Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Análisis de Lenguajes de Programación

R-322

Informe Trabajo Práctico Final

SCA: Lenguaje para la Definición de Automatas Celulares Estocásticos

Alumno

Rassi, Octavio R-4519/7

30 de Julio, 2025

1. Descripción del proyecto.

1.1. Introducción

1.1.1. Autómatas celulares

Un autómata celular [1] es un modelo computacional y matemático que describe la evolución de un sistema dinámico en pasos discretos. Este tipo de modelo resulta útil para representar fenómenos naturales en los que múltiples entidades simples interactúan entre sí de forma local.

Para definir formalmente un autómata celular, es necesario identificar sus componentes principales:

- **Espacio regular**: Generalmente se trata de una estructura geométrica como una línea, una grilla bidimensional o incluso un espacio de mayor dimensión. El espacio está dividido en unidades homogéneas llamadas células.
- **Conjunto de estados**: Es un conjunto finito de valores posibles que cada célula puede asumir. También se lo conoce como alfabeto, y los estados pueden representarse mediante colores, símbolos o números.
- **Configuración inicial**: Es la distribución de estados asignados a las células al comienzo de la simulación.
- **Vecindad**: Determina qué células se consideran adyacentes a una dada, y su posición relativa respecto a ella.

  Las dos vecindades más comunes son la de Von Neumann y la de Moore. La vecindad de Moore incluye las ocho células circundantes, mientras que la de Von Neumann considera solo las cuatro ortogonalmente adyacentes (arriba, abajo, izquierda y derecha).
- **Función de transición local**: Es la regla que rige la evolución del sistema. Esta función calcula el nuevo estado de cada célula en función de su estado actual y el de sus vecinos. Puede expresarse mediante fórmulas algebraicas, tablas o conjuntos de reglas.

1.1.2. Autómatas celulares estocásticos

Los autómatas celulares estocásticos (SCA, por sus siglas en inglés) [2] son una extensión de los autómatas celulares tradicionales, en los que se incorpora el azar como parte fundamental del modelo. Al igual que en los modelos clásicos, se trabaja sobre un espacio regular de celdas que evolucionan a lo largo del tiempo según reglas locales que toman en cuenta el estado de las células vecinas.

La diferencia clave con respecto a los autómatas deterministas radica en que las reglas de transición no son fijas, sino que incluyen componentes probabilísticos. Es decir, el nuevo estado de una célula no está completamente determinado, sino que depende de una distribución de probabilidad que puede estar influida por su entorno.

A pesar de que las reglas individuales pueden ser simples y estar basadas en decisiones locales aleatorias, estos sistemas logran generar comportamientos colectivos complejos. Fenómenos como la emergencia y la autoorganización aparecen de manera natural en este tipo de modelos, lo que los convierte en herramientas valiosas para simular procesos reales donde la incertidumbre o el ruido juegan un rol importante. Ejemplos comunes incluyen la propagación de incendios, la dinámica de enfermedades infecciosas o la evolución de ecosistemas.

1.2. Objetivo.

En este trabajo desarrollamos un lenguaje de dominio específico para definir y posteriormente simular tanto autómatas celulares estócasticos como clásicos. El lenguaje definido lleva el nombre de `SCA`, por las siglas de *Stochastic Celullar Automaton*, y nos permite definir en pocas sentencias las principales partes de un autómata celular para luego visualizar su evolución gráficamente.

Notemos que se trata de un primer acercamiento a la implementación de autómatas celulares, pues `SCA` no es lo suficientemente expresivo para poder definir la totalidad de los autómatas celulares (estocásticos o no) posibles. Hemos elegido representar autómatas celulares con ciertas características que simplifican su simulación y el desarrollo del lenguaje en general.

