Documento de Ejemplo para la Jornada Científica Estudiantil

Daniel de la Osa Fernandez D.OSA@ESTUDIANTES.MATCOM.UH.CU
Grupo C-411

Resumen

Con este proyecto se quiere mostrar una implementación de una solución para una variación del Sudoku donde las figuras que conforman el tablero de $9X9$ son figuras conexas que pueden tener diferentes formas no solo las típicas cuadradas $3x3$. Esto se logró usando el lenguaje Haskell haciendo uso del paradigma declarativo-funcional muy acorde a la naturaleza combinatoria del problema

1. Introducción

El problema resolver consiste en: dado un conjunto de nominoes, que no son mas que las piezas que conforman el Sudoku, deben ser encajadas de forma que formen el tablero clásico del Sudoku, un cuadrado de $9X9$.

Luego de haber logrado este tarea se precede a resolver el Sudoku resultante llenando todas las casillas en blanco del $1-9$. Todo esto cumpliendo las restricciones del juego de que no puede haber números repetidos en la misma fila columna o nomino que conforma el tablero.

Como se puede apreciar el problema pasa por un problema combinatorio que se reduce primeramente a de cuantas formas puede colocar los nominoes en un cuadro de $9X9$. Y luego pasamos al otro, que sería de cuantas formas se puede fijar los números del $1-9$, en las casillas en blanco de forma tal que cumplan las restricciones del juego.

2. Detalles para la Implementación

Para poder abordar el problema con menor complejidad se declararon nuevos tipos y así lograr una mayor abstracción y facilitar la solución. Estos tipos son :

- *Nomino*
- *Sudoku*

Con estos tipos se modelaron los principales componentes del problema. Cada una se declaró usando records de Haskell y podemos verlo a continuación en este fragmento de código.

La base de los nominos son lista de tuplas donde el primer elemento es la coordenada i indicando la fila, el segundo es la coordenada j para indicar la columna , con estos indices se indica la posición de cada celda del nomino en el tablero. El ultimo elemento de la tupla es el valor de la casilla que puede ser 0 si es vació o un número del 1 al 9.

Por otro lado el Sudoku esta conformado por una lista de nominos, un tablero representado por una matriz matemática clásica y una matriz de posibilidades que contiene en $[i, j]$ la lista de posibles valores, dadas las restricciones del Sudoku, que puede tomar la casilla $[i, j]$ en el tablero, si ya tiene fijado un número pues su correspondiente lista es vacía.

3. Ideas para la Solución

3.1 Encajando Nominos

Para resolver el primer problema de hacer encajar los *nominos* se siguió una solución usando *backtracking* conforme a la naturaleza de este.

Para lograr esto se toma una figura (*Nomino*), se genera una posición valida dentro del tablero actual donde se quieren encajar todas los *nominos*. Luego se coloca en esa posición y se vuelve a repetir el proceso con la próxima figura y el tablero resultado de poner la anterior.

Tener en cuenta que en la implementación de este algoritmo se uso la ventaja de la *evaluación perezosa* de Haskell para disminuir los cómputos y además como las posibles posiciones de la figura i se ve disminuida en cada iteración dado que el tablero donde lo tiene que colocar ya tiene $i-1$ figuras fijadas se hace una gran poda en la generación de las combinaciones posibles.

Si en algún momento se logran poner 9 figuras entonces se ha logrado formar un Sudoku por lo que ya queda resuelto este primer problema. Y para completarlo se calcula el campo *posibilidades* y se crea un Sudoku listo para resolver.

3.2 Resolviendo Sudoku

Este problema es abordado también mediante una solución recursiva. La primera idea que se pudiera pensar es elegir de todos los posibles pero eso rápidamente pudiera ser infactible ya que si digamos que existan 50 casillas vacías, esto nos pudiera llevar a tener que generar por esta vía

5153775207320113310364611297656212727021075220001 tableros de Sudoku por lo que se desecha esto en parte.

Se quiere lograr una manera de que al fijarse una celda repercuta en los posibles de las demás celdas vacías disminuyendo el número de tableros generados. Para lograr esto se siguen los siguientes pasos:

1. Se va actualizando el campo posibilidades del Sudoku actual hasta que no existan en ninguna posición $[i, j]$ una lista con un solo elemento posible, es decir fijar las casillas que tienen solo un posible valor ya que es seguro que ese va ahí dada las restricciones del Sudoku.

2. Con este nuevo tablero se entra en el caso base de la recursividad el cual tiene dos subcasos:

   - Está bloqueado el Sudoku, es decir existe un casilla $[i, j]$ del tablero que está vacía y ademas la lista de posibilidades para esa casilla esta vacía, por lo tanto dado las restricciones del Sudoku no puede ir allí ningún numero del 1 al 9. Por lo tanto se devuelve *Nothing* que indica que por esa rama del árbol que se eligió no se alcanzó la solución
   - Esta completo el Sudoku , llegamos a un estado donde todas las casillas están llenas y no existen conflictos. Alcanzamos una solución

3. Llegando al caso recursivo, primero se generan dos Sudokus nuevos a a partir del actual. Para lograr esto se elige primero una casilla pivote. Esta sera la que menor cantidad de posibles valores a colocar tenga de esta manera será más probable que se fije el correcto. A partir de aquí se generan dos nuevos subcasos para la generación de los dos nuevos sudokus:

   - La casilla pivote elegida tiene exactamente dos posibles valores por lo que los sudokus que se generan son el resultado de fijar un valor o el otro. Creando así dos caminos de decisión para seguir buscando otra solución.
   - La casilla pivote tiene más de dos posibilidades por lo que los tableros que voy a generar son uno con el primer valor de la lista de posibilidades fijado y el otro con la casilla vacía pero removiendo el valor que se fijó en el anterior de los posibles valores para esa casilla. De esta manera se garantiza que si fijando ese valor falló va a regresar a este punto se va a ir por la otra rama en la cual no va a poder fijar en esa casilla el valor que ya se fijo anteriormente.

   Con estos dos nuevos sudokus se llama recursivo con uno y con el otro comenzando el proceso de nuevo usando, de manera de que si el primero devuelve *Nothing* intenta irse por la otra rama del árbol de decisión. El algoritmo hace DFS por un árbol binario donde cada nodo representa una decisión tomada para llenar una casilla o eliminar una posibilidad de llenado para esta.

4. Uso

Para correr el programa es necesario el el programa stack con el se puede compilar el programa después de descargue las dependencias con el comando *stack build* y ejecutarlo con el comando *stack exec Sudoku-exe.exe*.

La entrada se puede cambiar en el $Main.hs$ dentro de la carpeta app. Ahi se pueden cambiar los nominos de entrada que no son más que listas de tuplas donde los primeros dos elementos son la posición inicial del nomino y el último valor es el numero del 1 - 9. Si el nomino se desea pasar con esa casilla vacía se le pasa un -1 en el ultimo valor de la tupla.

5. Recomendaciones

Analizar si es posible mejorar mejor este algoritmo de solución teniendo en cuenta el patrón de que si en una misma fila aparecen exactamente k valores que pertenecen unicamente a k lista de posibilidades es posibles fijarlos directamente a cada uno en las casilla correspondientes a esas listas ya que son ellos los que van seguro ahí.