Autómatas celulares

1- Instalacion del proyecto

El código fue testeado en una computadora con windows 10 y otra con Ubuntu 24.0.1.

Para ambos casos es necesario tener instalados ghc y cabal.

En la versión de windows se tenía las siguientes versiones:

- GHC: 9.6.5
- Cabal: 3.12.1.0

En la versión de Ubuntu se tenía las siguientes versiones:

- GHC: 9.6.6
- Cabal: 3.14.1.1

Adicionalemente es necesario, en el caso de linux, instalar los paquetes “freeglut3-dev” y ” libglfw3”.

Una vez se tiene lo necesario tan solo se debe correr, desde la carpeta del proyecto, el comando:

```
cabal build
```

2- Como correr un Autómata

Una vez construido el proyecto se puede correr el programa con el comando:

```
cabal run cellularAutomata -- [[Path (relativo/absoluto)]]
```

3-Estructura del código

La estructura del proyecto es la siguiente:

```
.
|-- app
|    |-- Main.hs
|-- examples
|     |-- game_of_life.ca
|     |-- rule90.ca
|-- modules
|   |-- AST.hs
|   |-- CAGraphics.hs
|   |-- CelAutoGen.hs
|   |-- Gen.hs
|   |-- Parse.hs
|-- AutomataCel.pdf
|-- celauto.cabal
```

```
|-- freeglut.dll -- necesario para que corra en windows
|-- parse.y
|-- README.pdf
```

4-Decisiones tomadas

La estructura de la grilla Debido a que la función de transición se debe aplicar a cada una de las celdas y además esta función hace referencias a los vecinos de la celda, era necesario que la estructura utilizada para representar la grilla de celdas tuviese accesos rápidos a cualquier celda. Originalmente utlicé un hash map para representar la grilla pero resultó ser insuficiente (O(log(n))) para grillas de tamaño medio, 100x100 por ejemplo. Por lo tanto la grilla se representa con un Vector (Vector ((Int,Int), String)) (O(1)).

Los gráficos Una vez realizado todo lo relacionado con el parseo y construccion del autómata, el final del proyecto consistió en encontrar una forma de que dada la estructúra que conforma al autómata celular obtener una imágen de dicha representación. No solamente eso, si no que también estose fuese repitiendo indefinidamente mientras a su vez se aplicaba la función de transición. Como el paquete "gloss" brinda las herramientas exactas para poder realizar esto (simulaciones) terminó siendo el paquete gráfico utilizado. Es necesario mencionar que buena parte del código gráfico (la función que dibuja la celdas y la grilla) es tomado (copy/paste) y modificado de un repositorio de un proyecto para hacer el juego de la vida de Conway.

Como definir un autómata celular

El DSL tiene 5 comandos para poder definir el autómata celular. Estos comandos son:

- GRID
- STATE
- NEIGH
- STEP
- START – No obligatorio

El orden de aparición en el archivo de definición no es importante (excepto para el comando STEP).

Todos los comandos deben finalizar con un ";" excepto el último el cual no debe terminar con nada.

GRID

Este comando permite definir las dimensiones de la grilla y como se comporta esta en los bordes.

Es importante aclarar que la coordenada origen (0,0) de la grilla es la superior izquierda, las coordenadas en X crecen a la derecha y las coordenadas en Y crecen hacia abajo.

La sintaxis del comando es:

```
GRID x y [TOROIDAL]
```

Ejemplos de comando GRID:

```
GRID 100 100
GRID 23 189 TOROIDAL
```

Este comando solo puede ser invocado una única vez en todo el archivo de definición.

De los tipos de comportamientos está el finito y el toroidal:

FINITE Si una celda se quiere referir a un vecino que no existe en la grilla, entonces se asume que ese vecino está en el estado por defecto. Un ejemplo sería el de una celda que está en el borde derecho de la grilla y quiere preguntar por el vecino inmediatamente a la derecha, y, como este no existe, cuando se pregunta por este se devuelve el estado por defecto.

Se asume por defecto este comportamiento y por lo tanto no es necesario aclararlo cuando se define la grilla.

TOROIDAL Este tipo de grilla se comporta como si fuera la superficie de un toroide. Si en una grilla toroidal de 100x100 la celda (99,99) se quiere referir a la celda (100,100) entonces es lo mismo que si se quisiera referir a la celda (0,0).

Una forma informal sería decir que la grilla tiene "efecto pac-man" en donde el personaje podía salir por una de las paredes del nivel y aparecía en la pared opuesta (Por favor ignorar todos lo relacionado al terraplanismo en donde lo utilizan, la acuñación del termino es previa).

Si se quiere definir este tipo de grilla entonces se debe aclarar "TOROIDAL" al final de la declaración de las dimensiones.

STATE

Este comando permite definir un estado, su color y si este es el estado por defecto.

La sintaxis del comando es:

```
STATE [Nombre] [#rgb] [DEFAULT]
```

Ejemplos de comando STATE:

```
STATE Gato #ffffff DEFAULT
STATE Perro #123456
```

```
STATE Pantera #000000
```

La sintaxis rgb debe ser completa. No es válido rgb como "#0" o "#123"

Solo se puede definir un único estado por defecto.