2. El lenguaje `SCA`.

Como mencionabamos previamente, `SCA` tiene ciertas limitaciones a la hora de representar autómatas celulares. En particular, nos hemos restringido a representar aquellos con las siguientes características:

- **Espacio regular bidimensional**. El espacio sobre el que definiremos los autómatas sera exclusivamente en dos dimensiones, permitiendo asi visualizar su evolución a través de una grilla.
- **Conjunto de estados predefinido**. El conjunto de estados a utilizar no podrá ser definido arbitrariamente, sino que se podrá utilizar únicamente dos estados: *Vivo* o *Muerto*.
- **Frontera periódica**. Una cualidad relevante a definir sobre los autómatas que no habiamos mencionado hasta ahora son las condiciones de frontera. Al querer representar a nuestros autómatas en máquinas lamentablemente finitas, es inevitable establecer fronteras de algún tipo a nuestro espacio regular. Incluso la idea de redimensionar dinámicamente el espacio se encontrará indefectiblemente con una frontera dada por la finitud de la memoria.

  Es por ello que en este trabajo elegimos implementar autómatas celulares con *frontera periódica*. Una forma de pensar a las fronteras periódicas es ver al espacio regular como si sus extremos se tocaran. En el caso particular del espacio bidimensional que utilizaremos, si nos alejamos una dimensión podríamos visualizar a nuestra grilla como un toroide.

Ahora, conociendo las limitaciones del lenguaje y su poder expresivo, presentaremos su sintaxis.

2.1. Sintaxis abstracta.

init ::= grid neigh layout seed step

grid ::= **defineGrid** nat nat | ϵ
neigh ::= **defineNeighborhood** neighborhoodType | ϵ
neighborhoodType ::= **moore** | **vonNeumann**
layout ::= **defineLayout** [pairList] | ϵ
pairList ::= pair | pair, pairList
pair ::= (nat, nat)
seed ::= **defineSeed** nat | ϵ
step ::= **defineStep** stateExp

stateExp ::= **alive** | **dead**
| **if** boolExp **then** stateExp **else** stateExp
| **not** stateExp
| **withProbablity** prob **become** stateExp **else become** stateExp

prob ::= **random** | floatExp
neighbor ::= **left** | **right** | **top** | **bottom**
| **topleft** | **topright** | **bottomleft** | **bottomright**
| **self**

floatExp ::= float
| $-_u$ floatExp
| floatExp + floatExp
| floatExp $-_b$ floatExp
| floatExp $\times$ floatExp
| floatExp $\div$ floatExp
| **count** stateExp

boolExp ::= **true** | **false**
| floatExp == floatExp
| floatExp $\neq$ floatExp
| floatExp $<$ floatExp
| floatExp $>$ floatExp
| boolExp $\wedge$ boolExp
| boolExp $\vee$ boolExp
| $\neg$ boolExp
| neighbor **is** stateExp

2.2. Sintaxis concreta.

digit ::= `'0'` | `'1'` | ⋯ | `'9'`
nat ::= digit | digit nat
float ::= nat `'.'` nat

init ::= grid neigh layout seed step
grid ::= `'defineGrid'` nat nat | ϵ

neigh ::= `'defineNeighborhood'` neighborhoodType | ϵ
neighborhoodType ::= `'moore'` | `'vonNeumann'`

layout ::= `'defineLayout'` `'['` pairList `']'` | ϵ
pairList ::= pair | pair `','` pairList
pair ::= `'('` nat `','` nat`')'`

seed ::= `'defineSeed'` nat | ϵ

step ::= `'defineStep'` stateExp

floatExp ::= nat
| `'count'` stateExp
| `'-'` floatExp
| floatExp `'+'` floatExp
| floatExp `'-'` floatExp
| floatExp `'*'` floatExp
| floatExp `'/'` floatExp
| `'('` floatExp `')'`

boolExp ::= `'true'` | `'false'`
| neighbor `'is'` stateExp
| floatExp `'=='` floatExp
| floatExp `'!='` floatExp
| floatExp `'<'` floatExp
| floatExp `'>'` floatExp
| boolExp `'&&'` boolExp
| boolExp `'||'` boolExp
| `'!'` boolExp
| `'('` boolExp `')'`

stateExp ::= ‘alive’ | ‘dead’
| ‘if’ boolExp ‘then’ stateExp ‘else’ stateExp
| ‘not’ stateExp
| ‘withProbablity’ prob ‘become’ stateExp ‘else become’ stateExp
| ‘(’ stateExp ‘)’

prob ::= ‘random’ | float | float ‘%’ | nat | nat ‘%’

neighbor ::= ‘left’ | ‘right’ | ‘top’ | ‘bottom’
| ‘topleft’ | ‘topright’
| ‘bottomleft’ | ‘bottomright’
| ‘self’

