An a-maz(e)-ing game

Program Construction and Data Structures 2020/2021

Jonas Björn, Viktor Hultsten, Tim Molnig

March 1, 2021

Contents

1 Introduction

This project aimed to extend our knowledge of functional programming in general, and the Haskell language in particular.Our fascination of graph theory and the mathematics behind mazes combined with the will of making a game made us choose to make a playable game with a graphical interface.

1.1 Summary

The outcome of the project is a mazerunner game where one is supposed to navigate a randomly generated maze to find the way to the goal. When the goal is reached your performance will be displayed with number of steps and time taken. When the game restarts it will generate a slightly bigger maze and the clock restarts.

2 Usage

2.1 How to play

To be able to play the game one need to install the ghc compiler as well as the *Haskell Gloss* package. With those installed, open the terminal and navigate into the game folder. Type **ghc -threaded PlayGame.hs**,press enter, wait for the modules to finish loading, then type **./PlayGame**, press enter. The game starts and the start screen appears.

Once in the game, one uses the arrow keys to navigate through the maze to find the way to the goal down in the left corner.

When the goal is reached a new menu appears and one can choose to continue playing, in which case a slightly bigger maze will appear, or to quit.

2.2 Example

3 Program documentation

3.1 Overview

No user input is required when starting the program. The program calls an "initial state" that is passed on through each iteration.

3.2 Flow chart of function

3.3 Modules

Dividing code inte modules combined with good naming of functions makes it easier to understand and survey for someone unfamiliar with the code.

The modules we choose to this project into are divide into are as follows.

PlayGame

The "main" module which only contains the information about the initial state of the game as well as the function which starts the whole game process.

Graphs

Functionality only regarding the generation of the maze. It uses the System.Random and IO.Unsage packages to make the randomness of the algorithm.

Render

Includes all Gloss graphical functionality and renders new frames of the game.

Move

Handles the keystrokes from the player and updates the state of the game using the Interface.Pure.Game part of the Gloss package.

GameData

Contains only the declaration of the datatype **GameState**. This was easier than to declare the datatype in one of the other modules and then export/import it around.

3.4 Data structures

```
data GameState = Game { ... }
```

"Variables" that change during gameplay is stored in a separate data type, which simplifies passing it through functions.

- *startMenu*: True if the start menu should be active. Only True when the game starts
- *goalMenu:* True if the player has reached the goal and indicates that the goal menu should be active
- *gridSize:* The width/height of the grid. gridSize = 10 is a 10x10 grid
- *mazePicture:* The rendered maze based on a list of coordinates
- *walls:* The maze in a list, where each element is a wall between two cells in the grid
- *playerCoords:* The cartesian coordinates of the player
- *playerLevel:* The player level
- *goalCoords:* The cartesian coordinates of the goal
- *steps:* The number of steps taken from the start position.
- *testImageP:* The rendered picture of the player icon

- *testImageG:* The rendered picture of the goal icon
- *seconds:* The number of seconds passed since the game started

Coordinates When the player and goal target is rendered on the screen, the actual Gloss Window coordinates vary depending on the grid size of the maze. This could cause rounding errors which, to the human eye, would not be noticable. Though, to the computer, the numbers differ and the player would not be able to "reach" the goal. Therefore, the player and goal coordinates are cartesian, which leads to a discrete comparison.

Additional information about the mazePicture and walls: The maze is stored in a similar way, with a graphical pre-rendered picture and a list of cartesian coordinates. The graphical picture is only rendered when the player hits the play button instead of re-rendering a static picture each frame.

```
type Cell = (Float, Float)
```

The position of a cell in cartesian coordinates, starting from top left.

```
type Wall = (Cell, Cell)
```

The position of a wall between two cells.

```
type Maze = [Wall]
```

The complete maze where each element is a wall between two cells.

```
newtype Stack a = StackImpl [a]
```

This is a stack type.

3.5 Graph theory

A maze can be though of as a connected graph where the edges are possible places for a wall and a face is part of the paths through the maze. The generation of a maze then consists of an algorithm deleting certain edges from the graph (essentially deleting walls and binding together paths) and different such algorithms vary in complexity and the level of difficulty the later generated maze will have.

Depth-first-search

Depth-first-search is a graph traversal algorithm which looks for paths as long as possible down the graph before backtracking. citeDFS

Maze generation

The algorithm we chose to implement uses a randomized depth-first-search approach where the algorithm is given a grid of cells, and a random starting cell in the grid. Then, one of the four walls surrounding the cell is deleted randomly. Then the cell i considered visited and pushed to a stack. A cell neighbouring an already visited cell is then chosen randomly and the same process is repeated with a random wall being deleted, and the cell marked as visited and pushed to the stack. When a cell with no unvisited neighbours is picked, we mark it visited and pop a cell from the stack and start the algorith over. Once the stack is empty, every cell in the grid is in the path of the maze.

