UNIVERSIDADE FEDERAL DE SANTA CATARINA
Disciplina: Paradigmas de Programação
Professor: Maicon Rafael Zatelli
Alunas: Gabriela Furtado da Silveira e Samantha Costa de Sousa

Relatório Trabalho I - Programação Funcional - Haskell

Análise do Problema:

O problema resolvido foi o jogo Makaro, que consiste em um tabuleiro *n*x*n* que contém células brancas a serem preenchidas com números (sendo que alguns números já começam preenchidos), células pretas que são apenas regiões vazias ignoradas e células que apresentam setas apontando na direção vertical ou horizontal. Há também regiões de células brancas, delimitadas pelas linhas mais grossas.

Células de uma mesma região devem ser preenchidas com números de 1 a N (tamanho da região), sem repetições. Além disso, uma célula não pode conter o mesmo número que nenhuma de suas vizinhas ortogonais (estejam dentro ou fora da região). Por fim, a célula apontada por uma seta deve conter o maior número entre os vizinhos ortogonais da seta.

Solução:

O tabuleiro foi modelado na forma de uma matriz, que corresponde a uma lista de listas:

```
module Matrix (GenMatrix (Matrix), Position,
               getElement, changeElement, getColumnsNumber, getRowsNumber,
               addElement, getRow, printMatrix, getListFromMatrix, deleteFirst,
               changeElementList, addElementList) where

import Data.List

data GenMatrix t = Matrix [[t]]
            deriving (Eq, Show)
```

Durante a leitura do tabuleiro, são criadas duas matrizes tipo Matrix: "*regionsMatrix*", contendo as regiões (Matrix de String) e "*certaintyMatrix*", contendo os valores do tabuleiro (matriz de Int). Após, é criada uma terceira matriz denominada "*regions*" (que é uma lista de listas, não necessariamente uma matriz, mas utilizamos a mesma estrutura com o propósito de reaproveitamento do tipo e funções criadas). A estrutura "*regions*" contém o mapeamento das regiões na forma de listas de tuplas, nas quais a primeira tupla contém o número de elementos na região, seguida pelo número de posições não preenchidas, e as demais tuplas são as posições das células pertencentes à região.

Para a solução do puzzle, há primeiramente um conjunto de funções que preenchem os números de que se tem certeza.

```haskell
-- Chama a função fillMat1 para todas as regiões, de forma a preencher todasas regiões com apenas uma posição
-- não preenchida (só há uma opção para o número, nesse caso)
fillMissingOne :: Int -> (GenMatrix Int, GenMatrix Position) -> (GenMatrix Int, GenMatrix Position)
fillMissingOne i (mat, regions) =
    if i+1 < ((getRowsNumber regions)) then
        fillMissingOne (i+1) missing_num
    else
        missing_num
    where
        missing_num = (fillMat1 i (missingNumbers (getRow regions i) mat) mat regions)
```

```haskell
-- Chama a função fillMat2 para todas as regiões, de forma a preencher as regiões com apenas duas posições vazias
-- nas quais é possível determinar com certeza os valores de cada uma por causa de seus vizinhos ortogonais
fillMissingTwo :: Int -> (GenMatrix Int, GenMatrix Position) -> (GenMatrix Int, GenMatrix Position)
fillMissingTwo i (mat, regions) =
    if i+1 < ((getRowsNumber regions)) then
        fillMissingTwo (i+1) (fillMat2 i mat regions)
    else
       (fillMat2 i mat regions)
```

A função *fillMissingOne* atua sobre as regiões, preenchendo aquelas em que resta apenas uma única célula não preenchida com o número correspondente (como as demais estarão com outros números, se tem certeza do número que deve ser colocado). Já a função *fillMissingTwo* atua de forma parecida, porém preenchendo as células de regiões onde há apenas duas posições vazias, caso seja possível determinar com certeza qual número deve ir em qual posição em virtude dos vizinhos ortogonais.

Após obter a matriz de entrada preenchida com os casos de certeza, se inicia o algoritmo de backtracking. A solução possui duas lógicas de backtracking, uma interna à região e uma referente ao puzzle inteiro.

Dentro da região (módulo *SolveRegion*): após adquirir, a partir das estruturas existentes, os números possíveis e as posições para serem preenchidas, é feita a tentativa de preencher cada número em uma dessas posições. Ao conseguir preencher um número em uma posição (passou das validações, que é uma junção de funções que implementam as regras do puzzle, no módulo *Validations*), ele passa adiante com o próximo número e as posições restantes. Ele faz isso até preencher a região inteira, guardando o caminho em uma estrutura específica para isso.

- Um caminho interno à região consiste de uma lista de inteiros referente a indexação das possibilidades de preenchimento na região. Ou seja, numa região que existem 3 casas a preencher, existem inicialmente 9 possibilidades, que são geradas pelo código (um número em uma posição, tipo "*Possibility*"). Então um caminho nesse caso poderia ser [0, 3, 6], esses números referentes ao índice da lista de possibilidades.