2.3. Ejemplos.

Para visualizar mejor la semántica del lenguaje lo mas simple es analizar un ejemplo. El autómata celular (no estocástico!) mas reconocido es el Juego de la Vida de Conway. Convenientemente para nuestro trabajo, es posible representarlo sobre un toroide sin mayores complicaciones en SCA.

Listing 1: El Juego de la Vida de Conway

```
defineGridSize 40 40

defineNeighborhood moore

defineLayout [(1,25),(2,23),(2,25),(3,13),(3,14),(3,21),(3,22),(3,35)
   ↪,(3,36),(4,12),(4,16),(4,21),(4,22),(4,35),(4,36),(5,1),(5,2),(5,11)
   ↪,(5,17),(5,21),(5,22),(6,1),(6,2),(6,11),(6,15),(6,17),(6,18),(6,23)
   ↪,(6,25),(7,11),(7,17),(7,25),(8,12),(8,16),(9,13),(9,14)]

defineStep if self is alive
           then (if (count alive < 2)
                 then dead
                 else (if (count alive > 3)
                       then dead
                       else alive))
           else (if (count alive == 3)
           then alive
           else dead)
```

Este programa SCA define una grilla de tamaño 40×40, con un tipo de vecindad Moore, la regla usual del Juego de la Vida, y una configuración inicial donde las únicas celulas que iniciarán con vida serán las situadas en las coordenadas listadas (su elección no es casual, esta disposición genera una colonia de celulas comunmente llamada *Glider Gun* [5] en el contexto del *Game of*

Life). Además, por omisión de la expresión `defineSeed`, la semilla de números aleatorios que se utiliza en la evaluación de expresiones probabilísticas será inicializada a 42.

Notemos que una regla en nuestro lenguaje no es mas que una *expresión de estado*. Se trata de una expresión que puede hacer uso de distintas expresiones lógicas y numéricas, y además utilizar información de la grilla a traves de los bloques `count` e `is`. De esta manera, en todo momento dada una grilla y una posición (en realidad, tambien es necesario saber el tipo de vecindad y que semilla de aleatoriedad utilizamos) será posible reducir esta expresión de estado a un valor (es decir, un estado `dead` o `alive`) que terminará siendo el estado de esa celda en la próxima generación.

En este pequeño ejemplo hemos definido un autómata celular clásico. Veamos un programa un poco mas extenso que utilice las expresiones `withProbability` que permiten introducir no determinismo a las ejecuciones.

Listing 2: Nuestro primer autómata celular estocástico.

```
defineGridSize 40 40
defineNeighborhood vonNeumann

// Patron inicial: Dos colonias que interact an
defineLayout [
    // Colonia 1 (forma estable)
    (10,10),(10,11),(10,12),
    (11,10),        (11,12),
    (12,10),(12,11),(12,12),

    // Colonia 2 (estructura en crecimiento)
    (20,20),(20,21),(20,22),(20,23),
    (21,20),                (21,23),
    (22,20),                (22,23),
    (23,20),(23,21),(23,22),(23,23),

    // Semillas aleatorias adicionales
    (5,30),(6,30),(7,30),
    (30,5),(30,6),(30,7),
    (35,35),(36,36)]

defineSeed 2025  // Semilla fija para reproducibilidad

defineStep if self is alive
           then (if (count alive > 4)
                 then withProbability 80% become dead else become alive
                 else (if (count alive < 2)
                       then dead
                       else alive))
           else (withProbability ((count alive) * 0.1)
                  become alive
                  else become dead)
```

En este programa vemos mas funcionalidades de `SCA`. Por un lado, notamos que es posible incluír comentarios que faciliten la lectura de los programas prefijandolos con `'//'`.

