Modelos Avanzados de Computación

Proyecto Final:

Rubik 2x2

Raúl Jiménez Suárez

Uadad Sidelgamun

Índice

1. Introducción

Este proyecto tiene como objetivo crear un simulador del cubo de Rubik 2x2 utilizando el lenguaje Haskell. La idea surgió como una forma de poner en práctica los conceptos básicos de programación funcional que aprendimos en clase. Aunque Haskell puede parecer complicado al principio, este proyecto busca demostrar que también es una herramienta poderosa para resolver problemas y crear programas interactivos.

El cubo de Rubik es un rompecabezas mecánico tridimensional que ha entretenido a personas de todo el mundo desde su invención. En esta versión simplificada (el modelo 2x2), se trata de un desafío más accesible pero igualmente interesante. Con este simulador, podemos representar el cubo, aplicar movimientos, mezclarlo, y resolverlo automáticamente. Además, hemos incluido un menú interactivo para que el usuario pueda explorar todas estas funciones de manera sencilla.

A través del desarrollo de este proyecto, hemos trabajado con conceptos como el manejo de tipos de datos, funciones recursivas, y listas, además de diseñar una interfaz básica en la terminal. Más allá de aprender Haskell, también hemos descubierto cómo descomponer un problema complejo en partes más manejables y resolverlas paso a paso.

2. Descripción del problema

El cubo de Rubik 2x2 consta de seis caras, cada una dividida en cuatro pegatinas de un mismo color. Resolver el cubo implica organizar las pegatinas de forma que cada cara tenga un único color.

En el simulador, las caras del cubo se representan mediante listas, y los movimientos posibles se modelan como transformaciones funcionales. El programa permite:

- Visualizar el estado actual del cubo.
- Mezclar el cubo mediante movimientos definidos previamente.
- Resolver el cubo utilizando un algoritmo de búsqueda que encuentra una secuencia de movimientos desde cualquier estado desordenado hasta el estado resuelto.

3. Diseño e implementación

3.1 Representación del cubo

El cubo se representa como una lista de listas caracteres, donde cada sublista contiene los colores de una cara:

```haskell
-- Representación del cubo
type Cara_Cubo = [Char] -- Cada cara tiene 4 pegatinas
type Cubo = [Cara_Cubo] -- Un cubo tiene 6 caras
```

El estado inicial del cubo (resuelto) se define así:

```haskell
-- Estado inicial del cubo (suponemos que esat resuelto)
estadoIni :: Cubo
estadoIni = [
    ['Y', 'Y', 'Y', 'Y'], --arriba
    ['R', 'R', 'R', 'R'], --front
    ['G', 'G', 'G', 'G'], --der
    ['O', 'O', 'O', 'O'], --izq
    ['B', 'B', 'B', 'B'], --tras
    ['W', 'W', 'W', 'W']  --abajo
    ]
```

3.2 Movimientos

Los movimientos se definen como un conjunto de transformaciones. Cada movimiento rota una cara y afecta las pegatinas de las caras adyacentes. Los movimientos básicos y sus inversos están modelados como un tipo de datos:

```haskell
-- definimos los movimientos que tenemos
data Movimiento = U | U'| F | F'| R | R'| L | L' | B | B'| D | D'
  deriving (Show, Eq)
```

Para cada movimiento, se implementa una función que aplica la transformación correspondiente al cubo, a continuación, se explica el movimiento de la cara superior en sentido horario, que para los demás ejemplos sigue una lógica similar:

```haskell
movArriba :: Cubo -> Cubo
movArriba [u, f, r, l, b, d] =
    [[u !! 1, u !! 3, u !! 0, u !! 2],  -- arriba
     [r !! 0, f !! 1, r !! 2, f !! 3],  -- frente
     [b !! 3, r !! 1, b !! 1, r !! 3],  -- derecha
     [f !! 0, l !! 1, f !! 2, l !! 3],  -- izquierda
     [b !! 0, l !! 2, b !! 2, l !! 0],  -- trasera
     d]                                 -- abajo
```

Los parámetros de la función son las listas que componen el cubo (cara superior, frontal, inferior...). La función anterior se encarga de rotar la cara superior del cubo 90º en sentido horario y ajusta las respectivas caras adyacentes intercambiando los elementos que la componen, y en consecuencia la cara inferior no se modifica.

