Universidade Federal de Santa Catarina - UFSC
Departamento de Informatica e Estatistica (INE)
Disciplina: INE5416 - Paradigmas de Programação
Aluno: Lucas Broering dos Santos (22100909)

Relatório Kojun

Problema:

O Kojun é um puzzle lógico onde o tabuleiro é dividido em regiões, cada região precisa ter suas casas preenchidas do 1 ao tamanho da região. Ex: Região de tamanho 5, em suas cédulas terão valores de 1 … 5. Seria uma variação muito distante do sudoku. Ele possui algumas regras básicas, como:

- Não pode haver casas com valores repetidos adjacentes.
- Dentro da região não pode haver números repetidos na linha.
- Os valores de uma coluna na mesma região precisam estar em ordem decrescente.

Estratégias de Solução:

O código está dividido em dois módulos, um módulo simples que define alguns tipos para estruturação de uma Matriz e o módulo Kojun, eu também coloquei dentro da pasta do T1, um exemplo que desenvolvi de solução do jogo sudoku em Python e Haskell, utilizando backtracking.

```haskell
module Matriz(Valor, Linha, Matriz, linhas, colunas, ordem) where

import Data.List ( transpose )
import Data.List

-- Tipo que representa o valor de uma célula da matriz.
type Valor = Int
-- Tipo que representa uma linha da matriz.
type Linha a = [a]
--- Tipo que representa uma matriz.
type Matriz a = [Linha a]

-- Retorna a dimensao da matriz (ordem)
ordem :: Matriz a -> Int
ordem m = length (m !! 0)

-- Retorna as colunas de uma matriz
colunas :: Matriz a -> [Linha a]
colunas = transpose

-- Retorna as linhas de uma matriz
linhas :: Matriz a -> [Linha a]
linhas a = a
```

Aqui está o código do módulo Matriz, onde estão sendo definidos alguns tipos de dados, e funções para retornarem a ordem, colunas e linhas de uma matriz.

```
no commands
import Data.List

no commands
import Matriz

-- Representa o grid do jogo
type Grid = Matriz Valor

-- Representa os valores possíveis para cada célula
type Seleção = [Valor]
```

No módulo Kojun, existem mais esses dois tipos, um para representar o Grid (tabuleiro do jogo), que é uma matriz com valores e outro, Seleção para representar os valores possíveis para cada casa.

Esses foram os tipos criados para a solução do jogo Kojun, agora a lógica de solução, utilizei o backtracking, ou seja testa todas as possibilidades (tentativa e erro). Eu “dividi‘ o código em duas partes, a primeira foca mais em conseguir extrair as regiões e valores dos grids e a segunda aplica as soluções. Vou destacar algumas funções.

```
-- divide a matriz base em regiões seguindo a matriz de regiões
matrizRegioes :: Eq m => Matriz m -> Grid -> [Linha m]
matrizRegioes valores regioes = [regiaoFilter regiao tuples | regiao <- regioesMap ]
    where
        -- monta uma lista de tuplas com os valores e suas regiões
        tuples = montarTuples valores regioes
        -- agrupa as regiões
        regioesMap = nub (map snd tuples)
        -- filtra as regiões
        regiaoFilter regiao list = map fst $ filter ((== regiao) . snd) list
```

Essa função “matrizRegioes” retorna uma lista de linhas, cada lista é uma região com seus respectivos valores. Esse foi um jeito de juntar os dois grids.

```
-- valida adjacência, não pode haver valores iguais em células adjacentes
validarAdjacencia :: Eq a => Linha [a] -> Bool
validarAdjacencia [] = True
validarAdjacencia [x] = True
validarAdjacencia (x:y:xs)
    | length x <= 1 && length y <= 1 = (x /= y) && validarAdjacencia (y:xs)
    | otherwise = validarAdjacencia (y:xs)

-- valida linha, não pode haver valores iguais na linha
validarLinha :: Eq a => Linha [a] -> Bool
validarLinha [] = True
validarLinha (x:xs) = if (length x <= 1) then not (elem x xs) && validarLinha xs else validarLinha xs

-- valida linha em ordem descrecente
validarDescrescente :: Ord a => Linha [a] -> Bool
validarDescrescente [] = True
validarDescrescente [x] = True
validarDescrescente (x:y:xs)
    | length x <= 1 && length y <= 1 = (x >= y) && validarDescrescente (y:xs)
    | otherwise = validarDescrescente (y:xs)
```

```haskell
-- valida seguindo todas as verificacoes do jogo
validaMatriz :: Matriz Seleção -> Grid -> Bool
validaMatriz valores regioes = all validarAdjacencia (colunas valores) &&
                               all validarAdjacencia (linhas valores) &&
                               all validarLinha (matrizRegioes valores regioes) &&
                               all validarDescrescente (regioesColunas valores regioes)
```

