School of Computer Science, University of Nottingham Ningbo China

AE1PGP Programming Paradigms, Spring 2019

Jonathan Thaler

Haskell Coursework – Schelling Segregation

Deadline: 12th April 2019

This coursework covers the material from Lectures 1–8, and is **worth 15% of the module mark**. You should aim to make your definitions as simple, clear and elegant as possible. If you have any questions about this exercise, please post to Moodle, ask me after a lecture, in a lab, or by email to jonathan.thaler@nottingham.edu.cn. If any questions require this exercise to be clarified then this document will be updated and everyone will be notified via Moodle.

Introduction

The *Schelling Segregation* model is a very simple model to demonstrate the effect of preferences on segregation. It was conceived by the economist Thomas C. Schelling in 1969, who went on to receive the Nobel Price in economics for his work in 2005.

In this model we assume a discrete 2-dimensional world on a regular NxN cell grid. In this world a given number of agents are randomly placed on cells. Each agent has a color of Red, Green or Blue. It is assigned randomly at the start of the model and will stay the same throughout the whole simulation. Each cell can only hold a single agent and is thus either occupied by an agent or empty. For example if the worlds dimensions are 20x20 then we might place 250 random agents on the 400 available cells, 150 will be left empty.

Agents have a happiness measure, which is defined as the ratio of the number of neighbours with same color to the total number of neighbours. As neighbourhood we assume the Moore Neighbourhood (see Figure 1), which are the 8 surrounding cells of an agent. Note that in case of the border of the world, there might be less than 8 neighbours, but we will still use the same measurement of happiness.

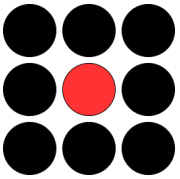
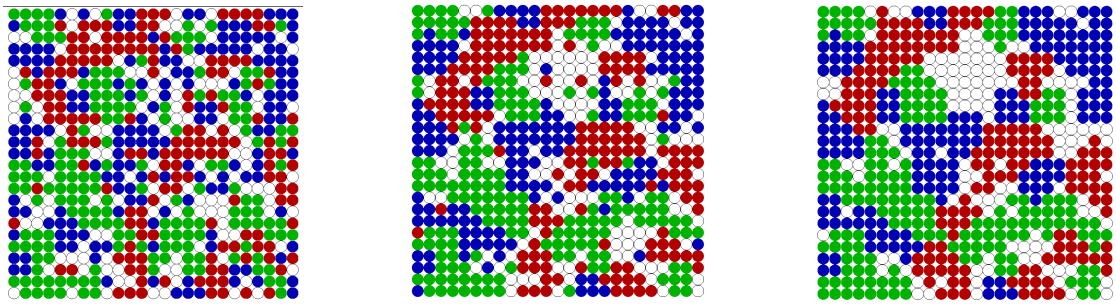


Figure 1: The Moore neighbourhood with the red cell being the point of reference and the black cells the 8 surrounding neighbours.

Agents which are unhappy will move to empty cells in the world until they found a place they are happy with. This works as follows: simulation time advances in discrete steps t=1,2,3,... In each time-step all unhappy agents move to a randomly chosen empty cell and all happy agents stay where they are. Then the next time-step happens until all agents are happy and no more movement is happening. See Figure 2 for a visualisation over time.



(a) t = 0 (b) t = 5 (c) Stable at t = 40

Figure 2: Visualisation of Schelling Segregation. After 40 steps the simulation is stable: all agents are happy and no more movement is happening.

You can think of this abstract description as people in a city which have preferences about their neighbourhood: they are happy if a given fraction of their neighbours are of their color / wealth / ... The striking result of this model is that the world will become segregated even if the agents seem to be quite tolerant e.g. having a preference of only a 40% similar neighbourhood. So we see that it is possible that out of simple and seemingly relaxed *local* rules, a surprisingly different *global* system behaviour can emerge.

Task

The task you have to do in the coursework is to implement the core algorithm of the Schelling Segregation as described above. To successfully finish this task you need to combine all the things you learned in lectures 1-8 and do a little research on your own (see below). The amount of code you have to write should be average - my working solution has about 80 lines of code including a lot of break-lines. There is already code provided for you, which performs the rendering and creating of the initial world, which is all implemented in 'src/Main.hs'. You need to implement the missing functions in 'src/Schelling.hs' for a working solution - this is also the only file you will submit.

Core Algorithm

Open 'src/Schelling.hs' and search for the 'step' function (line 30). This function will be called for you from the Main module to compute the next step with all arguments provided for you. Read the comments, which describe the function and its arguments, and implement it. Follow these steps as a guide:

1. Filter all agents which are unhappy. Write a function, which decides if an agent is happy. For this you need to define the neighbourhood and filter out the neighbouring cells.

