Uppsala University

Information Technology

PKD 2018/2019

CandyBreak

Authors:
Fredrik Yngve, Erik Junghahn,
August Bredberg, Markus Winberg

March 4, 2019

Contents

1 Introduction and purpose

In order to improve our knowledge about Haskell and functional programming in general we decided to program a game similar to the widely successful Candy Crush which we chose to call CandyBreak.

Programming games in Haskell can be quite a challenge especially when working with a lot of random elements as Haskell handles randomization quite poorly.

Our plan was to make a gameboard consisting of a list with multiple candies with x- and y-coordinate, list index and color. Giving the candies an index was supposed to make it simpler to look up a specific candy and retrieve its specific coordinates and color value. In addition to this we also planned to generate a list of random color elements at the start of the game to serve as a bank to retrieve colors from as three or more equal candies were lined up and removed.

The program works a lot with updating and rendering a game state that is affected by the various functions and are explained in detail in this documentation.

During the development we encountered a lot of problems and obstacles mostly associated with generating random numbers in large amounts and in deciding which was the optimal way to design certain functions. This resulted in a frequent change of strategy in the beginning of development until we agreed on an optimal method involving the design of the **Candy** datatype.

1.1 Abstract

This paper is an attempt at proving our thesis that it is possible to create a fully functioning Candy Crush game in the programming language Haskell. The report covers the structure of the code and explains how each function is connected to the other for the game to work. All of the important functions are given both a short and a detailed description and each written data type is explained in order for the reader to fully understand the logic behind it.

Another goal of the report is to inspire and encourage others to try and do their own take on programming a game in Haskell or any other language. This is both a great way to improve ones knowledge about Haskell and to be introduced to the great world of programming.

The report also covers some of the challenges that a person can expect to encounter while writing a similar program and how these issues were solved in this project.

Along with this, the report also goes through how to play the game and how to set it up on a Linux and Mac OS operating system in step-by-step description.

2 Use cases

CandyBreak is started by running a "Unix executable"-file built using Stack. In order for a new user to start the executable they must follow these steps.

TL:DR
1) Install the latest version of Stack by following this guide `https://docs.haskellstack.org/en/stable/README/`.
2) Locate the folder **candyBreak** in the terminal.
3) Run command "stack build && stack run" to build and run the program.
Done!

2.1 How to play

In this image you can see an active session of the game. In the center of the screen is a board of 7 different randomly generated candies that are distinguished by their colors. A score-counter is located to the left of the screen and helps the player keep track of their current score and a timer is located to the right, telling the player how much time they have left. At the start of the game, the player starts in the top left position and must use the arrow keys to navigate the gameboard.

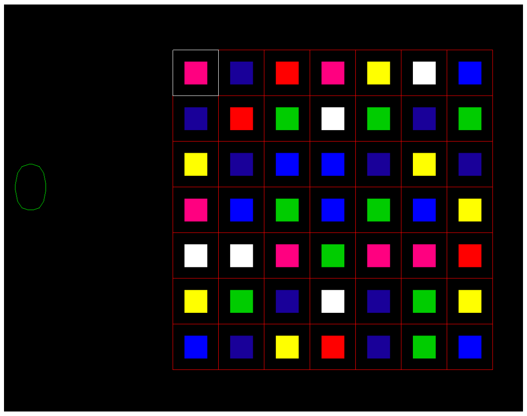


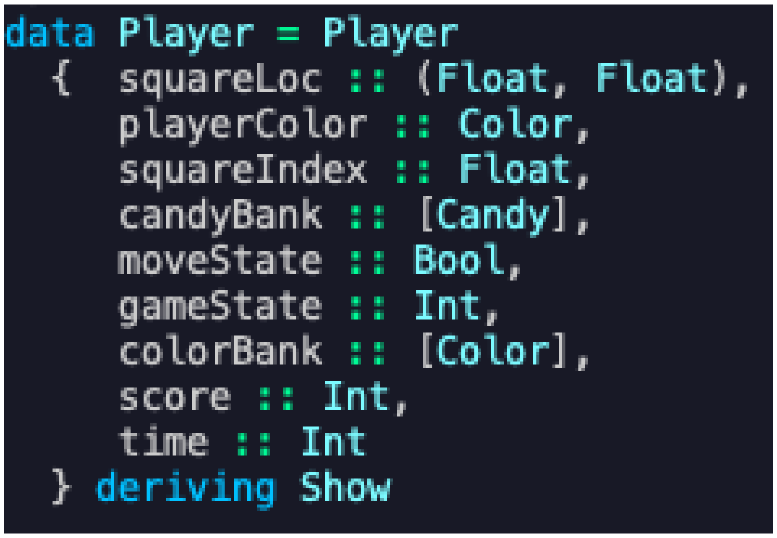
Figure 1: Gameboard

The objective is to find candies where a row of three or more of the same type of candy is achievable by making a single swap between two adjacent candies in any direction. In order to make a move the player locks onto the candy that he or she wishes to swap by pressing the "enter"-button on the keyboard and by once again using the arrow-keys to choose which adjacent candy is supposed to take its place. If a move results in that a row of three or more candies are created the move is made, otherwise the move is *not made.*

If the move is valid the row of candies is removed and the player is given a point. The removed row leaves space for more candies to fall into its place and more randomly generated candies are summoned at the top of the grid. Whenever new candy falls down from the top of the grid it is possible for new rows to appear if the player is lucky or has planned ahead. Once the timer has run out the game is over and the score is displayed in the window. If a player would ever wish to quit playing, the **ESC** key will close the program. The player can also start over by pressing 'm' to go back to the start menu.

3 Documentation

3.1 Datatype



```
data Player = Player
  {  squareLoc :: (Float, Float),
     playerColor :: Color,
     squareIndex :: Float,
     candyBank :: [Candy],
     moveState :: Bool,
     gameState :: Int,
     colorBank :: [Color],
     score :: Int,
     time :: Int
  } deriving Show
```

We created the datatype Player to keep track of all the variables Candy-Break needs in order to function. Every attribute is given a value at the start of a game, with the help of the **initState** function. **initState** takes help from various functions mentioned in the subsection Description of important algorithms and functions, to give every attribute a starting value. Some starting values are hardcoded and others depend on randomly generated integers.