Essas funções servem para para fazer todas as validações, seguindo as regras do jogo prescritas no problema. Validar o valor adjacente, validar que todos os valores sejam diferentes em uma linha e por último validar coluna decrescente.

```haskell
-- busca escolhas possíveis em cada espaço
buscaEscolhas :: Grid -> Grid -> Matriz Seleção
buscaEscolhas valores regioes = map (map escolha) (zipWith zip valores regioes)
    where
        -- escolhe um valor para o espaço, os valores já selecionados são removidos. As demais possuem valores referentes ao tamanho da região
        escolha (x, y) = if x == 0 then [1..tamRegiao y regioes] `minus` (valoresRegiao valores regioes y) else [x]
```

Essa função busca os valores possíveis para cada casa da matriz. As listas vão de 1 até o tamanho da região, menos os valores que já estão presentes naquela região.

```haskell
-- Faz a subtração entre duas listas
menos :: Seleção -> Seleção -> Seleção
xs `menos` ys = if apenasUmValor xs then xs else xs \\ ys
```

Função auxiliar menos que faz essa subtração, descrita acima.

```haskell
-- faz a filtragem das solucoes possíveis, seguindo a Seleção de valores
-- retorna uma lista de grids
buscaSolucoes :: Matriz Seleção -> Grid -> [Grid]
buscaSolucoes valores regioes
    -- quando não tem solucao retna uma lista vazia
    | semSolucao valores regioes = []
    -- quando todas as casas possuem escolha únicas, retorna a matriz
    | all (all apenasUmValor) valores = [map concat valores]
    -- expande as possibilidades de busca
    | otherwise = [x | valores' <- buscaExpandida valores, x <- buscaSolucoes (reduzirEscolhas valores' regioes) regioes]
```

A função buscaSoluções, retorna uma lista de grids solucionados. Ela faz a parte do backtracking, de tentativa e erro, caso alguma solução não contemple todas as regras do jogo.

```haskell
-- reduz escolhas usando a coluna dividida pelas regiões
reduzirEscolhas :: Matriz Seleção -> Grid -> Matriz Seleção
reduzirEscolhas valores regioes = colunas $ colunasOriginais (map reduzirEscolhasUnicas (regioesColunas valores regioes)) (ordem valores)
```

Aqui reduz as escolhas com as colunas divididas em regiões.

```haskell
buscaExpandida :: Matriz Seleção -> [Matriz Seleção]
buscaExpandida m = [linhas1 ++ [linha1 ++ [c] : linha2] ++ linhas2 | c <- cs]
    where
        (linhas1,linha:linhas2) = span (all apenasUmValor) m
        (linha1,cs:linha2) = span apenasUmValor linha
```

A buscaExpandida, faz a expansão da busca de soluções. Essa função do Haskell span, está fazendo a divisão da matriz em partes, os linhas1 contemplem os valores que satisfazem a condição e linha:linha2 são os valores que não satisfazem a condição. Condição essa que retorna True quando há apenas um valor e False quando tem mais que um.

Conclusão

A maior barreira para mim nesse desafio, vou me adaptar à linguagem funcional, como estruturar a solução por exemplo. Outra coisa que me deixou bem confuso em alguns momentos foi os métodos da linguagem para percorrer as estruturas, iterar pode ficar bem confuso às vezes. O que serviu de ajuda, foi fazer uma solução básica do sudoku em outra linguagem e depois tentar fazer em Haskell. Os exercícios passados em aula, como o sobre listas, foram essenciais.