Se debe definir al menos un estado en el archivo de definición.

Se debe definir al menos un por defecto en el archivo de definición.

El estado por defecto puede ser el único estado definido.

Dos estados distintos pueden tener el mismo color.

NEIGH

Este comando permite definir la vecindad de las celdas.

La sintaxis del comando es:

```
NEIGH [MOORE/NEUMANN/(x,y)]
```

Ejemplos de comando NEIGH:

```
NEIGH MOORE
NEIGH NEUMANN
NEIGH (0,1)
```

NEUMANN La vecindad de Von Neumann refiere a todas las celdas que se encuentran a distancia 1 de la celda.

MOORE Parecida a la vecindad de Von Neumann, la vecidad de Moore incluye todas las celdas a distancia 1 más las esquinas de la celda.

Referencia (x,y) Como la vecindad de Moore y Von Neumann no son suficientes para poder definir todos los tipos de vecindad. Queda la opción de definir a partir de coordenadas relativas a la celda. Supongamos que nuestra vecindad define que la celda (10,10) es vecina de la celda (2,1) entonces para poder expresar esto en el archivo habría que escribir:

```
NEIGH (8,9)
```

ya que respecto a la celda (2,1) la celda (10,10) está a 8 unidades en X y 9 en Y de distancia.

Si se define una vecindad que se escapa de las dimensiones del tablero entonces se hace modulo sobre dicha vecindad.

Estas dos decalraciones son equivalentes:

```
GRID 100 100;
NEIGH (101,101)
```

```
GRID 100 100;
NEIGH (1,1)
```

STEP

Este comando permite definir la funcion de transición.

La sintaxis del comando es:

```
STEP [Estado/{Estado1,Estado2,...}] {predicadoX  THEN EstadoX, predicadoY THEN EstadoY,...}
```

Ejemplos del comando STEP:

```
STEP Alive {COUNT Alive = 2 THEN Alive, COUNT Alive = 3 THEN Alive, COUNT Alive < 2 THEN Dea
STEP {On, Off} {COUNT On = 1 THEN On}
```

El primer argumento del comando consiste en el conjunto de estados a los que aplican los predicados. El segundo es el conjunto de predicados que debe cumplir la celda que se está evaluando y si este predicado es verdadero entonces cambia al estado explicitado luego de la plabra THEN.

Si existen dos predicados distintos que evaluan a verdadero para una misma celda en un mismo paso, entonces se elige el que primero aparezca definido en el archivo.

La sintaxis de los predicados es:

```
Condición1 [AND/OR/XOR] Condición2 [AND/OR/XOR] ... THEN Estado
```

Una condición sola también es un predicado.

Está permitido el usar parentesis entre condiciones.

Por ejemplo:

```
(COUNT Alive = 1 AND (1,1) = Dead) OR (COUNT Dead = 3 XOR (2,5) = Alive) THEN Alive
```

La sintaxis de una condición es:

```
[COUNT Estado1 / x] [>/</=/<>] [COUNT Estado2 / y]
(x,y) [=/<>] Estado  --Siempre va primero la referencia
(x,y) [=/<>] SELF    --Siempre va primero la referencia
```

COUNT cuneta cuantos vecinos están en el estado dado.

SELF refiere al estado que tiene la celda que se está evaluando.

(x,y) es una referencia que toma a la celda evaluada como origen (0,0).

START

Este comando permite definir el estado inical del autómata.

La sintaxis del comando es la siguiente:

```
START Estado {(x1,y1), (x2,y2),...}
```

En este comando, las coordenadas provistas comenzaran en el estado que es pasado como argumento.

No se puede definir más de un estado inicial a una misma celda.

Si no se aclara en que estado comineza una celda, entonces esta comienza en el estado por defecto.

Comentarios finales

A la hora de intentar correr el código surgió un error en todas las PCs en las que se intentó correr originalmente el programa. El error era "unkown GLUT entry glutInit" y estaba presente tanto en windows como en linux. Después de realizar múltiples búsquedas e intentar varias soluciones propuestas en distintos foros, encontré la solución para ambos SOs.

En el caso de linux la solución fue instalar los paquetes mencionados anteriormente en la sección 1 de instalación del proyecto.

En el caso de windows, pude encontrar un foro que explicaba como solucionar el problema. Era necesario tener en una carpeta, que contenga el ejecutable, el archivo "freeglut.dll". Sin embargo yo no contaba con dicho archivo así que tuve que encontrar donde descargarlo.

Afortunadamente un estudiante que tuvo complicaciones similares respecto de utlizar FREEGLUT en windows tenía publicado dicho archivo para 32 y 64 bits.

Bibliografía

Paquetes utilizados: https://hackage.haskell.org/package/vector

https://hackage.haskell.org/package/unordered-containers-0.2.20/docs/Data-HashMap-Strict.html

https://hackage.haskell.org/package/gloss

Juego de la vida de Conway en Haskell (inspiración para la parte gráfica)

Solución a problema de gloss en Windows

FREEGLUT para Windows