PROYECTO DE PROGRAMACIÓN FUNCIONAL EN HASKELL

PROGRAMACIÓN DECLARATIVA

Tema: **Generador y solucionador de hidatos sudoku**

Autores:
Leandro Rodríguez Llosa
Andry Rosquet Rodríguez

Date: 12 de julio de 2022.

¿Qué es Haskell?

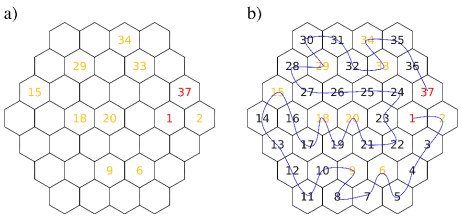
Haskell es un lenguaje declarativo, orientado a la programación funcional, los cuales se basan de manera general en el cálculo computacional de la función Lamda (λ). Haskell en particular resalta por ser un lenguaje funcionalmente puro, con un tipado estático fuerte y una poderosa inferencia de tipos. Entre alguna de las características más importantes a destacar están la transparencia referencial, la técnica de currying, el uso de funciones de orden superior, la correspondendia de patrones, la evaluación perezosa y las funciones y tipos polimórficos.

Hidato sudoku

Hidato sudoku, es un juego de lógica inventado por el matemático israelí Dr. Gyora M. Benedek. Un sudoku de este estilo puede adoptar cualquier forma, y está compuesto por un valor mínimo (generalmente 1), uno máximo (generalmente la cantidad de celdas), y algunos otros números dispersos en el puzzle. El objetivo del juego es llenar las cuadrículas vacías, con números consecutivos que se conectan horizontal, vertical o diagonalmente, y se sabe que existe un único camino que comienza en el valor mínimo dado, y termina en el máximo.

Mirándolo desde un enfoque un poco más computacional, se puede ver como un grafo no dirigido y conexo, con algunos nodos vacíos y otros con valores, y el problema consiste en encontrar un camino de Hamilton que comienza en el valor mínimo que se encuentra en el grafo y termina en el máximo.

Para entender mejor lo que se estaba explicando anteriormente se deja una imagen donde muestra un hidato no solucionado en el inciso *a)* y su solución en el *b)*.



Para modelar el problema, y representar cualquier figura que se desee, se tienen dos matrices, una con los números y otra de máscara marcando en *True* las casilla válidas del juego y en *False* las que no. Para ellos se creó un tipo *Hidato* que modela este concepto.

Hidato: este tipo se define para almacenar la información referente a un Hidato Sudoku. Entre sus propiedades se tiene una matriz de números que representa el tablero (*matrix :: [[Int]]*), una máscara booleana que marca en *False* las casillas que tienen obstáculos (*mask :: [[Bool]]*), un entero para tener la cantidad de casillas libres, y otros dos enteros que representan las dimensiones del tablero.

El tipo Hidato representa una instancia de la clase de tipos *Show*, y por tanto se redefine la función **show** con el objetivo de poder mostrar con facilidad un hidato.

Generador de hidatos

En esta sección abordaremos los aspetos más importantes para generar un hidato válido a partir de un *Template* dado.

Acercamiento general. Lo primero que se hace es cargar el *Template* y generar una posición de inicio del sudoku. Esta selección de una casilla inicial en el tablero se hace de manera aleatoria, haciendo uso del paquete *random* encontrado en módulo *System.Random*. Luego, se comienza en este punto la búsqueda de un camino de Hamilton en la matriz. Si la búsqueda no fue exitosa, se repite el mismo proceso.

Una vez se tenga dicho camino de Hamilton se procede a quitar casillas, analizando qué valores son obligatorios en la posición que se encuentran y los que son prescindibles. Al finalizar el proceso anterior tendríamos un tablero con solución única.

Métodos y estrategias. El método que busca el camino de Hamilton comenzando desde una posición aleatoria es *generateHidato*. Este genera una posición aleatoria con la función *getRandom*, luego intenta encontrar un camino de Hamilton cualquiera, y si lo halla, genera una solución única a partir de este, en caso contrario vuelve a intentar generar otro hidato (este proceso recursivo se hace hasta encontrar una solución válida).

Luego teniendo el tablero con una solución válida, se procede a utilizar la función *findUniqueSolution* la cual utiliza dicho sudoku ([[a]]) y su mask ([[Bool]]) para encontrar un nuevo tablero con espacios vacíos que tenga solución única([[a]]). Dicho método busca información necesaria y termina llamando a la función *iteration* la cual lógicamente itera del 2 al máximo posible del tablero, procediendo a realizarle un análisis a cada uno en relación con su ubicación en el sudoku resuelto.

¿En qué consiste este análisis?