Además, la regla o *step* definido utiliza en dos ocasiones la expresión `withProbability`, en un caso con un valor fijo y en otro caso incrementando la probabilidad de que una celula pase del estado muerto al estado vivo en función de su cantidad de vecinos vivos. Y, por último, presentamos un *layout* un poco mas interesante que el definido en el ejemplo anterior.

Con estos ejemplos, y siguiendo la sintaxis concreta que vimos anteriormente, tenemos suficiente información para poder definir autómatas celulares estocásticos complejos sin mayores dificultades.

3. Manual de uso e instalación.

Ya vimos en la descripción del lenguaje como podemos escribir programas `SCA`. Veamos ahora como podemos efectivamente ejecutar estos programas.

Para comenzar, debemos ejecutar el comando

```
stack build
```

desde el directorio raiz. Esto descargará las dependencias necesarias para poder ejecutar programas `SCA`.

Una vez finalizada la descarga e instalación de librerías necesarias, podremos ejecutar programas con el comando

```
stack exec cellular-automaton-dsl-exe <nombre_del_programa>  [<generaciones>]
```

El archivo de texto que contenga al programa a ejecutar debe encontarse dentro del directorio `test`. El argumento `generaciones` es opcional, y permite definir la duración de la simulación a ejecutar en cantidad de generaciones. Por defecto, este valor es 100.

4. Organización de los archivos.

El trabajo se encuentra dividido en tres directorios principales:

- **src**. Contiene el código fuente principal del lenguaje. En particular, podemos encontrar
  - `AST.hs`. Define los tipos utilizados a lo largo de todos los archivos.
  - `Parser.hs`. Define el parser del lenguaje que produce expresiones de tipo `Program` a partir del texto de un programa.
  - `Eval.hs`. Define al evaluador del lenguaje.
- **app**. Contiene un único archivo, `main.hs`, donde se encuentra el código que une las distintas partes de **src** para lograr ejecutar una simulación a partir de un programa. En este archivo se encuentran las funciones para representar gráficamente un `Grid` y el *loop* que simula las distintas generaciones del autómata celular.

- **test**. Es el directorio donde se almacenan los archivos a ejecutar. En el mismo se encuentran los ejemplos que definimos anteriormente, y es aquí donde deberían añadirse nuevos programas en caso de querer ejecutarlos.

Los demás archivos y directorios que pueden encontrarse en el directorio raíz son generados por la herramienta `Stack`, y pueden ser ignorados.

5. Decisiones de diseño y observaciones.

5.1. Expresiones.

La idea de `SCA` es poder escribir autómatas celulares estocásticos con una sintaxis amigable y fácil de aprender. Por ello, optamos por representar a las reglas como simples expresiones de estado, que tienen una estructura muy similar a un lenguaje de programación imperativo estándar.

De esta manera, una regla de evolución no es mas que una serie de expresiones de estado mas pequeñas, eventualmente conectadas con operadores `if then else` y su contraparte estocástica, `withProbability`.

Uno de los detalles que puede resultar llamativo en la definición de las expresiones del lenguaje en `AST.hs` es que no existen las expresiones enteras, sino que las expresiones numéricas son de tipo `Double`. Esta no fue la idea inicial, pero al querer representar probabilidades resultaba poco conveniente que las expresiones sean enteras, pues en ocasiones quisieramos permitir que las expresiones tomen valores decimales para operarlas como probabilidades.

Por ello, decidimos llevar todas las expresiones a tipo `Double`. Si bien puede resultar poco intuitivo que expresiones del lenguaje como `Count Alive` evaluén a valores en punto flotante (pues la cantidad de celulas adyacentes vivas es siempre un número natural), en la práctica es indistinto pues el posible error numérico no se presenta realmente.

5.2. Parsing.

Una de las primeras decisiones a tomar al implementar el lenguaje fue como realizaríamos el parseo de los programas. Optamos por utilizar la librería `Parsec` [3], que ya habíamos utilizado en otros trabajos de la materia, pues la sintaxis del lenguaje es relativamente simple y otras opciones como `Happy` resultaban innecesiaramente complejas para nuestro alcance.