A brief explanation to what every attribute represents and is used for:

squareLoc: This attribute represents the players coordinate. At the start of the game this attribute is set to the top left corner of the gameboard. These coordinates are calculated by using the current setting for how many boxes wide the gameboard is set to be.

playerColor: This attribute represents the players color. At the start of the game, the color is hardcoded to be white. However, if the player chooses to make a move, the players color is changed to violet until the play has been made.

squareIndex: This attribute represents the players current index within the list of squares (gameboard). At the start of the game the index is hardcoded to always start at one (the top left corner). As the game goes on, the players index changes every time the player moves. If the player decides to move right, the function **handleKeys** increases the players index by one. When the players index has increased, **render** notices that and paints the player on the square corresponding to that index.

candyBank: This attribute represents every single candies coordinates, index and color. We have made our own type Candy, which is a tuple consisting of another tuple and a color. The tuple within the tuple is the candies coordinates and its index. The color is the candies color. The candyBank attribute is a list of Candies. At the start of the game, the candyBanks candies gain coordinates and indexes with the help of the **candyLocations** function. And the candies gain colors with the help of the **randColorGen** function. Lastly, the coordinates, indexes and colors are welded together using the **createCandy** function.

moveState: This attribute represents whether or not the player currently is in a move-state or not. Every time the player presses enter, the player goes into a move-state. If the player is within a move-state, their next move is a switch-move. While performing a switch-move the player cannot move across the gameboard, they can only decide in which direction they wish to change their candy. Considering this, the players move-state at the start of the game is hardcoded to be false, since the player hasn't decided if they wish to make a move yet.

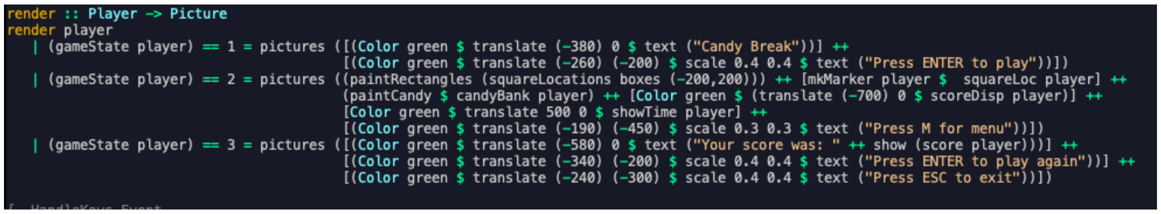
gameState: This attribute represents what game-state the game currently is in. The purpose of this attribute was to implement a splash screen at the start of the game. Pressing a specific key would then take the player out of this splash screen and into the game. We didn't have time to implement an advanced version of this but at the start of the game the player is greeted with a "Press enter to start" message that sets the gamestate to 2. this is the game-state corresponding to running the play loop (the game). The function **render** checks which gamestate the player is in and paints a picture corresponding to the current game-state.

colorBank: This attribute represents what colors the newly generated candies will have, in other words, what color the replacements for the crushed candies will have. We solved this by generating 10 000 random colors and putting them in the colorBank at the start of the game. So, if a player manages to break 10 000 candies without losing, the game will crash since it no longer knows what color to paint newly generated candies. This is however impossible as the player needs to achieve 1000 points in a hundred seconds.

score: This attribute represents how many crushes the player has managed to do. Every time a player makes a valid move and candies are crushed, the score goes up by 1. Therefore the score attribute is set to 0 at the start of the game. The function **render** paints the current score to the left of the gameboard.

3.2 Description of important algorithms and functions

3.2.1 render



```
render :: Player -> Picture
render player
   | (gameState player) == 1 = pictures ([(Color green $ translate (-380) 0 $ text ("Candy Break"))] ++
                                        [(Color green $ translate (-260) (-200) $ scale 0.4 0.4 $ text ("Press ENTER to play"))])
   | (gameState player) == 2 = pictures ((paintRectangles (squareLocations boxes (-200,200))) ++ [mkMarker player $  squareLoc player] ++
                                        (paintCandy $ candyBank player) ++ [Color green $ (translate (-700) 0 $ scoreDisp player)] ++
                                        [Color green $ translate 500 0 $ showTime player] ++
                                        [(Color green $ translate (-190) (-450) $ scale 0.3 0.3 $ text ("Press M for menu"))])
   | (gameState player) == 3 = pictures ([(Color green $ translate (-580) 0 $ text ("Your score was: " ++ show (score player)))] ++
                                        [(Color green $ translate (-340) (-200) $ scale 0.4 0.4 $ text ("Press ENTER to play again"))] ++
                                        [(Color green $ translate (-240) (-300) $ scale 0.4 0.4 $ text ("Press ESC to exit"))])
```

Basic description: Renders purpose is to,every loop, draw the gameboard. Render takes a Player as an argument, Player contains all the information render needs to draw the gameboard, that being the gameboard, the candies and the player.

Step by step description: The function consists of three guards, checking which gameState the player currently is in. The current gameState is stored within the datatype Player, which render takes as an argument.
If the player is in gameState 1, render will draw a splashscreen.
If the player is in gameState 2, render will draw rectanglepictures. If the player is in gameState 3, render will draw and endscreen. The rectanglepictures are:
The gameboard: **(paintRectangles (squareLocations boxes (-200,200)))**.
The gameboard is built-up by 200*200 pixel red rectangles. The positions of these rectangles are calculated by **squareLocations**. Every rectangle is then drawn together in the function **paintLocations** to create a single big picture containing the gameboard.
The gameboard picture is then added to the picture created by **mkMarker**. This picture is the player. **mkMarker** takes one argument, which is the players location. Using only the players location, the function draws a violet-colored rectangle at that position.
These two pictures are then added to a picture of the candies. This picture is created by the function **paintCandy** which takes one argument, being the candyBank stored within the datatype Player. The candyBank is made up of every current candy's properties, being location, index and color. Using this information, **paintCandy** creates a picture representing every candy.
And then, finally, a picture containing the players score is added to these pictures. **scoreDisp** is responsible for creating the picture containing the players current score, using the score attribute in Player.