For this algorithm we needed an implementation of a stack. We used the one which was given to us during one of the labs in the course.

3.6 Important functions

handleKeys och validMove

```
handleKeys :: Event -> GameState -> GameState
```

HandleKeys is used to give inputs to the game. It tracks arrowkeys during gameplay and spacebar in start and goal menu.

When spacebar is pressed in gameMenu or goalMenu it will promt the game to initiate the generation of a new level.

When a arrowkey is pressed in gameMode it will try to change the position of player. When this happens it takes the current coordinates of the player and creats a new coordinat in acordance with the direction the player like to move. It sends this information with gridSize and maze list to see if it is a valid move. If it is a valid move the player posistion will be updated to the new coordinates.

```
validMove :: Cell -> Cell -> Float -> Maze -> Bool
```

ValidMove is a functon that checks if a move is a valid move or not. It simply takes the current position cell and checks if the new position cell is not seperated by a wall. Furthermore it checks if the player will not cross outer bounds of the maze.

It has one special case built in and it is when the game starts the player is outside the maze and the only valid move will be into the maze.

Render

```
render :: GameState -> Picture
```

Render presents some of the GameData as a picture to the player. The picture is made with help from functions in Gloss. It has three diffrent screens and they are startMenu, goalMenu and gamePlay. Each screen will present the relevant GameData in a clear way.

StartMenu prints out a welcome message and a promt to press "spacebar" to start game or "ESC" to quit.

GoalMenu prints out information about how long and how many steps it took to solve the maze. It also has the same promt to press "spacebar" to start next level or press "ESC" to quit.

GamePlay prints out the maze, goal and level, these values will not change. GamePlay will also print out the current player position, timer and numbers of steps.

```
drawWalls :: Maze -> Float -> [Picture]
```

Takes a list of walls and a grid size as arguments and returns a list of pictures ready to be rendered to the screen. Each element is translated into a line with "Gloss Window" coordinates.

x_ mid and *y_ mid* represents the center point between the two cells in the element, which means the line should go half the length of the cell in each direction, depending on if the adjacent cells are horizontal or vertical.

3.7 External Libraries

Here we describe the external libraries that we have used to implement special functionality.

Graphics.Gloss

Gloss is one of the most used graphics modules built for Haskell. We used it to render the graphical game window showing the maze, the player and stats.

Graphics.Gloss.Interface.Pure.Game

This extension of the Gloss package holds the functionality that handles user input, i.e. pressing the arrow keys to move the player.

System.Random

This package helps to generate pseduo-random numbers in the Haskell environment. In the maze generation algorithm we used this to randomly choose alternatives for the paths in the maze.[1]

System.IO.Unsafe

This we had to use as a result of indecisive choices implementing the random functionality. The type of using any random function in System.Random will always be IO a and we needed to get rid of the IO part.[2]

Test.HUnit

A library used to create simple test which can be autimatically run, both alone and in groups. We used it create the test which can be found on rows xxx-xxxx in **PlayGame.hs**. [3]

4 Discussion

4.1 Shortcomings

The use of the IO.Unsafe module was needed due to a workaround. We started building the other parts of the algorithm generating the maze before creating the *randomness* which meant that the graph algorithm wanted to get the type Int and not IO Int which is returned from the functions in system.random. Therefore we chose to solve the problem by combining it with the IO.unsafe module to get the returned type to be of type Int. It is not statistically random, however there are no re-occurring patterns and that is the important part.

4.2 Conclusion

Throughout working with the project maybe the biggest problem that we stumbled upon was to how to figure out how to deal with the anti-state and lack of side effects in Haskell. All three members of the group hade some varying experience from object-oriented programming and that combined with the fact that we during the first part of the project had not really grasped the most important fundamentals of Haskell made for some annoying problems. What we in the beginning thought of being "variables" (e.g. size = 10 :: Int) were of course FUNCTIONS and later on when we tried to make the game restart with new , which we of course became aware of not only are properites of Haskell in particular, but of functional programming in general.

Another struggle throughout the proejct were our fights with the Gloss package and how the abscence of documentation and a great community made our work much harder. We found a few examples of how other people had used Gloss for similar projects as ours but our way to the final product still existed of an extensive amount of trial and error.

Furthermore, we changed the representation of GameData numerous times during the project.

References

[1] Haddock. Systemrandom.

[2] Haddock. Unsafe io operations.

[3] Haddock. Hunit: A unit testing framework for haskell.