2. Move each unhappy agent to a new empty cell. Be careful: only one agent can occupy a cell. Therefore when you move an agent to an empt cell, no other agent can be placed on this cell because it is occupied now. Think carefully about which higher-order function from the Standard Prelude you want to use for this. Also you will need the following functions:

   (a) Write a function which finds a random empty cell.

   (b) Write a function which takes a random element from a polymorphic list.

   (c) Write a function which swaps the contents of two cells.

   This should give you enough guidance to solve the task.

The Main module has error-checking implemented, which checks whether the number of cells and agents stays constant or not. If your program exists with an error, read the message carefully and figure out what the problem is - you might add / remove cells or agents during this step which is not allowed.

If you want to see how a different happiness factor influences the dynamics, feel free to change it: open 'src/Main.hs' and look for the declaration of *satisfactionRatio :: Double* (line 29-30) and change it to another value between 0.0 and 1.0. Also you can vary the number of agents by changing *agentCount :: Int* (line 38-39).

Randomness

In this coursework you will need to make use of a random-number generator, because the movement of the agents to a new cell is random. We haven't covered that topic in the lectures but it is not very difficult to learn when you have understood the concepts presented in the lectures. Consult the following page for an overview and good explanation how random-numbers work in Haskell: `https://en.wikibooks.org/wiki/Haskell/Libraries/Random#The_Standard_Random_Number_Generator` (ignore the other sections ...).

Note that the initial world is randomly generated but will always be the same random world, because the initial random-number generator seed is fixed. If you want to change it you can do so by opening 'src/Main.hs' and looking for the declaration of *rngSeed :: Int*

(line 43-44) and setting it to a different value. The seed is fixed to ensure reproducibility: repated runs will always have the same dynamics, even though they are initially random.

Converting numbers

You might need to convert between different numeric types (e.g. convert an Int to a Double or vice versa). Consult the excellent reference on converting numbers from: `https://wiki.haskell.org/Converting_numbers`.

Running the code

The provided skeleton code is actually a full Haskell project, which consists of different modules (Main and Schelling) and has dependencies to other libraries (containers, gloss and random), not included in the Standard Prelude. The Haskell Platform, which you should have installed on your machine, comes with the *cabal* tool to describe larger Haskell projects with multiple modules and their dependencies. This tool makes it easy to compile such a project because it automatically downloads the dependencies and builds the whole project into an executable. For this coursework the full project structure and cabal-file has been provided for you already and you don't need to worry about it. Still if you are interested in having a look at it and inspect its contents, open the file 'schelling.cabal'. To build and run the project, follow the steps described below.

Before the first usage of cabal you have to fetch / update the dependency informations for cabal. It will connect to the library repository to provide the local cabal installation with information about the versions of libraries available. You do this by invoking the following command on the command line (this could take a while):

```
cabal update
```

Next you should create a sandbox in which you build your project. By using a sandbox, the dependencies are installed in your local project folder instead of into the global cabal folder. This ensures that the dependencies which are downloaded, won't interfere with other Haskell projects. You create a new sandbox by navigating into the coursework project folder and invoke the following command on the command line:

```
cabal sandbox init
```

You are now ready to build the project! Simply invoke the following command on the command line. Note that this might take a while the first time because cabal will download and install the dependency libraries into your local sandbox:

```
cabal install
```

After having successfully built the project you then simply run it by invoking the following command on the command line:

```
cabal run
```

Note that the code provided for you works already and generates visual output but the agents don't move.

Note that you have to issue *cabal update* and *cabal sandbox init* only once for setting up the project. After having set it up successfully you only need to use *cabal install* followed by a *cabal run* to build and run your project after you have made changes.

Note that for this coursework I configured the build options of the compiler to be very restrictive: every warning is an error and will result in your project NOT being built

until you have removed all warnings. This is generally considered good style in software engineering and ensures better quality of code and avoids certain bugs. If you come across a warning then try to figure out what the problem is and fix it - if you struggle to do so, please post on Moodle or write an email.

Performance

This coursework is not about performance! Note that the simulation tries to calculate 3 steps per second but that due to a slower computer or inefficient implementation you might experience performance problems. For better performance, we would need to use a different data-structure instead of a list of cells and techniques which weren't covered in the lecture. If the simulation runs too slow on your machine, consider reducing the size of the world and number of agents. To do this, open 'src/Main.hs' and look for the declarations of *worldSize :: (Int, Int)* (line 33-34) and *agentCount :: Int* (line 38-39). Simply set them to a size and value which fits your implementation and computer specifications so it runs smooth. You also might consider lowering the number of steps computed per second: change *stepsPerSec :: Int* (line 47-48) to 1 or 2.