La función devRotacion Q se encarga de aplicar el movimiento correspondiente al cubo pasado por parámetro invocando a una función específica dependiendo el movimiento que realiza la rotación:

```haskell
-- funcion para delvolver la rotacion aplicada
devRotacion :: Movimiento -> Cubo -> Cubo
devRotacion U a  = movArriba a
devRotacion U' a = movArriba' a
devRotacion F a  = movFrontal a
devRotacion F' a = movFrontal' a
devRotacion R a  = movDer a
devRotacion R' a = movDer' a
devRotacion L a  = movIzq a
devRotacion L' a = movIzq' a
devRotacion B a  = movTrasera a
devRotacion B' a = movTrasera' a
devRotacion D a  = movAbajo a
devRotacion D' a = movAbajo' a
```

3.3 Resolución del cubo

El algoritmo de búsqueda resuelve el cubo de Rubik mediante una búsqueda en anchura (BFS). Comienza desde el estado inicial del cubo y explora todas las configuraciones posibles, generando nuevos estados mediante movimientos válidos. Cada estado se representa junto con la secuencia de movimientos que lo lleva desde el estado inicial.

Para evitar ciclos, se mantiene un registro de los estados ya visitados. Si el cubo llega a su estado resuelto, se devuelve la secuencia de movimientos correspondientes. Además, el algoritmo tiene un límite de profundidad (max) para evitar búsquedas infinitas.

En cada paso, se expanden los estados sucesores, y si alguno coincide con el estado resuelto, se devuelve la solución. Si no se encuentra una solución dentro del límite, el algoritmo termina sin éxito.

```haskell
-- funciion para resolver el cubo mediante una busqueda
resolverCubo :: Cubo -> [Movimiento]
resolverCubo a = buscaSol a

buscaSol :: Cubo -> [Movimiento]
buscaSol a = aux1 [[(a, [])]] [] 0
  where
    max = 10
    aux1 [] _ _ = []
    aux1 (n:ns) v p
      | any (\(x, _) -> x == estadoIni) n = snd ( head ( filter (\(x, _) -> x == estadoIni) n ))
      | p >= max = []
      | otherwise = aux1 (ns ++ [sigE]) (v ++ map fst n) (p + 1)
      where
        sigE = [(y, m ++ [z]) | (c, m) <- n, (z, f) <- lista, let y = f c, y `notElem` v]
```

4. Funcionamiento del programa

El programa ofrece una interfaz interactiva que permite al usuario:

1. **Mostrar el cubo:** Visualiza el estado actual del cubo.
2. **Mezclar el cubo:** Desordena el cubo mediante el índice de la lista de movimientos de 0 a 11.
3. **Resolver el cubo:** Muestra la secuencia de movimientos necesaria para resolver el cubo y aplica esos movimientos para mostrar el cubo resuelto.

El menú principal se implementa de la siguiente forma:

```haskell
-- Bucle principal para interactuar con el usuario
menu :: Cubo -> IO ()
menu a = do
    putStrLn "-------- Opciones ----------"
    putStrLn "1. Mostrar cubo:"
    putStrLn "2. Mezclar cubo:"
    putStrLn "3. Resolver cubo:"
    putStrLn "4. Salir"
    putStr "Elige una opcion: "
    i <- getLine
    case i of
        "1" -> do
            muestraC a
            menu a
        "2" -> do
            putStrLn "\nintroduce le numero para generar un movimiento:"
            i2 <- getLine
            let x = read i2
            let a2 = R2_2.mezclar a x
            putStrLn "Cubo mezclado:"
            muestraC a2
            menu a2
        "3" -> do
           if R2_2.sol a
                then do
                    putStrLn "El cubo ya esta resuelto."
                    menu a
            else do
                let m = R2_2.resolverCubo a
                putStrLn "\nMovimientos para resolver el cubo:"
                pintarMovimientos m
                let c2 = R2_2.aplicaMovimientoCubo a m
                muestraC c2
                menu c2
        "4" -> putStrLn "Gracias por jugar."
        _ -> do
            putStrLn "Numero incorrectol, Intentelo de nuevo."
            menu a
```

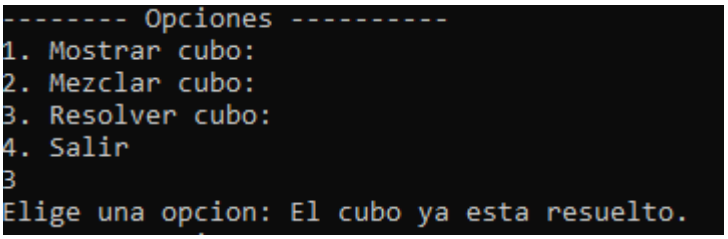
5. Pruebas realizadas

En esta sección vamos a ver cómo hemos comprobado que el simulador del cubo de Rubik 2x2 funciona correctamente. Para ello, hemos realizado varias pruebas, empezando por los casos más simples, como comprobar si detecta un cubo ya resuelto, hasta otros donde mezclamos el cubo y verificamos si puede resolverlo.