5.3. El evaluador.

Lo mas relevante en cuanto a la evaluación recae en la mónada definida para ello, `EvalM`.

```
1  newtype EvalM a = EvalM
2   { runEval :: Grid -> Position ->
3                Neighborhood -> StdGen -> Maybe (a, StdGen) }
```

Aquí podemos ver que la evaluación de una expresión del lenguaje puede depender de 4 parámetros: el estado actual de la grilla, la posición de la grilla desde la cual se evalúa, el tipo de vecindad que se considera, y la semilla aleatoria utilizada. Claramente, la evaluación de una expresión numérica simple como `Const 42` no requiere de ninguno de estos parámetros, pero una regla de evolución compleja que utilice las capacidades estocásticas del lenguaje y las expresiones como `Count` requerirán de esta información para evaluarse.

Además, el tipo `Maybe (a, StdGen)` nos está indicando dos cosas: una evaluación puede fallar, y en caso de no fallar, el generador de números aleatorios es acarreado entre evaluaciones. Lo primero sigue de que el lenguaje incorpora la división, por lo que es posible que se produzca una división por cero y no se pueda evaluar la expresión, y lo segundo sigue de como se implementa el generador de números aleatorios en Haskell; de no acarrearse el generador y usarse siempre el mismo, cada llamado a `getRandom` evaluaría al mismo valor, pues para garantizarse la transparencia referencial es necesario producir un nuevo generador cada vez que se utiliza uno.

Veamos ahora como se expresa esto en la instancia de `Monad` para `EvalM`,

```
1 instance Monad EvalM where
2   return x = EvalM $ \_ _ _ gen -> Just (x, gen)
3   m >>= f = EvalM $ \grid pos neigh gen ->
4     case runEval m grid pos neigh gen of
5       Just (x, gen') -> runEval (f x) grid pos neigh gen'
6       Nothing -> Nothing
```

Podemos notar que en caso de producirse un error (es decir, el resultado de evaluar `runEval m grid pos neigh gen` es `Nothing`) este se propaga como podíamos esperar, y en el caso contrario se acarrea la (posiblemente) nueva instancia de `StdGen` para el próximo cómputo monádico.

Una observación pertinente es que en un principio la idea de utilizar la mónada `Maybe` también podía ser útil para modelar aquellos casos en que una expresión de probabilidad se escapara del rango $[0, 1]$. Sin embargo, mas adelante en la implementación del evaluador optamos por acotar los resultados de estas expresiones, esto es, si una expresión de probabilidad evaluara a -16 o 42, entonces en vez de producirse un error se comportaría como 0 o 1, respectivamente.

Es esta mónada definida junto con las funciones auxiliares como `getCurrentState`, `getRandom`, o `getNeighbors` lo que simplifica significativamente la escritura del evaluador y permite modificarlo fácilmente pues lo vuelve modular.

5.4. La representación gráfica.

En este apartado experimentamos con distintas opciones, como `Gloss` o `Reanimate`, pero considerando el objetivo del trabajo, escogimos la opción mas simple y minimalista pues el foco no está en la visualización en si sino que en la implementación del lenguaje.

Es por ello que la ejecución de la simulación se puede visualizar como un programa en terminal

en código `ASCII`, pues consideramos que para generar una grilla con dos posibles estados es suficiente con esta representación. Esto lo implementamos con la librería `System-Console-Ansi` [4], que brinda funciones útiles para manipular la terminal.

6. Bibliografía.

Referencias

[1] Wikipedia. *Autómatas celulares.*
https://es.wikipedia.org/wiki/Aut%C3%B3mata_celular

[2] Wikipedia. *Stochastic cellular automaton.*
https://en.m.wikipedia.org/wiki/Stochastic_cellular_automaton

[3] Hackage. *Documentación de Parsec.*
https://hackage.haskell.org/package/parsec

[4] Hackage. *Documentación de System Console ANSI.*
https://hackage-content.haskell.org/package/ansi-terminal-1.1.3/docs/System-Console-ANSI.html

[5] Wikipedia. *Glider, Conway's Game of Life.*
https://en.wikipedia.org/wiki/Glider_(Conway%27s_Game_of_Life)