Resolution

The window created for generating the visual output is set to be 500x500 pixels. If you want to adjust this because you want a larger or smaller window you can do so by opening 'src/Main.hs'. Look for the *winSize :: (Int, Int)* declaration (line 25-26) and simply change the size.

Submission

You must submit a single zip file, which contains **only** the source code file **'Schelling.hs'** containing all your code for this exercise. The zip file must be named after your student ID number (ie, 6512345.zip) and must not require any other files.. The first line of the 'Schelling.hs' file should be a comment which contains your student ID number, username, and full name, of the form: – 6512345 zy12345 Joe Blogs

Unfortunately the Haskell Platform on the Linux servers is too old for this exercise, so I will use my own machine to run your program. I am using the Haskell Platform with GHC/GHCi version 8.2.2 and cabal version of 2.0. Make sure you have at least that version. If you installed the Haskell Platform at the beginning of this module, then you should be fine. To really make sure, check the version of your GHC/GHCi/cabal type 'ghc --version' / 'ghci --version' / 'cabal --version' on your console. If you have an outdated version, make sure you install an up-to-date version of the Haskell Platform first.

I will unzip your zip file into my local Schelling Segregation project directory. Your file must then compile without warnings or errors when I use the command 'cabal install'. If it does not compile, for any reason, then you will lose all the marks. If the file compiles but has warnings then you will lose some marks for not correcting the warnings.

I will then run the command 'cabal run' to run the complete project with your implementation. The simulation should reach a stable state after a maximum of 100 steps. Note that the speed by which the simulation converges to a final state depends on the size of the world, the number of agents, the satisfaction ratio and the initial random-number generator seed. These parameters are fixed in 'src/Main.hs' and cannot be changed by you in your submission because you will only submit 'Schelling.hs'. I will use the same values as in the code you receive. The steps are printed to the console, so you can test if your implementation converges within the required steps. Note that my implementation converges within about 40 steps, so 100 steps should be more than enough.

A few points (5) are awarded for style. Please don't focus on style right away: first find a solution for the problem, then try to improve the style of the program without breaking it - we call this process *refactoring*. When you improve the style of the program aim for the following: use of standard library functions (filter, map, foldr, length, not, zip, drop, take,...) instead of explicit recursion, use of list comprehensions, use of pattern matching, splitting up the whole problem into smaller problems and writing functions for the smaller problems. Obviously style is hard to measure and I wont be very strict on it but you should aim for a good style and show that you have at least tried.

A few points (5) are awarded for good comments on your code. You should write comments for code you write, describing briefly what the functions you write do, what the input arguments are and what the result is. I have written extensive comments for the functions in 'Main.hs' and 'Schelling.hs'. Look at how it is done in those files, I expect you to follow the same style of commenting.

The completed source code file should be uploaded to the Assessed Exercise 1 Submission link on the AE1PGP Moodle page. You may submit as many times as you wish before the deadline (the last submission before the deadline will be used). After the deadline has passed, if you have already submitted your exercise then you will not be able to submit again. If you have not already submitted then you will be allowed to submit once.

Plagiarism

You should complete this coursework on your own. Anyone suspected of plagiarism will be investigated and punished in accordance with the university policy on plagiarism (see

your student handbook and the University Quality Manual). This may include a mark of zero for this coursework.

You should write the source code required for this assignment yourself. If you use code from other sources (books, web pages, etc), you should use comments to acknowledge this (and marks will be heavily adjusted down accordingly).

You must not copy or share source code with other students. You must not work together on your solution. You can informally talk about higher-level ideas but not to a level of detail that would allow you all to create the same source code.

Remember, it is quite easy for experienced lecturers to spot plagiarism in source code. If you are having problems you should ask questions rather then plagiarize. If you are not able to complete the exercise then you should still submit your incomplete program as that will still get you some of the marks for the parts you have done (but make sure your incomplete solution compiles and partially runs!).

Marking

I will award points for the following:

- 15 marks: simulation reaches steady state before 100 steps.
- 5 marks: function which moves unhappy agents to random empty cells.
- 3 marks: function which determines if an agent is happy or not.
- 1 mark: function which finds a random empty cell.
- 1 mark: function which selects a random element from a polymorphic list.
- 5 marks: use of elegant functional style.
- 5 marks: detailed commentary on your code.

Note that when you follow the instructions as above you should reach the maximum of 35 marks.