
      randSudokuBoard x boardList = return $ makeBoard $ head $ drop x boardList
```

fixNewBoard is an IO-function that reads a randomized Sudoku from sudoku.txt and calls upon makeBoard to turn the string into a Board. This is done by creating a random number and then dropping that amount of Sudokus from the list of boards read from the file.

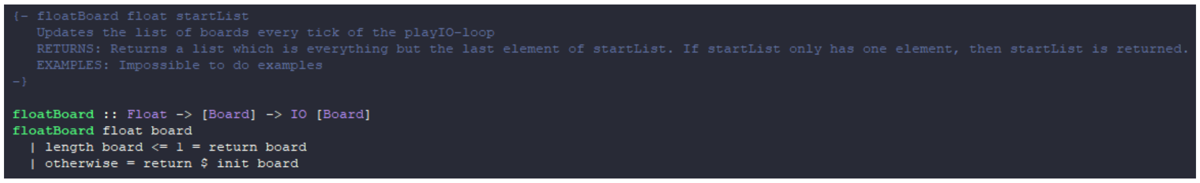
3.5.5 main



```haskell
{- main
   Runs the game using the playIO function
-}
main :: IO ()
main = do
  board <- fixNewBoard
  playIO FullScreen white 6 board displayBoardOnGrid eventBoard floatBoard
```

The main function does two things: create a board by calling upon fixNewBoard and call the function PlayIO from Gloss, which handles the game loop. PlayIO takes seven arguments: display mode, background color, number of iterations per second, the current board state, a function for drawing a board (displayBoardOnGrid), a function to handle events (eventBoard) and a function for updating the board each iteration (floatBoard).

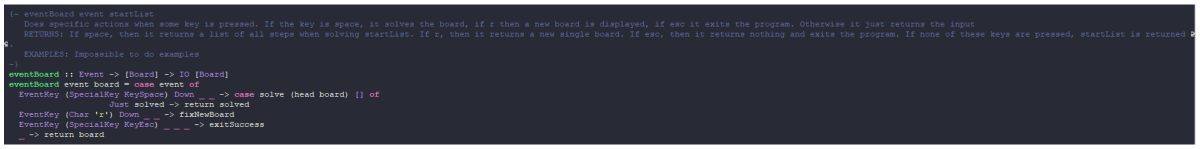
3.5.6 floatBoard



```
{- floatBoard float startList
   Updates the list of boards every tick of the playIO-loop
   RETURNS: Returns a list which is everything but the last element of startList. If startList only has one element, then startList is returned.
   EXAMPLES: Impossible to do examples
-}

floatBoard :: Float -> [Board] -> IO [Board]
floatBoard float board
  | length board <= 1 = return board
  | otherwise = return $ init board
```

floatBoard is the function that gets called on by PlayIO each iteration and is responsible for the animation taking place. Since displayBoardOnGrid displays the last board in the board state, the program is able to simulate animation by removing the last board each iteration. If there only is one board remaining in board state, the function does nothing since that will either be the solved or the initial board.

3.5.7 eventBoard



