Trinity College Dublin
Coláiste na Tríonóide, Baile Átha Cliath
The University of Dublin

CSU44012 Topics in Functional Programming

Minesweeper-o-Matic

Prathamesh Sai
4th year Integrated Computer Science
Student ID: 19314123

School of Computer Science and Statistics,
Trinity College Dublin

1 Introduction

In this assignment, I have developed an implementation of Minesweeper which also provides the option of "auto-playing" based on advanced tactics to win the game. A demo of the project is attached with this submission (*Minesweeper-Demo.mov*) and can also be found on YouTube here. This PDF and the comments in my code serve as documentation to show how I have implemented the project, but this PDF also includes my reflection on the project. Instructions on building the Stack project are inside the `README.md` file.

2 High level design choices

To build this project, I had to make reasonable high level design choices so that I could implement the project to a high degree. These are listed below under each file name for simplicity:

2.1 Grid.hs

Firstly, I had to display a grid on which the game could be played. I found out very quickly that there were many pieces of information that a grid needed before I could draw it. In the end, I boiled these down to width (*width of the Minesweeper grid*), height (*height of the Minesweeper grid*), amountOfMines (*number of mines within the grid*), squares (*all the squares within the grid*), squareStatus (*the status of every square in the grid*) and minesweeperStatus (*the status of the entire game*); these properties were held within a Grid data type. This was very handy to do, as otherwise I would have had to pass each piece of information to each function and it would add a layer of complexity for no reason. In addition, while working with these values I decided to change the Grid to a record type to have easy access to each for *setting* or *getting* values so it could simply be done as *"grid {minesweeperStatus = Running}"* for example.

A grid is most intuitively made as a 2D square with smaller squares inside of it, as you can simply use coordinates to differentiate different squares such as (x_1, y_1) for one square and (x_2, y_2) for another. Initially I was going to use a list but this proved to be troublesome as I had to deal with issues such as complex indexing and traversing the list by iterating. I decided to use vectors not only because it was more intuitive but also because the time complexity of accessing elements in a list (implemented as a linked list in Haskell) is $\theta(n)$ where n is the number of elements in the list. On the other hand, it's only $\theta(1)$ using vectors (which is implemented as a contiguous array in Haskell). Furthermore, I created data types for MinesweeperStatus and StateOfSquare since I found out through the development process that they are used quite a bit, so it made things a lot easier to make them into data types. During phase 1, I initially just used ASCII for the squares and after I had implemented the Grid enough, I knew it was time to start on the UI to further develop the project.

2.2 Minesweeper.hs

After the technical details for the grid was done, I had to implement the actual UI design of Minesweeper using Threepenny GUI. I decided to use *IORefs* to store and manage the state of the game instead of unnecessarily complex alternatives. I found that there are generally 3 levels of difficulty in Minesweeper - "Beginner", "Intermediate", and "Expert" which have dimensions of 8x8, 16x16 and 16x31 respectively. I decided to change the dimension of ”Expert” to 31x31 to keep the ratio of the grid as a square and also to make it more difficult. I added 3 buttons for these three levels of difficulty, and the idea was that pressing each would show different types of grids for each difficulty level. To show the grid I drew a *Canvas* to represent the grid as it was easy to use and familiar from lectures. I applied CSS for both the buttons as well as the background of the game.

After a user chooses a button for the difficulty level, the current elements on the page are deleted and new elements including a canvas are added. I decided that I needed 6 buttons for interacting with the game. The first 3 would be the mine button, flag button, and the question button (*if you're unsure about a particular square*). These options are displayed on the right of the canvas. When any of these are selected, the mode is saved and used whenever a user clicks on the canvas so that we can apply the correct type of input to the canvas. To draw the canvas, lines are drawn with a set width and height apart from each other so that they form the required grid. Afterwards, each square is filled with a number representing the number of mines next to it which are colored differently based on the original Minesweeper game. If a square is pressed (turning it from ”Unmined” to ”Mined”), only redrawing the square that was pressed is intuitive but I found it difficult to implement as it often led to glitches in the other squares. I resorted to redrawing the entire updated canvas every time one square is clicked and it still seemed to work well. The last 3 buttons are the back button *(to go back to the initial page)*, the auto-play button, and the refresh game button. The back button and refresh button were simple to implement as it was just a case of either refreshing the page or redrawing the grid from scratch. The use of *IORefs* was crucial to the implementation of the UI as it let me track various pieces of information such as the game mode or if the game had started already etc. Using the UI Monad allowed me to implement the UI and it had the benefit of letting me implement the game so it follows a particular order of steps such as updating the grid, evaluating the grid with new additions of mined squares, and finally redrawing the canvas; I also used *liftIO* to use *IORefs* with the UI monad. However, by this point I realised that the code was getting too long after multiple changes and additions so I made another utility file to contain some of the code.

2.3 UserInterfaceUtility.hs

As the code inside `Minesweeper.hs` got too clunky, I made the functions as short as possible and added some functions with a specific purpose into `UserInterfaceUtility.hs` to act as a utility file that can be imported for use in other files. I stored values such as the hard-coded width and height of the canvas (600px each) in here to avoid confusion in other files. Also, functions such as drawing the contents of the squares were contained in this utility file. I decided on using emojis for each of the buttons when the game is running to make the application appealing and fun to look at while also adding a GIF as the background for the game with CSS to keep users engaged.