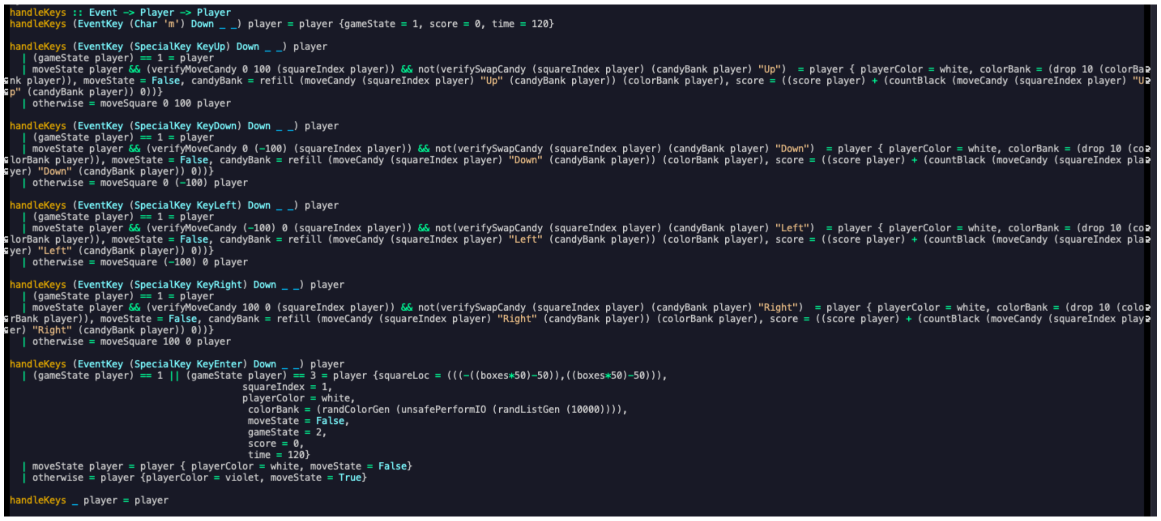
3.2.2 handleKeys

The gamestate is mainly updated through the use of player input via the arrow keys and enter key. The function **handleKeys** has various results for the same input depending on the current gamestate.

Since the arrow keys are used both for moving the player around and choosing which direction to perform a swap in, three arguments **moveState**, **verifyMoveCandy** and **verifySwapCandy** are introduced. Whenever **handleKeys** is called it checks whether or not these three arguments all have the value True and if that is the case it updates the Player datatype to contain a gamestate with the new and changed board of candy. If these criterias are not met, **handleKeys** instead updates the **squareIndex** attribute by calling the **moveSquare** function.

By pressing enter the player can also update the gamestate by changing the value of the attribute **moveState** between True and False. Pressing enter also updates the value of the attribute **playerColor** to alternate between white and violet.

Pressing any other key than the arrow keys or enter will simply return the unchanged gamestate.



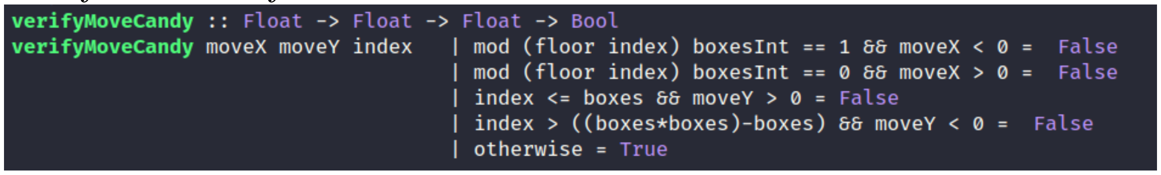
```haskell
handleKeys :: Event -> Player -> Player
handleKeys (EventKey (Char 'm') Down _ _) player = player {gameState = 1, score = 0, time = 120}

handleKeys (EventKey (SpecialKey KeyUp) Down _ _) player
  | (gameState player) == 1 = player
  | moveState player && (verifyMoveCandy 0 100 (squareIndex player)) && not(verifySwapCandy (squareIndex player) (candyBank player) "Up")  = player { playerColor = white, colorBank = (drop 10 (colorBa
nk player)), moveState = False, candyBank = refill (moveCandy (squareIndex player) "Up" (candyBank player)) (colorBank player), score = ((score player) + (countBlack (moveCandy (squareIndex player) "U
p" (candyBank player)) 0))}
  | otherwise = moveSquare 0 100 player

handleKeys (EventKey (SpecialKey KeyDown) Down _ _) player
  | (gameState player) == 1 = player
  | moveState player && (verifyMoveCandy 0 (-100) (squareIndex player)) && not(verifySwapCandy (squareIndex player) (candyBank player) "Down")  = player { playerColor = white, colorBank = (drop 10 (co
lorBank player)), moveState = False, candyBank = refill (moveCandy (squareIndex player) "Down" (candyBank player)) (colorBank player), score = ((score player) + (countBlack (moveCandy (squareIndex pla
yer) "Down" (candyBank player)) 0))}
  | otherwise = moveSquare 0 (-100) player

handleKeys (EventKey (SpecialKey KeyLeft) Down _ _) player
  | (gameState player) == 1 = player
  | moveState player && (verifyMoveCandy (-100) 0 (squareIndex player)) && not(verifySwapCandy (squareIndex player) (candyBank player) "Left")  = player { playerColor = white, colorBank = (drop 10 (co
lorBank player)), moveState = False, candyBank = refill (moveCandy (squareIndex player) "Left" (candyBank player)) (colorBank player), score = ((score player) + (countBlack (moveCandy (squareIndex pla
yer) "Left" (candyBank player)) 0))}
  | otherwise = moveSquare (-100) 0 player

handleKeys (EventKey (SpecialKey KeyRight) Down _ _) player
  | (gameState player) == 1 = player
  | moveState player && (verifyMoveCandy 100 0 (squareIndex player)) && not(verifySwapCandy (squareIndex player) (candyBank player) "Right")  = player { playerColor = white, colorBank = (drop 10 (colo
rBank player)), moveState = False, candyBank = refill (moveCandy (squareIndex player) "Right" (candyBank player)) (colorBank player), score = ((score player) + (countBlack (moveCandy (squareIndex play
er) "Right" (candyBank player)) 0))}
  | otherwise = moveSquare 100 0 player

handleKeys (EventKey (SpecialKey KeyEnter) Down _ _) player
  | (gameState player) == 1 || (gameState player) == 3 = player {squareLoc = (((-((boxes*50)-50)),((boxes*50)-50))),
                                          squareIndex = 1,
                                          playerColor = white,
                                           colorBank = (randColorGen (unsafePerformIO (randListGen (10000)))),
                                           moveState = False,
                                           gameState = 2,
                                           score = 0,
                                           time = 120}
  | moveState player = player { playerColor = white, moveState = False}
  | otherwise = player {playerColor = violet, moveState = True}

handleKeys _ player = player
```

3.2.3 verifySwapCandy and verifyMoveCandy

Basic description: Checks whether a move is valid or not. If a move is invalid it's either because the player tried to move outside the gameboard or because the move the player tried to do didn't connect three or more candies.

Step by step description: **verifySwapCandy** and **verifyMoveCandy** works together to make sure the move the player tried to do is valid or not.

