README: *Game of Life*

Bayard Walsh

October 2023

1 Introduction

For the final project, I built a recreation of Conway's Game of Life, through applying random generative States, Parsing, and more Haskell functions. A key feature in this project is the `Brick Library` [4], which is a fantastic Haskell library that manages the UI for the board within states.

2 Github Files

2.1 GameLife.hs

This file is the primary driver for the program. First, it defines the type `CordHashMap` which stores string (an "X", for an alive cell) at (Int, Int), for the cords on the board. Every permutation of (0,0) to (X, Y) is the board space, and the set of values in (X, Y) are the alive cells. Next we have `InteriorST` and Appstate, a type alias and an app type for `Brick IO`, which wrap the same data: `(CordHashMap, StdGen,(Int, Int), Int,[(Int, Int)])`, which are the map, the generator for random population, the size of the board, density of board population, and (dead, alive) stats for each turn on the board.

Next, we have DrawUI which implements a visual representation of the board as a `Brick Widget` by checking all permutations of (0,0) to (X, Y) and adds an appropriate dead or alive cell. The apparent function handles key presses, calls the function to exit the program, generates a random number of cells at the given density, sets the grid to empty, sets the grid to full, or makes the next step in the game according to the Conway rules. Note that a final call to `epochStep` is made at the q key press as this will include the final screen displayed in the end-game stat calculation.

`genRandomSquares` recursively adds new cells to the board based on a random seed. If a cell exists at that coordinate then the function continues until it finds a free spot, until the density of new cells has been added, when the function returns the modified `InteriorST`. Randomness is handled through the generator in `InteriorST`, and then using two `RandomR` function calls for (X, Y). If the number of random cells leads to a full board, then another function is called to populate the entire board, preventing an infinite loop of searching for a random

spot. The second main board functionality call is `epochStep`, which updates a `InteriorST` to the next `InteriorST`, based on the cells around each cell. Note that each cell is updated together, so the initial `InteriorST` hashmap is used to consider if every cell on the board is alive or not, based on the `killOrNot` function, which implements the rules of the initial Conway Game of Life [3]. This function is folded onto an empty hashmap, which is populated to become the next state. The tuple of (alive, dead) cells is added to a list used to calculate the end-game stats.

Next, we handle IO management helper functions for the MVC loop. We have a string for the beginning and end of the game. The beginning string explains the general idea of Conway's Game of Life and the usage of specific key presses the player can use once the board has been initiated. The end state shows a brief message that calculates various stats from the board. Note that a *round* is defined as stepping to a new state through the Conway rules, so generating or wiping the board are not considered rounds and are not considered in the end statistics. This function takes [(dead, alive)] cells from each round to generate stats. Next some IO functions are used to prompt the player to enter inputs so the player can input in configurations to decide various aspects of the model. These include loading PreSet shapes (discussed more in `PresetParser.hs`), generating a baseline random distribution for a board, or configuring the board shape and distribution manually. The user can also use `Q` to terminate the program early. To set a specific Preset, the user is prompted with 5 options tied to different keys, which load string corresponding to `Run Length Encoded` strings which are then parsed into a hashmap seed.

The main loop is simple; it generates an initial `StdGen`, displays the welcoming text, and then kicks off the `getConfigs` function to guide the user through the IO game setup. Next, the game starts and continues until the user presses `Q` to quit. The closing text is displayed and then the function terminates.

2.2 PresetParser.hs

This file contains the Parsing logic for analyzing `Run Length Encoded` strings. The design uses the data type PreSet, which contains (X, Y) for the overall board size, and a tuple of a hash map, a string (current parsing section, which is updated as the parser recursively calls itself) and another (X, Y) which indicates the current position on the board. `presSetParser` extracts the required board size from the string (as this varies between seeds) and then stores them in the first tuple. Next, it kicks off `parseSubstring`, which adds alive cells to the hash map based on "o" characters and will update its current position based on "b" characters. Following the formula of Run Length Encoding [1], if there is a number before an "o" or a "b", that indicates a continuous run of those cells along that row, which the parser uses a simple foldl function to apply to add cells and to maintain its spot for the next parsing case. $ indicates a new row, and ! indicates the end of the seed, which is also implemented in the parser. After completing parsing, the first (Int, Int) tuple and the populated hashmap are passed to `GameLife.hs` with a density of 25 percent and are then treated

the same as a randomly generated board from there. Note that these files are designed to step forward without new random cells to generate an interesting pattern, though the user can populate or kill all cells once the patterns have been loaded.

2.3 PreSetSeeds.hs

This file contains several popular seed maps for Conway's Game of Life that were created by the online community. I recreated five of them following the `Run Length Encoded` parsing form that is used on the LifeWiki, and this file contains those 5 encoded strings to be loaded by the parser.

2.4 State.hs

This file implements the `State` type from class [2]. It is unchanged and uses the implementation mentioned in lecture for creating random states to generate positions on the board.

3 Notes

3.1 Cabal

The cabal file is a standard build file including the necessary libraries for `Brick` and other imports. Build the game through `cabal build`, then run the game through `cabal run game-of-life`. From there enter text in the command line for control.

Unlike some Games of Life, the edge of the board is a null space, so shapes will move off the board and then die. Edge null spaces are calculated as unpopulated cells in the implementation but can't be populated themselves.

Another choice that I made was to keep an informal representation of board values in a Hash Map, rather than in a list of lists. Initially, I thought this would be a more efficient decision for larger boards, however, I realized that my bottleneck is limited by the `Brick` visualization of the board, instead of the update run time. However, the hash map did make parsing easier, as the majority of the shapes had far fewer alive cells than the overall board size, meaning an entire board could be stored in a short string, and both the hashmap and parser ignored dead cells at the end of a row.

I focused on adding more user input to the Game of Life; there was something off about a 'zero-player game' and I thought that giving the player control over the initial configurations, manual stepping between states, and letting them populate or kill the board at whim would add a new dimension to the otherwise standard simulation.

An implementation area where I struggled was attempting to add the feature to give the user the option to manually set a given cell to either dead or alive. I initially thought this would be easy, however, I realized that entering coordinates through the command line would be too clunky, so I would have to use the mouse

to switch cells. However, because the board size is dynamic (and I shrink the terminal for the larger boards) the terminal positions don't always align to the same cells, so mapping the clicks to a given cell became a far harder task than initially expected, though this could be developed given more time.

References

[1] Run length encoded. https://conwaylife.com/wiki/RunLengthEncoded, 2023.

[2] Ravi Chugh. State.hs. https://www.classes.cs.uchicago.edu/archive/2023/fall/22300-1/notes/state/State.hs, 2023.

[3] John Horton Conway. Conway's game of life, 1970.

[4] Jonathan Daugherty. brick. https://github.com/jtdaugherty/brick, 2023.