Se procede a remover dicho número del tablero y se hace un llamado a la función *solveNumber* que recibe la matriz de números actualizada ([[a]]) y su mask ([[Bool]]), devolviendo el número de soluciones factibles de ese sudoku. Esto conllevaría a dos posibles resultados que son explicados a continuación y se analizan en la función *iteration*, la cual se describirá a fondo más adelante:

- Si el resultado es 1 se elimina dicho valor de manera definitiva porque significa que dicho dígito siempre, necesariamente, va a ser ubicado en esta posición.
- Si el resultado es mayor que 1 esto implica que dicho dígito debe estar ubicado obligatoriamente en el tablero resultante, de lo contrario si no estuviese fijado generaría múltiples posibles soluciones.

Cuando se realizó dicho análisis a cada valor, se obtendría el tablero con solución única donde estarán números fijados si son necesarios y espacios disponible que solo brindarían solución válida si se ubican los números requeridos.

Las funciones anteriores se auxilian de otros métodos para funcionar:

- *genRandom:* Esta función recibe dos valores enteros que corresponden con el límite inferior y superior del intervalo, y devuelve un valor aleatorio comprendido en este. Se utilizan las funciones *randomR* y *getStdRandom* del módulo **System.Random**, de la siguiente manera:

  ```
  getRandom x y = getStdRandom $ randomR (x,y)
  ```

  Cabe destacar que esta funcionalidad fue una de las más desafiantes en el proceso de desarrollo, porque la idea de lo que se quiere hacer rompe con el principio de transparencia referencial. Para lograr cumplir el objetivo planteado, se utilizan los tipos *IO Monad* que dan acceso a obtener una instancia del tipo *StdGen*, y como esta tiene una variable global que va cambiando su estado en cada llamado a dicha función, producto a su interacción con un medio externo el cual va variando constantemente, se puede lograr un comportamiento no determinista. Para más información sobre lo antes plateando, consulte la documentación oficial.
- *searchHamiltonianPath:* Esta función recibe un hidato, una posición en dicho hidato, una dirección y el tamaño del camino que se tiene hasta el momento. Esta función es un algoritmo de Backtrack clásico, que avanza mientras queden casillas libres en el hidato y la nueva posición a la que se mueve es válida. Las direcciones hacia donde se mueven se controlan mediante un tipo definido llamado *Direction*, y las funciones *directionToRow* y *directionToCol* que dada una dirección devuelven el sumando que permite trasladarse en esa dirección.

  Es importante señalar que este algoritmo es capaz de resolver un tablero de hidato, y para esto, se añade la comprobación de que antes de trasladarse si algún vecino de la casilla actual es la próxima casilla del camino que se está formando, entonces se traslada hacia esta posición y se continua el algoritmo. Pero para añadirle mayor robustez a la solución se decidió crear dos algoritmos distintos, uno para generar el hidato, y otro para solucionarlo.
- *Direction:* este es un tipo que se define con el objetivo de modelar un array de direcciones. Junto con ese tipo vienen implementados dos funciones *directionToRow* y *directionToCol*, las cuales retornan dada una dirección el sumando que se necesita sumar a la posición actual para ser desplazado en dicha dirección.

- *isValidPosition:* esta función define un criterio para decidir si una casilla del hidato es válida. Se dice que una casilla es válida si está comprendida en los límites de la matriz, no es un obstáculo y la posición es igual a 0 (o sea, no ha sido visitada), o es igual a el tamaño de paso menos uno, que significa que será la próxima casilla por visitar en el camino que se está construyendo.
- *replaceHidato:* esta función recibe un hidato, una posición y un valor, y devuelve otro hidato con este valor insertado en dicha posición. Para la construcción del hidato resultante se hace uso de la función auxiliar *replaceMatrix*
- *replaceMatrix:* esta función recibe una lista de listas genérica ([[a]]), una posición y un valor genérico también. Esta función hace uso de otra función auxiliar *replace* la cual reemplaza un valor en una lista ([a]), junto con la estructura **let - in** primero reemplazando el valor en la fila correspondiente, y luego haciendo uso de la misma función *replace* se reemplaza ahora la fila completa en la matriz.
- *iteration:* recibe un entero x, el máximo, la posición de x en el sudoku resuelto, el tablero y su *mask*. Se encarga de iterar de ese x hasta el máximo, analizando que ocurre si se elimina x del tablero para finalmente devolver el tablero con solución única.
- *solveNumber:* básicamente es el controlador que recibe el tablero, recopila información necesaria y luego ejecuta *giveNumber*.
- *changeState:* recibe un entero, posición de columna y de fila y el tablero actual y devuelve una nueva matriz ([[a]]) con el entero ubicado en la posición recibida.
- *correctlyPlaced:* recibiendo un valor x que ya se encuentra ubicado en el tablero y la posición del antecesor a este brindará *True* si están adyacentes y *False* de lo contrario.
- *giveNumber:* recibe un entero, el máximo del tablero, posición de la columna y fila del antecesor, una lista de números que ya se encuentran fijados, el tablero y su mask, devolviendo finalmente un entero que representará el número de soluciones que tiene el tablero actual. Itera por cada número, solo se detiene si llega al número máximo del tablero, de lo contrario llama a *validPlace* con cada una de las posibles ubicaciones adyacentes al antecesor. Funciona recursivamente sumando el número de soluciones factibles generadas al ubicar un número adyacente a su antecesor en las múltiples posibilidades.
- *validPlace:* recibirá un entero x, el máximo del tablero, la posición donde se desea ubicar a x, la lista de fijados, el tablero y su *mask*. Se encarga de comprobar si la posición dada es válida según la máscara por medio de *notValidMask* y si la posición no se encuentra ya ocupada en el tablero con *notValidHidato*. Si cumple todos los requisitos entonces se hace un llamado a *giveNumber* con el sucesor y el tablero actualizado con el número ya ubicado. Lo anterior asegura un ciclo continuo hasta que se haya recorrido cada número.
- *notValidMask* y *notValidHidato:* fueron descritas un poco anteriormente pero a mayor profundidad ambas reciben fila y columna, lo cual constituye la posición que se desea comprobar, además reciben la *mask* y el hidato respectivamente. Proceden a comprobar si la posición es válida, devolviendo *True* si no lo es y *False* de lo contrario.