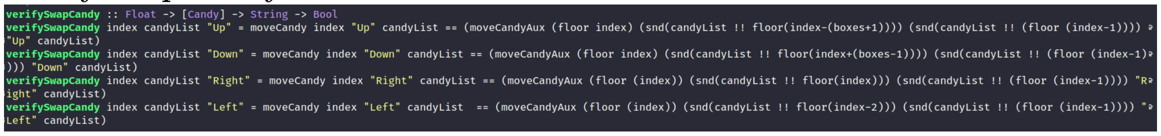
VerifyMoveCandy works like this:



```haskell
verifyMoveCandy :: Float -> Float -> Float -> Bool
verifyMoveCandy moveX moveY index   | mod (floor index) boxesInt == 1 && moveX < 0 =  False
                                    | mod (floor index) boxesInt == 0 && moveX > 0 =  False
                                    | index <= boxes && moveY > 0 = False
                                    | index > ((boxes*boxes)-boxes) && moveY < 0 =  False
                                    | otherwise = True
```

The function is built-up of guards checking whether the player is still within the gamebord after making a move.
The first guard checks if the player tries to move too far to the left
The second guard checks if the player tries to move too far to the right
The third guard checks if the player tries to move too far up
The fourth guard checks if the player tries to move too far down
If none of the criteria are met, the players move is valid. These calculations are done like following:
The first two guards use modulus to calculate if the player is on the first or last row. If the player is on the first or last row and tries to move left or right respectively, the players move is invalid.
The two last guards uses the players current index to calculate whether the player is on the first or last column, if so, and the player tries to move up or down respectively, the move is invalid.

VerifySwapCandy works like this:



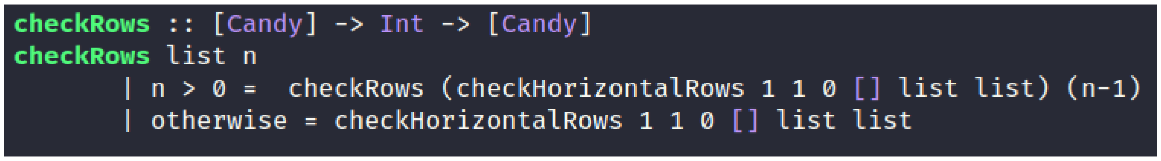
```haskell
verifySwapCandy :: Float -> [Candy] -> String -> Bool
verifySwapCandy index candyList "Up" = moveCandy index "Up" candyList == (moveCandyAux (floor index) (snd(candyList !! floor(index-(boxes+1)))) (snd(candyList !! (floor (index-1))))
"Up" candyList)
verifySwapCandy index candyList "Down" = moveCandy index "Down" candyList == (moveCandyAux (floor index) (snd(candyList !! floor(index+(boxes-1)))) (snd(candyList !! (floor (index-1)
))) "Down" candyList)
verifySwapCandy index candyList "Right" = moveCandy index "Right" candyList == (moveCandyAux (floor (index)) (snd(candyList !! floor(index))) (snd(candyList !! (floor (index-1)))) "R
ight" candyList)
verifySwapCandy index candyList "Left" = moveCandy index "Left" candyList  == (moveCandyAux (floor (index)) (snd(candyList !! floor(index-2))) (snd(candyList !! (floor (index-1)))) "
Left" candyList)
```

If a player tries to swap a candy, this function checks whether or not the swap results in at least one row with 3 or more candies of the same color. The function is built up by pattern-matching. The patterns the function matches are directions. It's purpose is to return a bool, False means that the move made is okay. This bool is calculated by comparing candyBanks (lists of candies). The current candyBank, applied to **moveCandy**, is compared to the very same candyBank applied to **moveCandyAux**. The first call will return a candyBank with every row of same-colored candies moved to the top of the playing screen. The second call will return a candyBank indistinguishable from the given one, except that two candies were swapped (swapped according to which direction the player chose). If these two candyBanks are exactly the same, the move was invalid, since this means that the move made didn't connect three same-colored candies.

3.2.4 checkRows and makeBlack

Basic description: In order for the game to be able to find vertical and horizontal rows of 3 or more candies the function **checkRows** is introduced. The sole purpose of the function is to find rows of candies and use these rows in a function that removes them.

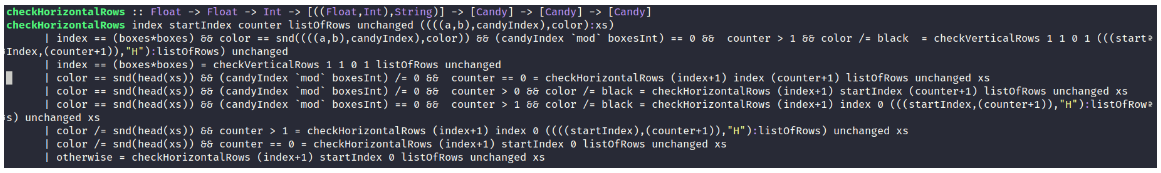
Step by step description: The function starts of by recursively calling **checkHorizontalRows** 4 or 27 times. 27 times at the start of the game and 4 times whenever the player makes a move in order to always make sure it never misses any of the possible rows that could theoretically appear.



```
checkRows :: [Candy] -> Int -> [Candy]
checkRows list n
       | n > 0 =  checkRows (checkHorizontalRows 1 1 0 [] list list) (n-1)
       | otherwise = checkHorizontalRows 1 1 0 [] list list
```

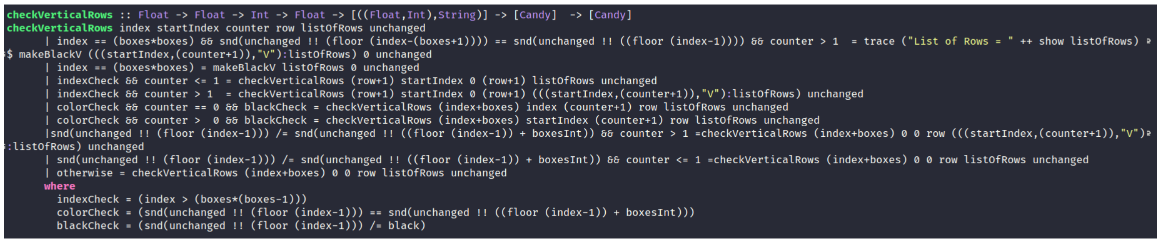
The function **checkHorizontalRows** begins with checking the first candy in the **candybank** attribute and compares its color value to the candy that is next in the list. If the value of the two color arguments match then the index is saved in the argument **startIndex** and another argument, **counter** that starts of with a value of 0 is increased by one. The function goes on to check if the third candy has a matching color value and keeps updating the counter value until the next color no longer match or the function is trying to compare the colors of two candies on different rows. This is done by resetting the counter to 0 whenever the candy being tested is on an index where (index mod boxes) == 0. In this test boxes is equal to the width of the board.

