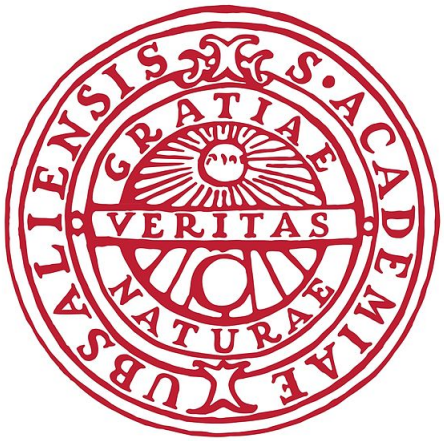

UPPSALA UNIVERSITET

Haskell Project Rubik's Cube

Group 33: Daniel Cada, Filip Eriksson, Isac Arrhenius

PkD February 2021

Contents

1 Introduction

This program is a representation of a Rubik's Cube. It displays a grid representing the current state of the front of the cube whereas the user will specify what line or row they wish to rotate, or if they want to change their perspective. The cells in the grid will be updated with their respective colour. When the user has completed the cube, the program detects it and returns a victory screen.

The program is divided into four files, each handling a different part of the program. The files are "Main", "Rubiks" , "Changer" and "Displays". The file Rubik's contains all the core functions of our Rubik's cube and was the only file we used before implementing the gloss library. The files "Displays" and "Changer" handles the display of the program and how the display changes with new inputs respectively. The "Main" contains the startup and functions as a "bridge" between the other files.

2 Use Cases

The contents in this section instructs how to use the program and what software is required to execute it.

2.1 Software requirements

Firstly to use the program the user needs the Haskell ghc compiler. Secondly since the program uses the Haskell Gloss Library and the Random library, they both need to be installed along with cabal, cabal handles the different packages to correctly execute the program.

2.2 Program flow

The program starts by scrambling a completed Rubik's Cube which is then presented to the user. The user also has several directional pointers to instruct what buttons to click to move a certain part, that can be seen in the picture below.

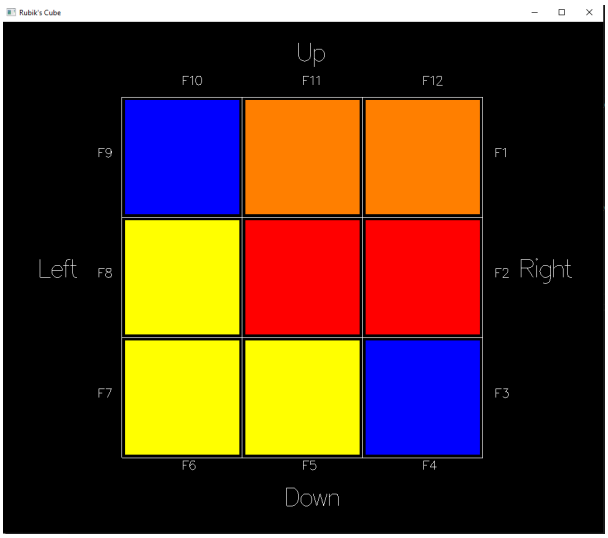

Figure 1: A starting gamestate of the Rubik's Cube

The user then clicks the arrows on the keyboard to change the perspective on the cube or F1-F12 to rotate a given segment of the cube. With those core moves the user solves the cube in a fluent way until it's complete and a victory screen is displayed. Inputs that are not valid are just blanks and

doesn't affect the program, except Esc which closes the window

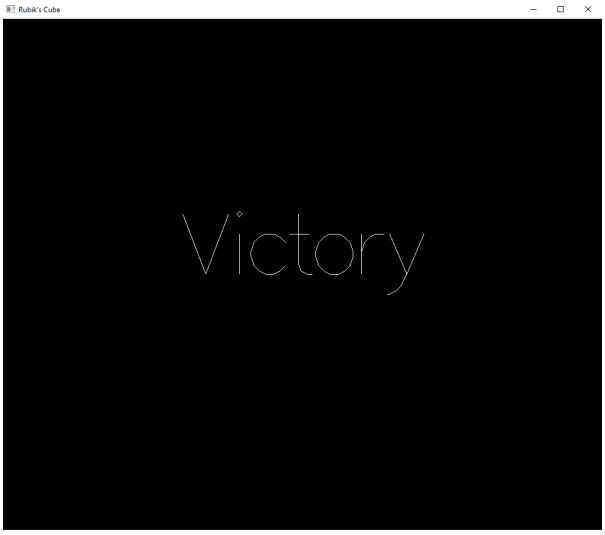

Figure 2: A screen when the Rubik's Cube is solved

3 Program Documentation

3.1 Data structures

We decided in order to most easily represent a cube we would first want a data type for each of the colors on a Rubik's cube. Then we put together nine colors for each cell to make a side. A side consists of lists of colors where there are three lists with three colors each, representing a three by three cube. A cube has six sides, so we made a type called cube which itself is a list consisting of six sides.

```
data Colour = Blue | Green | Red | Orange | White |
              Yellow
type Side = [[Colour]]
type Cube = [Side]
```