La idea principal de estas pruebas es asegurarnos de que todo funciona como debería, desde las mezclas hasta la resolución, y que el programa cumple con lo que esperábamos cuando lo diseñamos. A continuación, explicamos los casos que probamos y cómo salieron.

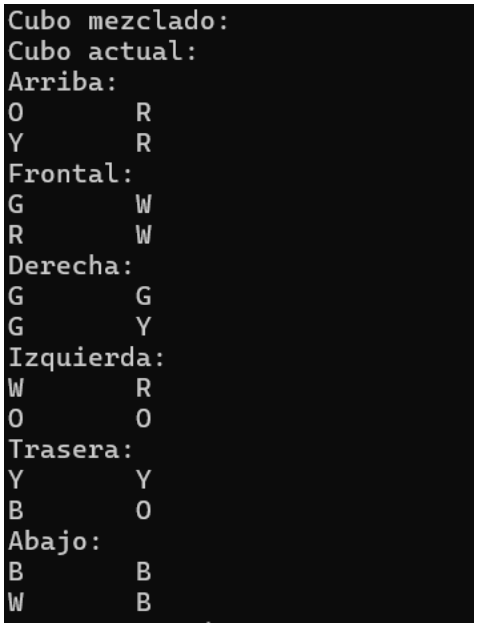
- **Caso 1:** Cubo ya resuelto

Entrada: Estado inicial del cubo. Salida: El programa indica que el cubo ya está resuelto.

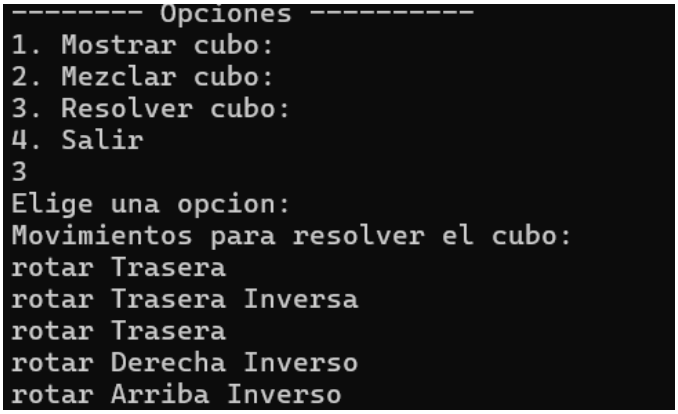


- **Caso 2:** Mezcla y resolución

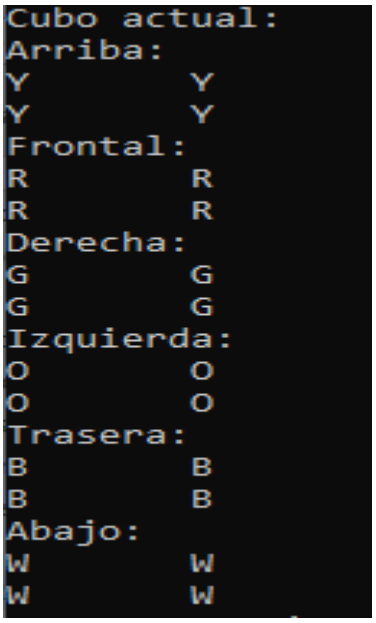
1. Se mezcló el cubo con 3 movimientos aleatorios.



2. El programa generó una secuencia para resolver el cubo.



3. El cubo volvió al estado resuelto tras aplicar los movimientos generados.



6. Conclusión

Este proyecto ha sido una experiencia interesante para adentrarnos en el mundo de Haskell y explorar cómo se pueden resolver problemas combinatorios de manera funcional. Aunque el algoritmo que implementamos para resolver el cubo de Rubik 2x2 es bastante sencillo, nos permitió entender conceptos clave como el diseño de algoritmos, el manejo de estructuras de datos y cómo trabajar con un lenguaje funcional.

A lo largo del desarrollo, nos enfrentamos a retos como representar el estado del cubo, simular sus movimientos y comprobar si estaba resuelto, lo cual nos ayudó a reforzar nuestra lógica de programación. Además, ver cómo todo encajaba y funcionaba al final ha sido muy satisfactorio.

Este proyecto nos deja una buena base para seguir explorando problemas más complejos en el futuro y para seguir sacándole provecho a Haskell, que, aunque al principio puede parecer un poco diferente a otros lenguajes, demuestra ser muy potente cuando se trata de resolver problemas como este.