Once the code has found that the next candy has a color value that doesn't match the previous one it checks if the value of counter is greater than 2 and then adds a tuple containing the three arguments: **startIndex**, **counter** and "H" to a **listOfRows** and continues testing the remaining indexes until it reaches the end of the **candyBank** list. The "H" in this tuple is there so the program knows that the row found is horizontal.



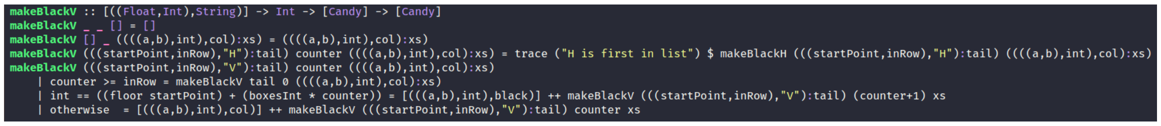
```
checkHorizontalRows :: Float -> Float -> Int -> [((Float,Int),String)] -> [Candy] -> [Candy] -> [Candy]
checkHorizontalRows index startIndex counter listOfRows unchanged ((((a,b),candyIndex),color):xs)
      | index == (boxes*boxes) && color == snd((((a,b),candyIndex),color)) && (candyIndex `mod` boxesInt) == 0 &&  counter > 1 && color /= black  = checkVerticalRows 1 1 0 1 (((start
Index,(counter+1)),"H"):listOfRows) unchanged
      | index == (boxes*boxes) = checkVerticalRows 1 1 0 1 listOfRows unchanged
      | color == snd(head(xs)) && (candyIndex `mod` boxesInt) /= 0 &&  counter == 0 = checkHorizontalRows (index+1) index (counter+1) listOfRows unchanged xs
      | color == snd(head(xs)) && (candyIndex `mod` boxesInt) /= 0 &&  counter > 0 && color /= black = checkHorizontalRows (index+1) startIndex (counter+1) listOfRows unchanged xs
      | color == snd(head(xs)) && (candyIndex `mod` boxesInt) == 0 &&  counter > 1 && color /= black = checkHorizontalRows (index+1) index 0 (((startIndex,(counter+1)),"H"):listOfRow
s) unchanged xs
      | color /= snd(head(xs)) && counter > 1 = checkHorizontalRows (index+1) index 0 ((((startIndex),(counter+1)),"H"):listOfRows) unchanged xs
      | color /= snd(head(xs)) && counter == 0 = checkHorizontalRows (index+1) startIndex 0 listOfRows unchanged xs
      | otherwise = checkHorizontalRows (index+1) startIndex 0 listOfRows unchanged xs
```

Upon testing each element in **candyBank** the function **checkVerticalRows** is called with **listOfRows**. The function performs the same tests as **checkHorizontalRows** does but with slightly different mathematical solutions as instead of always comparing candies at index n and n+1, **checkVerticalRows** needs to compare the candies at index n and n+boxes. A similar tuple as before is then added to the **listOfRows** argument but with a "V" instead of a "H" in its second slot. When both the row checking functions have been called, a list of tuples is generated with the following structure: [((startPoint,inRow),"H"), ((startPoint,inRow),"V")] if for example one vertical row and one horizontal row was detected. This list is then used in the makeBlackV and makeBlackH function that turns these rows of candies black.



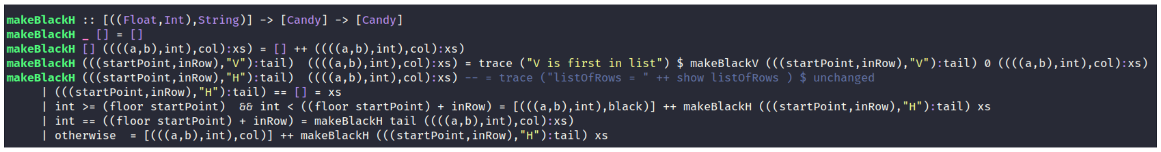
```
checkVerticalRows :: Float -> Float -> Int -> Float -> [((Float,Int),String)] -> [Candy]  -> [Candy]
checkVerticalRows index startIndex counter row listOfRows unchanged
      | index == (boxes*boxes) && snd(unchanged !! (floor (index-(boxes+1)))) == snd(unchanged !! ((floor (index-1)))) && counter > 1  = trace ("List of Rows = " ++ show listOfRows) 
$ makeBlackV (((startIndex,(counter+1)),"V"):listOfRows) 0 unchanged
      | index == (boxes*boxes) = makeBlackV listOfRows 0 unchanged
      | indexCheck && counter <= 1 = checkVerticalRows (row+1) startIndex 0 (row+1) listOfRows unchanged
      | indexCheck && counter > 1  = checkVerticalRows (row+1) startIndex 0 (row+1) (((startIndex,(counter+1)),"V"):listOfRows) unchanged
      | colorCheck && counter == 0 && blackCheck = checkVerticalRows (index+boxes) index (counter+1) row listOfRows unchanged
      | colorCheck && counter >  0 && blackCheck = checkVerticalRows (index+boxes) startIndex (counter+1) row listOfRows unchanged
      |snd(unchanged !! (floor (index-1))) /= snd(unchanged !! ((floor (index-1)) + boxesInt)) && counter > 1 =checkVerticalRows (index+boxes) 0 0 row (((startIndex,(counter+1)),"V")
:listOfRows) unchanged
      | snd(unchanged !! (floor (index-1))) /= snd(unchanged !! ((floor (index-1)) + boxesInt)) && counter <= 1 =checkVerticalRows (index+boxes) 0 0 row listOfRows unchanged
      | otherwise = checkVerticalRows (index+boxes) 0 0 row listOfRows unchanged
      where
        indexCheck = (index > (boxes*(boxes-1)))
        colorCheck = (snd(unchanged !! (floor (index-1))) == snd(unchanged !! ((floor (index-1)) + boxesInt)))
        blackCheck = (snd(unchanged !! (floor (index-1))) /= black)
```

Both makeBlackV and makeBlackH work in a similar way. makeBlackV is called first with the **listOfRows** argument and the **candyBank** attribute. The function starts of with an **index** argument with a value of 1 and if the value is equal to the **startPoint** it changes the color of the candy associated with that index to black and does the same for all candies in that vertical row whilst each time increasing the value of a **counter** argument by one. When the **counter** has the same value as the **inRow** argument the function stops changing the color to black and instead keeps testing **index** values until it finds one that once again is equal to a **startPoint** in the **listOfRows** list.