We also considered other variations of this way of what we wanted from our data types. For example we thought about not have sides be several lists, rather being just one list of nine colors. While doing things that way might be better in one way, it might make pattern matches for sides more readable, we therefore decided against it. Having side as three lists of three

colors made it easier for us to visualize the cube while writing the code, and will hopefully make it easier to visualize for anyone reading the code as well.

Another data structure we considered was to make each cell have two properties, one property for the color and one for the position of the cell in a side. This, however, was in the very early in our thinking about our data structures. When we came up with the idea that we later decided to run with, we put away the idea of cells with two properties in favour of being able to easily pattern match with lists where the position in the list corresponded to the position in the cube.

3.1.1 Eq Colour

In order to check if a cube was complete, we had to be able to check if a color was equal to another color, therefore we added an instance Eq Colour for each color as such:

```
instance Eq Colour where
    Red == Red = True
    Green == Green = True
    Yellow == Yellow = True
    White == White = True
    Blue == Blue = True
    Orange == Orange = True
    _ == _ = False
```

3.1.2 Cube visualization

One way to visualize a Rubik's cube is seeing it as a two-dimensional cross, instead of a three-dimensional cube. This was our mindset when designing our code. The side of the cube that we display in our program, what we call the front, is the in the middle of the cross. The right, left, up and down sides is to the right, left, up and down respectively on the cross, while the back side could be seen as occupying any of the ends of the cross, to the left of left side, over top side etc.

```
Each side is indexed as: Front = 0  Right = 1  Left = 2
Up = 3  Down = 4  Back = 5
```

A cube, which again is a list of sides, is ordered so that the position of an element in the cube corresponds to the direction of the side on a three-dimensional cube. When we want to look at the front side of the cube, we simply tell the cube to give us its first element. In order to look at, for

example, the right side of the cube, we do not tell the cube to give us the second element in the cube. Instead we change the order of the elements.

This has one major benefit and one major flaw. The benefit is that when using the rotate function (described later), we only ever have to think about how to rotate a row from the front side of the cube. The code for the rotate function is quite long and complex for just the front side, so being able to not worry about rotating from the perspective of other sides saves a great deal of pain.

The flaw is that it is not as easy as just changing to order of the elements in the cube. For example, if we want to turn the cube upwards so that the downwards facing side comes up to the front, we will also have to do something with the left and right sides. When doing this operation on a three-dimensional cube, if you observe the right side of the cube, you can see that each row is turned 90 degrees in a clockwise fashion. So, in addition to changing the position of sides in a cube we also had to perform a spin operation on sides that were not directly affected.

Another way in which this flaw manifests is when you take the back side of the cube and put it on the top or bottom sides. On a three-dimensional cube, back side will be flipped if you turn it up or down twice so that the previous back comes to the front, compared to if you take the back to the front by turning the cube sideways. On a two-dimensional cube however the side is the same no matter which direction it was turned from. In order to remedy this we had to perform the spin operation twice on the back side, as well as changing its position, whenever we turned the cube up or down.

Therefore a further improvement of the data structure would be shaping it in such a way that the spinning operations would not be necessary.

3.2 Files

Here are the functions which builds up the Rubik's Cube program, divided into their respective files. See figure 3 in appendix for the flow of the program through the functions.

3.2.1 Rubiks

Originally we just started off by creating a Rubik's cube without the gloss library. While doing so, we only had the Rubiks file. We therefore had a more basic main function which contained a display function and a askPromt function inside Rubiks. Which has now been moved to the "Main" file. The remaining functions are:

```
changePerspective :: Cube -> String -> Cube
```

