Technical Report
COMP1100 Assignment 2

Jacob Bos
ANU u7469354

May 3, 2022

Lab: Tuesday 11am

Tutor: Abhaas Goyal

Word-count beyond cover page at $\leq$ 1250 words

Contents

1 Introduction

The program detailed in this report is an implementation of Erik Fransson's *QR World* cellular automata with a graphical representation in Haskell. The automata is contained within a module called `Automata` with user input handling in module `App` and graphical output handled in `GridRenderer`. Testing is handled by three modules with unit tests within `AutomataTest`.

2 Documentation

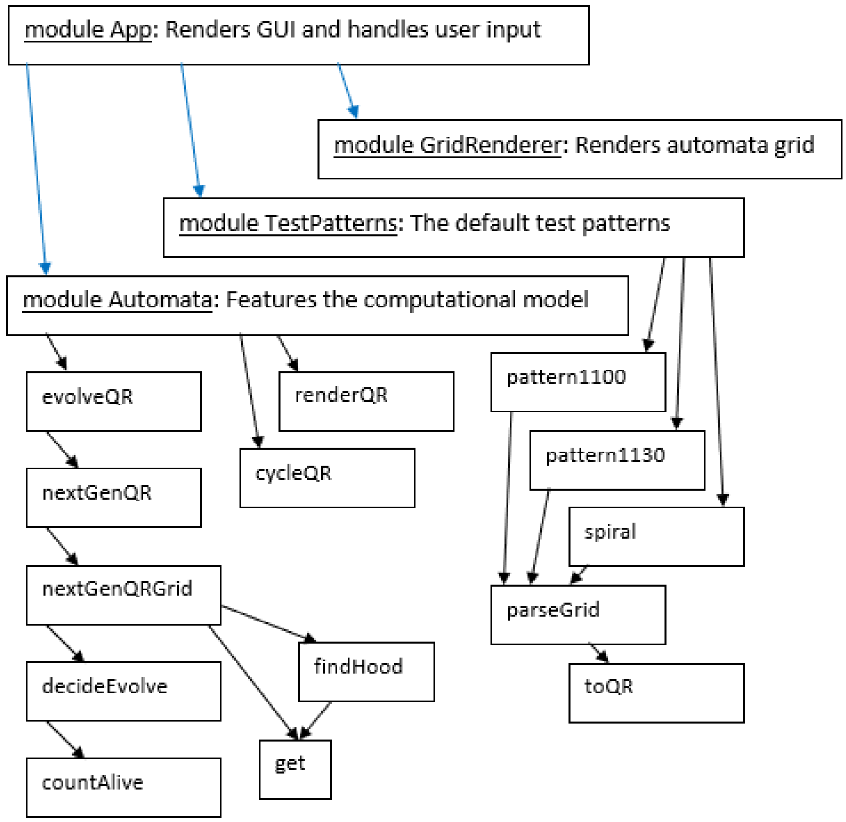
2.1 Design Documentation and Technical Decisions

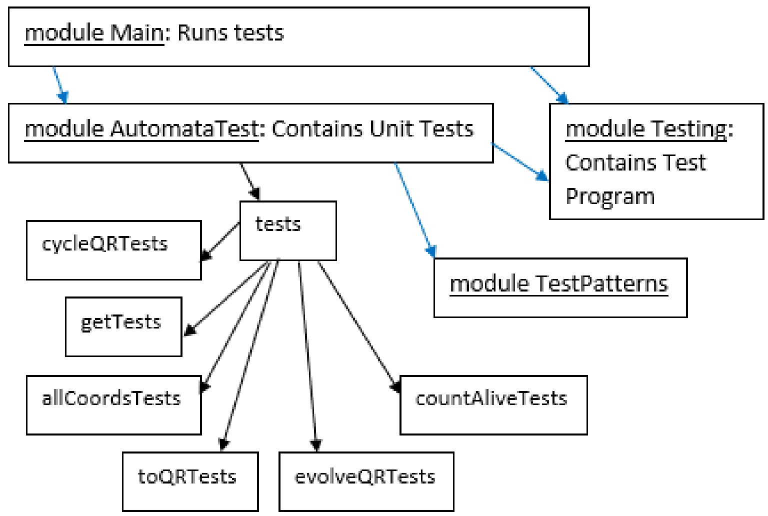
Task 1 consists of 5 functions. The type of `QRCell` is defined as either if the values either `Dead` or `Alive` an algebraic type was chosen as it was more descriptive of the program's meaning than just boolean values. An if then else (ITE) statement was used for `toQR` to convert values in the textual representation to useful values with `'A'` to `Alive::QRCell` and any other characters (else) as `Dead`. `cycleQR` swaps the value of a cell upon cursor clicks using a case statement. If `type QRCell = Bool` then the function could just be `not`. `renderQR` renders each cell as the specified codeWorld picture using a piecewise case definition for style. `get` retrieves the value of the model at a given `GridCoord`. Guards were chosen as they can protect against retrieving elements outside the automata grid and just return `Nothing` for nonsensical arguments. Elsewhere it just retrieves the appropriate element of the model list. `allCoords` generates a row-major list of all grid coordinates in an $a \times b$ for $a, b > 0$ grid and is broken up into helper functions for ease of understanding. It returns an error for nonsensical arguments of $a, b \ngtr 0$ as par specifications. Otherwise it calls 3 helper functions. `nList` generates an ascending list from 0 to (a-1). `nPair` then pairs each value in the `nList` with some integer. `allPairs` then does this to create one list from (0,0) to (a-1,b-1).