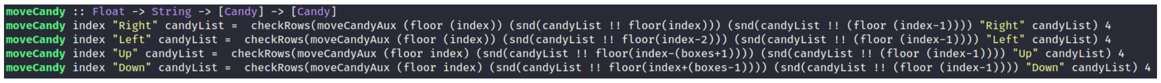
```haskell
makeBlackV :: [((Float,Int),String)] -> Int -> [Candy] -> [Candy]
makeBlackV _ _ [] = []
makeBlackV [] _ ((((a,b),int),col):xs) = ((((a,b),int),col):xs)
makeBlackV (((startPoint,inRow),"H"):tail) counter ((((a,b),int),col):xs) = trace ("H is first in list") $ makeBlackH (((startPoint,inRow),"H"):tail) ((((a,b),int),col):xs)
makeBlackV (((startPoint,inRow),"V"):tail) counter ((((a,b),int),col):xs)
    | counter >= inRow = makeBlackV tail 0 ((((a,b),int),col):xs)
    | int == ((floor startPoint) + (boxesInt * counter)) = [(((a,b),int),black)] ++ makeBlackV (((startPoint,inRow),"V"):tail) (counter+1) xs
    | otherwise  = [(((a,b),int),col)] ++ makeBlackV (((startPoint,inRow),"V"):tail) counter xs
```

If the first element in the **listOfRows** list has a ”H” as its second value, the **makeBlackH** function is called which performs a similar calculation as the **makeBlackV** function but instead of changing colors to black from the startPoint and downwards until the **counter** value is equal to the **inRows** value it changes the colors from left to right from the **startPoint**.



```haskell
makeBlackH :: [((Float,Int),String)] -> [Candy] -> [Candy]
makeBlackH _ [] = []
makeBlackH [] ((((a,b),int),col):xs) = [] ++ ((((a,b),int),col):xs)
makeBlackH (((startPoint,inRow),"V"):tail)  ((((a,b),int),col):xs) = trace ("V is first in list") $ makeBlackV (((startPoint,inRow),"V"):tail) 0 ((((a,b),int),col):xs)
makeBlackH (((startPoint,inRow),"H"):tail)  ((((a,b),int),col):xs) -- = trace ("listOfRows = " ++ show listOfRows ) $ unchanged
     | (((startPoint,inRow),"H"):tail) == [] = xs
     | int >= (floor startPoint)  && int < ((floor startPoint) + inRow) = [(((a,b),int),black)] ++ makeBlackH (((startPoint,inRow),"H"):tail) xs
     | int == ((floor startPoint) + inRow) = makeBlackH tail ((((a,b),int),col):xs)
     | otherwise  = [(((a,b),int),col)] ++ makeBlackH (((startPoint,inRow),"H"):tail) xs
```

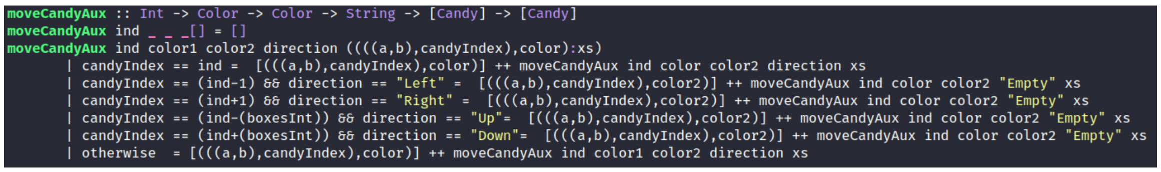
3.2.5 moveCandy



```haskell
moveCandy :: Float -> String -> [Candy] -> [Candy]
moveCandy index "Right" candyList =  checkRows(moveCandyAux (floor (index)) (snd(candyList !! floor(index))) (snd(candyList !! (floor (index-1)))) "Right" candyList) 4
moveCandy index "Left" candyList =  checkRows(moveCandyAux (floor (index)) (snd(candyList !! floor(index-2))) (snd(candyList !! (floor (index-1)))) "Left" candyList) 4
moveCandy index "Up" candyList =  checkRows(moveCandyAux (floor index) (snd(candyList !! floor(index-(boxes+1)))) (snd(candyList !! (floor (index-1)))) "Up" candyList) 4
moveCandy index "Down" candyList =  checkRows(moveCandyAux (floor index) (snd(candyList !! floor(index+(boxes-1)))) (snd(candyList !! (floor (index-1)))) "Down" candyList) 4
```

Basic description: The function swaps the color of a candy with the color of an adjacent candy depending on the direction in the given string. And with the help of **moveCandyAux** and **checkRows** figures out if any new rows have appeared. If new rows have appeared, these new rows are colored black.

Step by step description: **moveCandy** takes a direction as an argument, this direction is pattern-matched. After being pattern-matched, the function calls both **moveCandyAux** and **checkRows** in that order. So, the candies the player chose to switch are switched, and then the candies are checked by **checkRows** to, eventually if needed, change the colors of candies in a row. **moveCandyAux** works like this:

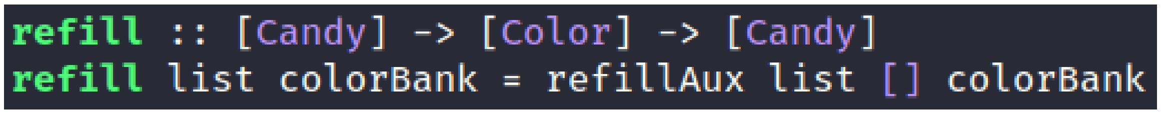


```
moveCandyAux :: Int -> Color -> Color -> String -> [Candy] -> [Candy]
moveCandyAux ind _ _ _[] = []
moveCandyAux ind color1 color2 direction ((((a,b),candyIndex),color):xs)
       | candyIndex == ind =  [(((a,b),candyIndex),color)] ++ moveCandyAux ind color color2 direction xs
       | candyIndex == (ind-1) && direction == "Left" =  [(((a,b),candyIndex),color2)] ++ moveCandyAux ind color color2 "Empty" xs
       | candyIndex == (ind+1) && direction == "Right" =  [(((a,b),candyIndex),color2)] ++ moveCandyAux ind color color2 "Empty" xs
       | candyIndex == (ind-(boxesInt)) && direction == "Up"=  [(((a,b),candyIndex),color2)] ++ moveCandyAux ind color color2 "Empty" xs
       | candyIndex == (ind+(boxesInt)) && direction == "Down"=  [(((a,b),candyIndex),color2)] ++ moveCandyAux ind color color2 "Empty" xs
       | otherwise  = [(((a,b),candyIndex),color)] ++ moveCandyAux ind color1 color2 direction xs
```

The function is built-up of several guards. These guards contain comparisons of indexes. The purpose of this is to eventually replace the 'current' candy with the candy adjacent to it. Therefore we have given every direction its own guard.
The last guard is a recursive call.