As our code only can show one side at a time, we need to be able to rotate the entire cube to see the other sides. changePerspective does just that. If you want to rotate the cube, say up, you give the function your cube and your desired direction as arguments. It then pattern matches to see which direction to rotate in, and then does the corresponding rotation as seen in the following four functions:

```
turnUp :: Cube -> Cube
turnDown :: Cube -> Cube
turnLeft :: Cube -> Cube
turnRight :: Cube -> Cube
```

```
spinSideCW :: Side -> Side
spinSideCCW :: Side -> Side
```

spinSideCW and spinSideCCW (ClockWise, CounterClockWise) handles the problem that, when changing your perspecitve to either left or right, the top and bottom sides have to rotate 90 degrees to CounterClockWise respectively ClockWise.

```
isComplete :: Eq a => [[[a]]] -> Bool
sideComplete :: Eq a => [[a]] -> a -> Bool
```

isComplete checks if the entire cube is complete or not. It does this by recursively checking the sides of the cube via the "sideComplete" helpfunction, which checks if the entire side is of the same colour. As we started with a solved cube we can be sure that two different sides won't have the same colour, hence we don't need to remember what colours have been checked. Just that all the sides are of a singular colour.

```
rotate :: Int -> Cube -> Cube
```

The rotate function rotates a column or row (the different possibilities represented by an int $1 \leq x \leq 12$). We first solved how to rotate the first 3 rows CounterClockWise by patten-matching, then in order to rotate it ClockWise we just did the respective CounterClockWise rotation 3 times. Then in order to rotate something vertically, we just rotate the cube 90 degrees to it's side, do the horizontal rotation and the flip it 90 degrees back again. The 90 degree tilt is done by calling the spinSide functions on all of the sides of the cube.

```
scramble :: Cube -> Int -> IO Cube
```

The scramble function takes a solved cube and rotates a randomly selected row or column by 90 degrees, this process is repeated "n" times. We have arbitrarily selected n = 50 for our starting scramble. Having a too low number would make the solution trivial, but a too high number would be unnecessary. This number can be used to change the difficulty as well.

3.2.2 Changer

The Changer file handles the users input with the changeCube function.

```
changeCube :: Event -> Rubiks.Cube -> Rubiks.Cube
```

The changeCube function pattern matches the valid inputs with their corresponding change on the cube. The empty pattern match helps with invalid key presses as they don't change the cube at all.

3.2.3 Displays

The Displays file handles the visual representation of the cube and displays everything in the window.

```
cubeAsPicture :: Cube -> Picture
textPics :: Picture
cubeGrid :: Picture
colorSquares :: Cube -> Picture
placeSquares :: [(Float,Float)] -> [Picture] -> [Picture]
makeSquare :: Colour -> Picture
```

The main function is the cubeAsPicture function, it either displays a victory screen or combines the static visuals with the cube on the screen. The static visuals in the program are the textPics which are the directional instructions to the user and cubeGrid that distinctly divides the nine cells of the cubes face. The makeSquare function is a pure pattern match that constructs a rectangle in a given color, placeSquares takes a list of positional coordinates and places the list of pictures on those coordinates. By combining these two functions the colorSquares function concatenates the front side and maps the makeSquare function on the list of colours, this list is then given to the placeSquares function along with a list of predefined coordinates.

3.2.4 Main

```
screenWidth :: Int
screenHeight :: Int
window :: Display
```

The Main file is where the program loop has begun and certain parameters for the application are inputed. The screenWidth/Height where first planned for the window as the cube would fully fill the window, that was later changed as we needed a margin around the cube that made room for the directional instructions. The main function first scrambles a solvedCube then passes it to the play function along with the window settings, a backgroundcolour, a fps, the functions for changing the cube and displaying the cube. The amount of scrambles is set to 50 rotations.

3.3 External Libraries

These are the two libraries that the program needs to run that we did not write ourselves.

3.3.1 System.Random

From System.Random we only import randomRIO, a function to randomly get a number somewhere in an interval. The function is only used in one place, which is the scramble function. While we could have used something like getStdRandom, we wanted the scramble function to perform uniquely each time the program ran, without having to store a random generator to make it so.

3.3.2 Graphics.Gloss

This library is what we used to make the step from a text-based program to something which lets the user see and interact with the cube in a two-dimensional setting. Everything seen on screen when the program is running has been made through Gloss. Gloss is handled in the Main, Changer and Displays functions, while most of the logic that makes the cube work is in the Rubiks file.

3.4 Testing with external libraries

One difficulty we encountered with gloss and randomRIO was when we tried to write test cases for the functions. We therefore decided to test the func-

tions using HUnit in order to make sure that the function returned the expected result. Since randomRIO returns an IO output, we could not make HUnit tests for functions using randomRIO, so instead we had to manually test it to make sure it seemed correct. Functions using gloss were tested in a similar way, since their returned values were to us unpredictable, we could not estimate what the functions would return. We therefore decided to, instead of writing testcases, manually test if the functions worked. Since using gloss brings up a graphics window, we could simply look at it and compare to a real life cube when testing to see if it was working.