Algunas funciones de las anteriores utilizan otros métodos para hacer más coherente y sencillo el funcionamiento. Ejemplo: *changeState* utiliza *changeRow* el cual de manera muy descriptiva cambia la fila ubicando un nuevo número en cierta columna, los métodos *notValidMask* y *notValidHidato* utilizan *checkMask* y *checkHidato* respectivamente para iterar por las filas del tablero y luego una vez encontrada la fila deseada, hacer un llamado de *checkColumn* y *distintZero* respectivamente para encontrar la columna del tablero y así obtener el resultado deseado.

Además se utilizan útiles como *searchNumber* que localiza un número dado en el tablero recibido, devolviendo su posición, *searchTop* que dado un tablero devuelve el número máximo que se encuentra ubicado en este, *getRows* y *getColumns* que dada una matriz brinda el número de filas y columnas respectivamente que esta tiene, *searchUbicated* recibiendo un tablero retorna una lista de enteros que se encuentran ubicados en esta, *biggerThanOne* recibe una fila con enteros y devuelve una lista con los que son mayores que 1 y otros submétodos auxiliares que realizan pequeñas tareas pero contribuyen al funcionamiento del generador.

Solucionador de tableros

Acercamiento general. Recibiendo un tablero con valores fijados, el 1 y el máximo entre ellos, y la máscara correspondiente a dicho sudoku, el método *solve* se encarga de darle solución. Similar al generador esta función itera desde 2 al máximo pero en este caso se analiza que ocurre si se ubica el número en cada posición adyacente a su antecesor, comprobando siempre que sea válida, y luego se avanza a comprobar el sucesor.

Métodos y estrategias. El iterador solo se detendrá cuando se compruebe que el antecesor al máximo se encuentra bien ubicado, una vez haya ocurrido esto significa que tenemos la solución, de lo contrario se continua comprobando el resto de posiblidades.

Se utiliza la función *ubica*, que es la encargada de iterar por todos los números y analizar que ocurre si este se fija en cada posición válida adyacente al antecesor por medio de la función *validPlace*. Si se realiza un análisis cuidadoso es un método muy similar a *giveNumber*, con la diferencia que *ubica*, por conveniencia lo que retorna es un booleano y la matriz solucionada, y no un entero con el número de soluciones. Se utiliza en el resultado ese primer parámetro booleano para en cada llamado recursivo recibir si se encontró solución de ser *True* y *False* en caso contrario.

- *ubica:* como se decribió superficialmente, recibirá un entero x, la posición de su antecesor, el máximo del tablero, la lista de los valores ubicados inicialmente, el sudoku sin resolver y su *mask*. Este de manera recursiva utilizando *validPlace* al igual que *giveNumber*, con la pequeña diferencia que este devuelve la tupla bool-tablero (Bool,[[Int]]), iterará por cada número hasta el máximo ubicando x de manera adyacente a su antecesor y procediendo a comprobar el sucesor. Lo anterior procede ubicando siempre en una posición válida, ya descrita en el generador.

El solucionador utiliza al igual que el generador funciones como *changeState*, *correctlyPlaced*, *searchTop*, *searchNumber*, *getRows*, *getColumns*, *notValidMask*, *notValidHidato* y muchos otros con exactamente la misma finalidad.

Templates

Se define un template como la máscara booleana de un tablero hidato. Este para ser añadido se necesita saber que es válido, y esto se puede comprobar buscando al menos una casilla del tablero desde donde se puede formar un camino de Hamilton, por lo que se utiliza la función *searchHamiltonianPath:* antes añadida.

Ejecución del software

Para encontrar la informacioón de apectos más particulares de la ejecución del software se recomienda dirigirse al directorio raíz del proyecto y apoyerse en la documentación brindada en el *README.md*.