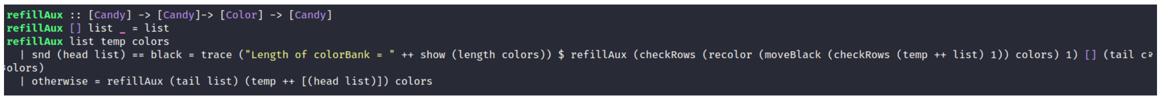
3.2.6 refill

<u>Basic description:</u> Moves all the black boxes to the top of their respective columns and gives them a new color from the list of colors that are stored in the datatype Player.



```
refill :: [Candy] -> [Color] -> [Candy]
refill list colorBank = refillAux list [] colorBank
```

<u>Step by step description:</u> **refill** takes a list of Candy and a list of Color and starts out by calling on its auxiliary function **refillAux** with the list of Candy, an empty list of Candy and the list of Colors.



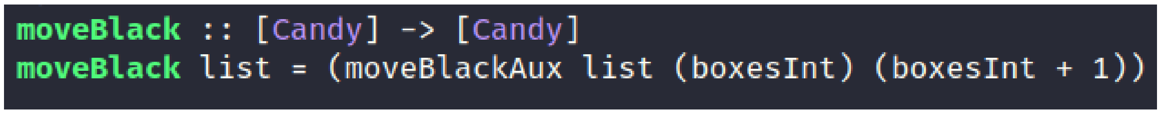
```
refillAux :: [Candy] -> [Candy]-> [Color] -> [Candy]
refillAux [] list _ = list
refillAux list temp colors
  | snd (head list) == black = trace ("Length of colorBank = " ++ show (length colors)) $ refillAux (checkRows (recolor (moveBlack (checkRows (temp ++ list) 1)) colors) 1) [] (tail c
olors)
  | otherwise = refillAux (tail list) (temp ++ [(head list)]) colors
```

refillAux recursively goes through the whole list of Candy and during every recursion checks if the first Candy in the list of Candy has the color black assigned to it. If so all the black Candy in the list gets recolored by calling the function **recolor** which goes through the whole list and generates a new color for all Candy with the color black and then the recursive call checks once again if there are three or more Candy in a row and so it keeps on calling until there are no black Candy in the list and no rows of three or more Candy of the same color. If the first Candy in the list has another color than black **refillAux** gets called with the tail of the first list and the head of the list concatenated with the second list.

3.2.7 moveBlack

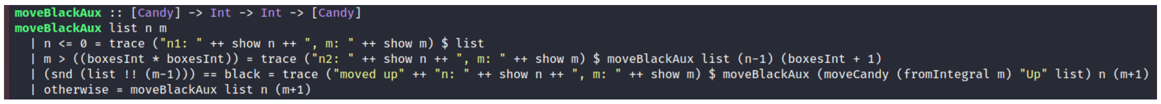
Basic description: The **moveBlack** functions purpose is to find each black square generated by the **makeBlack** function and move them all upwards one step at a time to make it seem like the black candy disappears and then candies above it fall down to take its place.

Step by step description The **moveBlack** function only has one job and that is to call the **moveBlackAux** function with the current list of candy that may or may not contain three or more black candies.



```
moveBlack :: [Candy] -> [Candy]
moveBlack list = (moveBlackAux list (boxesInt) (boxesInt + 1))
```

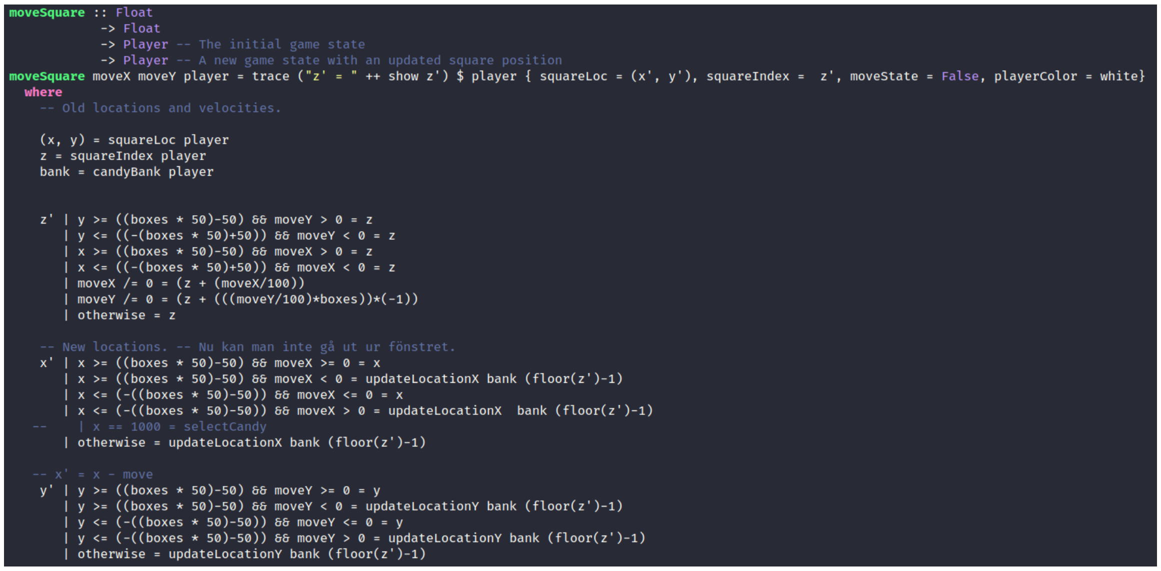
When the **moveBlackAux** function is called it starts off by looking at the top right candy to check if its second element is equal to black. If that is the case the function **moveCandy** is called with the **index** of the black candy, the string "Up" and the current **candyBank** which results in the black candy being moved up one step. The **moveBlackAux** function is then called with the new updated list with one black candy moved up. This recursive call is made until the whole list has been checked for black candies and all the candies have been moved the top where they later get a new random color via the **refill** function.



```
moveBlackAux :: [Candy] -> Int -> Int -> [Candy]
moveBlackAux list n m
  | n <= 0 = trace ("n1: " ++ show n ++ ", m: " ++ show m) $ list
  | m > ((boxesInt * boxesInt)) = trace ("n2: " ++ show n ++ ", m: " ++ show m) $ moveBlackAux list (n-1) (boxesInt + 1)
  | (snd (list !! (m-1))) == black = trace ("moved up" ++ "n: " ++ show n ++ ", m: " ++ show m) $ moveBlackAux (moveCandy (fromIntegral m) "Up" list) n (m+1)
  | otherwise = moveBlackAux list n (m+1)
```