4 Shortcomings

The biggest shortcoming is the confusion and difficulty that comes with just being able to see one side of the cube at once. But after a while you get the hang of the controls and it becomes very intuitive. But a 3d representation of the cube instead of a 2d would decrease confusion and help navigation of the cube.

Currently, the user has no way of indicating the difficulty of the scramble. It can easily be modified in the code, but the user has currently no way to modify it themselves. You can't re-scramble or restart the scramble, other then by restarting the program. There is also no way to save your progress with your current cube if you wish to exit the program. This we don't feel is a major shortcoming as in most cases the user will solve it in one sitting.

5 Summary

This project focused on coding a Rubik's cube game with a visual display window and fast and responsive inputs. The main parts in the program are the coded Rubik's cube logic and the application of the Gloss library. The foundation of a good data structure was a major factor in the success of the project.

The possible changes of the program are to change the cube from 2d to 3d, but that would complicate the code, and a way to save the progress of the cube.

Appendices

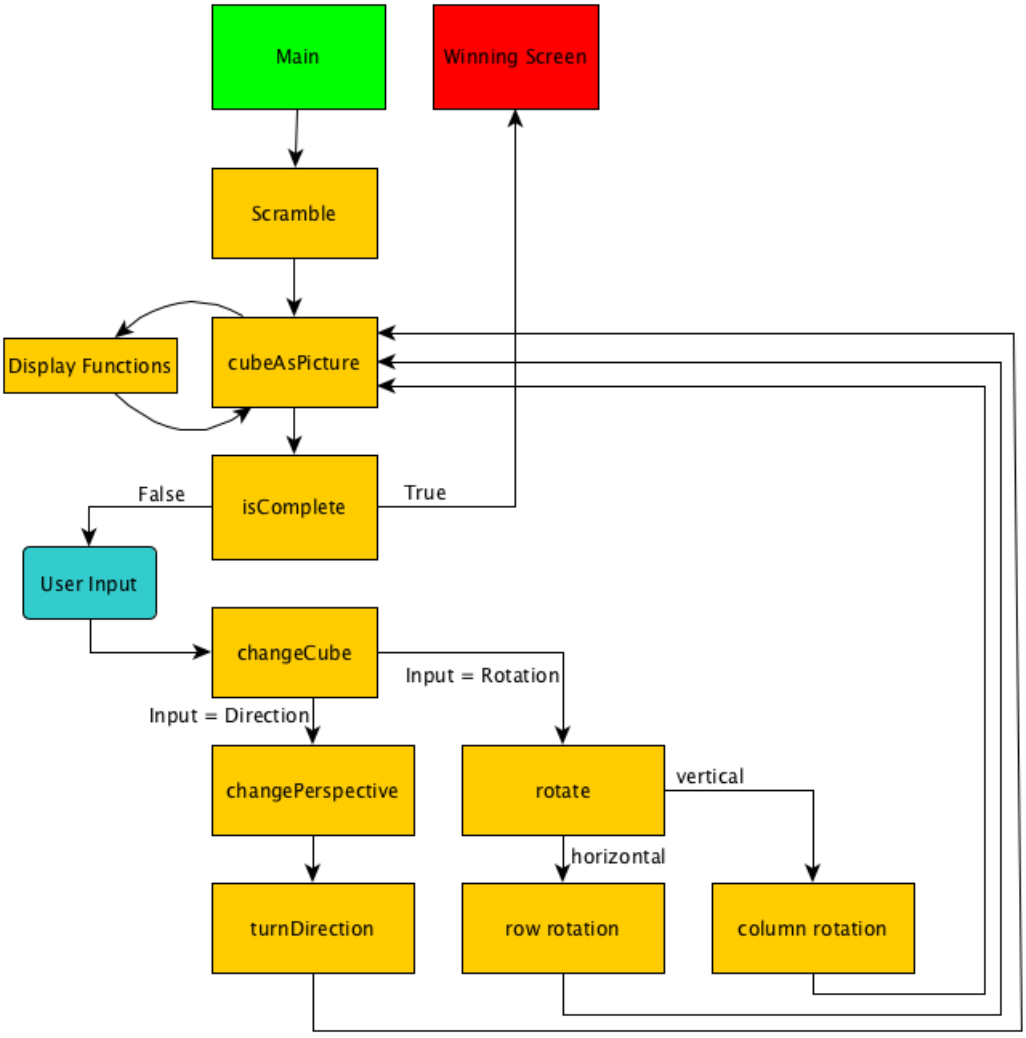

Figure 3: A Flowchart of how the program functions.