```
{- eventBoard event startList
   Does specific actions when some key is pressed. If the key is space, it solves the board, if r then a new board is displayed, if esc it exits the program. Otherwise it just returns the input
   RETURNS: If space, then it returns a list of all steps when solving startList. If r, then it returns a new single board. If esc, then it returns nothing and exits the program. If none of these keys are pressed, startList is returned
.
   EXAMPLES: Impossible to do examples
-}
eventBoard :: Event -> [Board] -> IO [Board]
eventBoard event board = case event of
  EventKey (SpecialKey KeySpace) Down _ _ -> case solve (head board) [] of
                  Just solved -> return solved
  EventKey (Char 'r') Down _ _ -> fixNewBoard
  EventKey (SpecialKey KeyEsc) _ _ _ -> exitSuccess
  _ -> return board
```

The function eventBoard reacts whenever one of the specified inputs are made. It handles the escape key, which exits the program, the r key which calls fixNew-Board to randomize a new board, and the space-bar which solves the board and then shows it animated.

4 Discussion

4.1 Shortcomings

The code itself is entirely functional for the desired purpose. However, the animation is made to be slow so that the user with ease can follow along with the solution and therefore it may seem like the solver is very slow, when in fact it can solve the Sudoku in a couple of milliseconds. Since the current implementation of the solver saves each board state in a single list, this list may very extremely in size and slow down the computing time. Because of this, the solving time increased after adding the support for animating.

The animation waits for the solving to completely finish before it starts to animate, and in cases where the Sudoku takes several seconds to solve the program may freeze and stop responding. We assume this is because the game loop suspects something has gone wrong since one function hasn't returned anything for a couple of seconds. This problem is solved by waiting a couple of seconds for the solution to finish. In these cases, the animation will already have gone through some steps of the animation before it shows anything.

There is one problem with the program that can cause a crash. This is when the user presses the button to solve a board that already has been solved. This was not fixed because of lack of time, but would not be hard to fix.

4.2 Improvements & future developments

There is currently no way of importing a Sudoku board directly into the program itself. In order to have the program solve a requested board, the code would have to be rewritten a little bit while also changing the text file that is imported into the program. A solution to this would be to implement a feature that makes it possible for the user to import a board while the program is running. Alternatively, make it possible to choose a file and change the randomizer, so it works for the given file.

After a Sudoku board has been solved, you can not press the solve button again without the program crashing. This could be improved by instead of making the program crash, the user is noticed of the already solved board and what button to press if they want a new board to solve.

A future development that could be implemented is to make it possible for the player to actually play Sudoku in the program. When the user is done solving the board, it is possible to correct it with the help of the Solver.

Another possible improvement is to make the actual solver faster and add some form of logic to it the algorithm. For example, imagine a board with the following row:

```
[1, 5] 2 3 [6, 8, 9] 4 [6, 7] [1, 6, 7] [5, 6, 7] 8
```

To a human solving a Sudoku, it might be obvious to set the fourth cell to 9, since its the only cell in the row with 9 as a possibility, but not for this algorithm. Instead of approaching the cell with the least amount of possible numbers and try combinations that will result in invalid boards, the algorithm could first look through the rows and locate these types of scenarios. This could result in a faster solution time, and would not be that difficult to implement. The previously mentioned blog (2) achieved a 200x faster solution time after combining this method with other specific edge cases.

The interface could also use an upgrade. For example, to make it more visible what has changed and how the board initially looked, the solved cells could be of a different color. An alternative solution of the same principle is to make two boards in the program, the unsolved and the solved board; therefore, the user can immediately compare the two boards against each other.

4.3 Final result and conclusion

The group has been using both Trello and GitHub very frequently and have been the tools that helped the workflow of the project.

The Sudoku Solver has two minor issues. When the user tries to solve an already solved board, the program then closes, and the user will have to restart. The other one is that some Sudoku boards are harder to animate, the animation freezes and skip frames, usually in the beginning. Other than that, the program runs smoothly without any problems. To run the solver, the user might need to go through an installation process that is of no difficulty, and therefore, the solver is easy to test and use.

Finally, even despite the minor issues, the final program is functional and can solve any standard 9x9 Sudoku board. The program is also able to create an animation of how it solves the Sudoku.

References

[1] "What is sudoku?" Available at http://www.sudoku-space.com/sudoku.php (2020/03/02).

[2] A. Sarkar, "Fast sudoku solver in haskell #1: A simple solution," Available at https://abhinavsarkar.net/posts/fast-sudoku-solver-in-haskell-1 (2020/03/04).

[3] "How to convert a list of tuples into a regular list in haskell?" Available at https://stackoverflow.com/questions/47020057/how-to-convert-a-list-of-tuples-into-a-regular-list-in-haskell (2020/03/02).

[4] A. Kutepov, "Procedural vs functional - rendering.hs," Available at https://github.com/tsoding/profun/blob/master/functional/src/Rendering.hs (2020/03/03).