Task 2 consists of two primary functions, `nextGenQR` which parses the automata through one iteration, and `evolveQR` which iterates the automata through n iterations. `nextGenQR` calls the helper functions `allCoords` and `nextGenQrGrid` which is the main function handling the evolution of the grid. `nextGenQrGrid` recurses through the `allCoords` list using `get` to retrieve the state at each position and the helper `findHood` to retrieve a list of the states of the four neighbors. The helper `decideEvolve` then chooses updates the state of the cell according to the QRWorld rules. This new state is then prepended to the recursive call on the rest of the list. Guard based recursion with the (++) operation was chosen for `allPairs` and `nList` to get the lists in ascending order. Case based (:) recursion was used for `nPair` to maintain the order of the input list. `allCoords` is guarded to return an error for nonsensical grid dimensions to avoid irrational program operation. A guarded case recursion was chosen for `countAlive` to only count specific elements. `decideEvolve` was chosen to use a case to direct the function to guards based on the number of alive neighbours determined by `countAlive` to then decide how to change the state. It was chosen for `nextGenQR` to call `nextGenQrGrid` so that the helper functions could be called appropriately and allow for a recursion through the list of `allCoords`. `findHood` uses `get` to retrieve a four element list of `[Maybe QRCell]` to give the states of the neighbouring cells. `decideEvolve` then calls `countAlive` to then make a decision about what each cell state should evolve to depending on how many alive neighbours it has. To do this it cases on the state of the cell and is then guarded by the number of alives to evolve the state properly. `countAlive` just uses a case and nested guard recursion to sort through the list of neighboring states and returns the number that are alive. `evolveQR` recurses down to a base of 1 from a natural n applying `nextGenQR` to itself n times.

2.2 Structure

The program and the test program have module and function dependencies according to the following graph:





2.3 Assumptions

Whilst writing `get` it was assumed that attempts to retrieve a point outside the grid should return `Nothing :: Maybe QRCell` as doing so eased implementation of `findHood` and `countAlive` far more than returning an error would. It was initially assumed that nonsensical inputs to `allCoords` should return an empty list but this was revised to return an error as specified.

3 Testing

Unit tests were divided into 6 groups: cycleQRTests, getTests, allCoordsTest, toQRTests, evolveQRTests and countAliveTests. Each test group tests a particular function or group of functions. `cycleQRTests` is a fully comprehensive test group for `cycleQR` indicating its correctness. `toQR` is tested against tests two typical cases and an edge case. The tests for `get` cover most possible edge cases and also some typical cases as documented in the `AutomataTest` file. The tests for `allCoords` covers some expected inputs to both the main function and helpers. There were no edge case tests written as such cases are written to return an error and there was no provision to test if an error is returned.

`evolveQR` was tested with three unit tests which in turn test `nextGenQR` due to their dependencies. The first two tests tested if the 1100 pattern would get to an alternating steady state after 12 evolutions and the third if the 1130 pattern reached steady state eventually as specified. All tests passed indicating program correctness. All these tests are documented with comments in `AutomataTest.hs`

GUI tests focussed on the behaviour of functions that the user directly interacts with. `get` and `cycleQR` were tested by clicking cells with various states and checking if the appropriate cell switched state. `decideEvolve` was tested by observing the evolution of a single cell in various neighbourhoods compared against the rules of QRWorld. All tests passed indicating program correctness.

4 Reflection

Development of the program followed a linear process parallel to order of assignment specifications. Design decisions were made with both functionality and style in focus to make proper use of Haskell's recursive propensity. Consequently, program is quite readable especially due to its effective documentation comments. Many of the design decisions are detailed in Section 1.1. Generally ITE statements were used whenever it was necessary to only check for one case and anything else returning the same answer. As detailed, guard and case recursion was selected based on whether the function needed to iterate n times or traverse a list. When iterating n times (++) was used to order the list properly as n counts down but the list counts up. When traversing a list (:) was used to maintain the input ordering.

If they were to re-develop the program the author believes they could rewrite or remove the helper `nPair` by using `map` and the anonymous function `(\y x -> (x,y))`. However, they would make no changes to the structure which was dictated largely by the specifications.

The program was within the authors technical abilities and so there were no significant problems encountered however, the author suspects that there is a better way to write `allCoords` as it is rather convoluted but other than the above mentioned, inspiration escapes them. The author had some trouble defining the type `QRCell` but came to a good solution under closer reading if the specifications which allowed the rest of the program to develop smoothly. Consequently, they did not need to collaborate with others in any significant way.