3.2.8 moveSquare

Basic description: The purpose of **moveSquare** is to move the square where the player is located to a new location by either receiving a value of movement in x- or y-axis.



```haskell
moveSquare :: Float
            -> Float
            -> Player -- The initial game state
            -> Player -- A new game state with an updated square position
moveSquare moveX moveY player = trace ("z' = " ++ show z') $ player { squareLoc = (x', y'), squareIndex =  z', moveState = False, playerColor = white}
  where
    -- Old locations and velocities.

    (x, y) = squareLoc player
    z = squareIndex player
    bank = candyBank player


    z' | y >= ((boxes * 50)-50) && moveY > 0 = z
       | y <= ((-(boxes * 50)+50)) && moveY < 0 = z
       | x >= ((boxes * 50)-50) && moveX > 0 = z
       | x <= ((-(boxes * 50)+50)) && moveX < 0 = z
       | moveX /= 0 = (z + (moveX/100))
       | moveY /= 0 = (z + (((moveY/100)*boxes))*(-1))
       | otherwise = z

    -- New locations. -- Nu kan man inte gå ut ur fönstret.
    x' | x >= ((boxes * 50)-50) && moveX >= 0 = x
       | x >= ((boxes * 50)-50) && moveX < 0 = updateLocationX bank (floor(z')-1)
       | x <= (-((boxes * 50)-50)) && moveX <= 0 = x
       | x <= (-((boxes * 50)-50)) && moveX > 0 = updateLocationX  bank (floor(z')-1)
   --    | x == 1000 = selectCandy
       | otherwise = updateLocationX bank (floor(z')-1)

   -- x' = x - move
    y' | y >= ((boxes * 50)-50) && moveY >= 0 = y
       | y >= ((boxes * 50)-50) && moveY < 0 = updateLocationY bank (floor(z')-1)
       | y <= (-((boxes * 50)-50)) && moveY <= 0 = y
       | y <= (-((boxes * 50)-50)) && moveY > 0 = updateLocationY bank (floor(z')-1)
       | otherwise = updateLocationY bank (floor(z')-1)
```

Step by step description: The arguments that **moveSquare** takes are a Float which represent the amount of pixels to move in x-axis and a Float for the move in y-axis and also a Player which contains the previous gamestate. When **moveSquare** gets called it first decelerates the coordinates where the player is located to x and y. After that the index for that square gets declared to z and at last the list of Candy stored in player gets declared to bank. When everything is declared the function calculates the new index and also compares the x and y values so that the new box is not outside of the gameboard. When the computations are done, the function returns the datatype Player with the new values from x, y and z and also with the moveState and color of the player changed so that the next keypress gets registered as a move of the square.

4 External libraries

These external libraries were used in the code to get access to features previously not available in the standard haskell package.

4.1 Control.Monad

From the library Control.Monad, the function replicateM is used to replicate a function a specific number of times. In CandyBreak replicateM is used to create a list of random Ints.

Documentation: `http://hackage.haskell.org/package/base-4.12.0.0/docs/Control-Monad.html`

4.2 Graphics.Gloss

Gloss makes it possible to display graphics in a window. As this is a central part of CandyBreak this package got used to its full extent.

Documentation: `http://hackage.haskell.org/package/gloss`

4.3 Graphics.Gloss.Data.Pure.Game

To be able to register when a key is pressed the Graphics.Gloss.Data.Pure.Game library was used because it provides datatypes such as Event and Key that can read and handle the users input.

Documentation: `http://hackage.haskell.org/package/gloss-1.13.0.1/docs/Graphics-Gloss-Interface-Pure-Game.html`

4.4 Graphics.Gloss.Data.Picture

Graphics.Gloss.Data.Picture provides the datatype Picture that is used to create and handle figures such as a circle, text, line, or any other polygon.

This was used to create the gameboard, boxes and other graphical parts of the game and display them in the Gloss-window.

Documentation: `http://hackage.haskell.org/package/gloss-1.13.0.1/docs/Graphics-Gloss-Data-Picture.html`

4.5 System.IO.Unsafe

The System.IO.Unsafe is a special library such as it allows IO computations to be performed at any time. This is useful for creating random numbers which is complicated in Haskell outside of the IO monad.

Documentation: `http://hackage.haskell.org/package/base-4.12.0.0/docs/System-IO-Unsafe.html`

4.6 System.Random

System.Random is used to generate random numbers that are later converted into colors so that all the boxes in the gameboard have a random generated color.

Documentation: `http://hackage.haskell.org/package/random-1.1/docs/System-Random.html`

5 Discussion

5.1 Known shortcomings of the program

The program has a couple of shortcomings, some because we have created the program using functional programming and some just because we haven't had time to implement them yet.

5.1.1 Delays in animation:

A problem that is caused by the functional programming is that it's difficult to create a delay in the gameloop so that you could see the candy move and then be destroyed and replaced. If this could be implemented the game would be more enjoyable and a lot more attractive because you could actually see what is happening and not just the results.

5.1.2 Recognizing overlapping rows:

In the current version of the game it is coded to primarily always check for horizontal rows of similar candies. This results in the code removing the horizontal rows and any vertical rows that may have overlapped this row is is not counted as it no longer exists. This makes it so that the code can't detect "T" rows, or rows where a vertical row overlaps a horizontal one and the player is only given score for the first row it detects.

5.1.3 Image handling:

While trying to implements images in the game we encountered problems with importing image files into the code. Thus the candies are all represented with different colored squares and the background of the game is simply just black. If this could be changed the game would look more professional and appealing to the user.

Except for these shortcomings we still believe we have created a fully functioning game with all the features we set out to implement.

6 Conclusion

We have developed CandyBreak; our own take on a CandyCrush game using the programming language Haskell that is fully operational and runs with the help of the Gloss data package.

The games code is purely functional and performs in a way that we are very proud of given the time frame we were given to complete the task. The development of the game have vastly improved our knowledge about Haskell and our ability to work collectively with a group on a larger scale project.

We learned a lot during this project. If we ever make a similar project, we would do it quite differently. The biggest mistake we made was to not properly think through how we wanted to represent data within the game. This lead to us creating a datatype that we had to rethink completely after about a week's worth of work, unnecessary work. The next time we do anything similar to this, we would properly and thoroughly think through how to formulate and represent all the data we would need to work with, within a single new datatype. By doing this, we can effortlessly handle and ”save” all the information we need. In addition to that, adding new attributes (if necessary) is easy.

Another thing we learned during the course of this project was how to effectively use git, as well as navigating and using the terminal in an effective way.