2.4 Main.hs

This file contained the top level of the program, and calls `Minesweeper.hs` when the game is going to begin so that the grid is drawn. The reason for splitting up the code into 5 different files is that it separates different functionality very nicely. Following the concept of *separation of concerns* was ideal and worked out well this way. This file mainly passes in parameters to functions in `Minesweeper.hs` such as the number of mines (which is 10, 40 and 99 respectively for beginner, intermediate and expert modes of the game) and the number of rows and columns required for the grid.

2.5 Autoplayer.hs

The final task was implementing the auto-player and this was probably the most difficult task. Implementing an auto-player that plays randomly was easy by generating pseudo random numbers, so I did this and had it as a backup option. I wanted the auto-player to be separated from the code that is for a user's own input so I only exported the function `autoPlay` to use in `Minesweeper.hs` when the auto-play button is clicked. Although I had the auto-player choosing squares randomly as a backup, I was able to implement 2 advanced tactics to make the auto-player which I learnt from the Minesweeper Wiki. The first strategy is that if the value of a square is equal to the actual number of nearby flags, then the remaining nearby squares have no mines. With this in mind, one could mine those squares with no fear that they could be dangerous. This strategy was executed by iterating over each square and was implemented in a way such that it would use the first square that it could use this pattern with.

The second strategy is that if the value of a square is equal to the number of unmined squares nearby, then they are all mines. This could be used to flag certain squares and therefore allow the auto-player to recognise that it should not click on these. This strategy was fairly efficient and worked the majority of the time. Having strategy 1 for mining squares and strategy 2 for knowing which squares to flag was really helpful as it balanced out the auto-player from doing too much of either one. Using these 2 strategies as the forefront of the auto-player and resorting to choosing randomly if they could not be applied was an efficient strategy overall so I decided to go with this. In the end, this implementation of the auto-player was pretty good. From my tests, it seems to win on average about 6-7 times out of 10 runs. I was pretty happy with this result as it was fairly difficult to implement the auto-player to begin with since Minesweeper is NP-complete.

3 Reflection

I am very happy with the outcome of the project since I was able to satisfactorily provide all project deliverables. The final result is a game that is fairly quick, responsive to the users needs and is designed in a user friendly manner. The auto-player uses advanced tactics to try to make the safest move possible and seems to win the majority of the time. The use of various files for differentiating various functionality makes the project easy to understand and make changes to as well. However, there are some intricacies that are important to mention - since we are using Haskell, what benefits did the language provide and how did the software development process go?

3.1 How suitable was Haskell as a language for tackling each phase of the project?

Haskell was a great language for tackling both phase 1 and phase 2 of the project. To start off with phase 1, I was mainly dealing with the technical details of implementing Minesweeper. Using the functional aspect of Haskell to divide these technical details into functions (*and subsequently those into even more functions*) was a great way to get started on phase 1. I found myself making functions for other functions, and realising that they could be used elsewhere with little to no changes for minimal extra programming effort; modularisation in this way was really helpful. For example the function `drawSquare` is used in `getListOfSquaresAndDraw` which is subsequently used in `updateCanvas` in `UserInterfaceUtility.hs`.

3.2 What was my experience of the software development process (including things like testing and debugging)?

The software development process was fairly smooth. Any issues I had were manageable with the constant reminder to structure my thoughts in a functional manner (*although I found myself frequently going back to an imperative way of thinking by accident and having to remind myself not to*). If I faced any issues with a function returning a value that I did not expect, usually Haskell caught these errors beforehand as it was usually type related (*i.e. accidentally returning (Int, Int) instead of IO (Int, Int)*). If Haskell wasn't able to catch these, I was able to copy the code into another Haskell file and try to debug by either using print statements or using an extension on Visual Studio Code called ”phoityne-vscode”. Doing this usually led me down a path of making sure I really understood everything in each function. Simple mistakes such as using fmap instead of map was usually the issue meaning it was easy enough to fix with time and effort.

3.3 What features of the language proved useful?

As mentioned in 3.1, one main feature that proved useful was the functional aspect of Haskell but another one is laziness. Laziness in Haskell was a big help since it speeds up the program by only evaluating what is absolutely necessary. This was helpful in phase 2 when I was building the auto-player. When it was looking through any potential moves, laziness allows the auto-player to only spend time looking for the first move that is actually allowed in Minesweeper instead of doing multiple moves at a time. Finally, I felt like Haskell was definitely satisfactory for things like updating the HTML DOM (*document object model*) by removing elements, running the game on localhost with graphics using Threepenny, and also maintaining and evaluating the state of the game with *IORefs* and more. Using features such as pattern patching with guards along with the wide range of libraries offered by Haskell made it a language that was very suitable for implementing Minesweeper during both phases.

4 Minesweeper

You can learn more about Minesweeper at https://freeminesweeper.org/how-to-play-minesweeper.php and on the Wiki at http://www.minesweeper.info/wiki/Strategy. You can view the demo of this project on YouTube at https://youtu.be/NZ9gfJnvIgU.