Porém, se chegar em um ponto no qual não é possível preencher o número da vez em nenhuma das posições restantes (não passa nas validações) então ele retorna ao preenchimento do número anterior. Para manter o controle do que falhou e não repetir o preenchimento errado, existe uma estrutura auxiliar *errorList*, que, para cada número, possui uma tupla (tipo "*RegionError*") de:

- (Caminho acima do preenchimento atual, Posições que deram errado nesse ponto)

Essa lógica é mais visualizável se imaginarmos o preenchimento como a geração de uma árvore. As posições referente àquelas possibilidades só são uma falha naquele exato ponto da árvore, por isso é salva uma parte do caminho junto com o que causa a falha, para ser feita a verificação. Por exemplo:

- O caminho atual é [0, 4]. A próxima tentativa de preenchimento falha. Então ele salva na lista de erros, no segundo elemento da lista (referente ao segundo número a ser preenchido no caminho): ([0], [Posição referente à possibilidade 4]).

Após isso são feitas as limpezas necessárias e, ao tentar preencher um número, ele sempre exclui as posições que estão alocadas na sua lista de erros (se caminho atual == caminho salvo na lista de erros) das posições possíveis. Assim, esse erro não é repetido.

```haskell
-- Dado um caminho, verificar se não há posições falhas a serem removidas nesse ponto da árvore.
-- (a partir da errorList). Retornar a lista de posições alterada (ou não).
removeErrorPositions :: [Int] -> [RegionError] -> [Position] -> [Position]
removeErrorPositions path errorList possiblePositions
    | isCurrentError || isFirstNumberError = removeItemsFromList possiblePositions (snd currentError)
    | otherwise = possiblePositions
    where
        order = length path -- Referente ao indice do número na lista de erros
        currentError = errorList!!order -- É uma tupla, tipo RegionError
        lenErrorList = length (snd currentError)
        isCurrentError = order > 0 && lenErrorList > 0 && (fst currentError) == path
        isFirstNumberError = order == 0 && lenErrorList > 0
```

(Exclusão de posições falhas)

Todo esse procedimento pode, no fim, gerar uma região válida naquele ponto e retornar "*True*" com as devidas estruturas alteradas, ou, ao não conseguir um caminho possível, retornar "*False*" para falha total da região.

A captura da falha total da região ocorre no outro backtracking, externo à região (módulo *Solve*). Ele segue uma lógica muito semelhante ao backtracking interno, exceto que:

- O caminho atual que vai se formando é uma lista dos caminhos internos da região. Exemplo: [[0], [1, 2], [2, 4, 7]].
- Os erros associados à *região* no puzzle não são posições, e sim caminhos de região. Ex, o caminho [2, 4, 7] seria considerado um caminho "falha" para aquele ponto caso a próxima região obtivesse falha total. Esse valor deve ser passado em uma lista de "caminhos errados" para a região anterior à falha, que vai tentar ser preenchida novamente de uma forma que não gere algum caminho que cause falha nesse ponto da árvore.

O backtracking interno à região, então, possui mais uma verificação enquanto faz suas tentativas de preenchimento de números. A partir de uma lista de "caminhos falhas" (que não é vazia caso eu esteja tentando preencher aquela região mais uma vez no mesmo ponto da árvore), ele verifica, a cada preenchimento, se não está seguindo algum desses caminhos falhas, e tenta alterar o preenchimento; Caso sim, ele continua com o novo preenchimento que evitou um caminho falha, caso não, ele continua. No fim do preenchimento de região, há uma verificação específica para ver se o único caminho que a região conseguiu preencher é um que já falhou, então também retorna falha total para a região.

```haskell
-- Enquanto existir números para preencher, vai para o backtracking.
-- Se não há, verifica se todas as posições foram preenchidas ou se o caminho final é um que já falhou.
fillWholeRegion :: GenMatrix Int -> GenMatrix String -> [Int] -> [Position]
                -> [Position] -> [Possibility] -> [Int] -> [RegionError]
                -> GenMatrix Int -> (Bool, GenMatrix Int, [Position], [Int])
fillWholeRegion mat matRegions possibleNumbers possiblePositions region possibilities path errorList wrongPaths
    | (((length possibleNumbers) == 0) && ((getSecond (region!!0)) /= 0)) =
      (False, mat, region, path)
    | (((length possibleNumbers) == 0) && ((getSecond (region!!0) == 0))) =
      ((not (elem path (getListFromMatrix wrongPaths))), mat, region, path)
    | otherwise =
      backtrackingTryFillNumber mat matRegions possibleNumbers possiblePositions region possibilities path e
```

(A verificação de ter só conseguindo repetir um caminho que já falhou se refere ao "*path*" ser igual a um caminho falha existente na "*wrongPaths*", verificado com a função *elem*)

Entrada e resultado:

O programa recebe como entrada um tabuleiro do jogo Makaro na forma de um arquivo de texto com extensão ".txt", cujo nome é passado na *main* obedecendo o seguinte formato:

- Na primeira linha há apenas um número *n* que corresponde ao número de linhas do puzzle (ex: para um tabuleiro 4x4, *n* = 4).
- As próximas *n* linhas devem conter as linhas do puzzle representadas de acordo com as regiões: todas as células pertencentes a uma mesma região devem ser representadas pelo mesmo número, sendo que os números correspondentes a cada região são atribuídos em ordem crescente (1, 2,…), da esquerda para a direita e de cima para baixo conforme a primeira aparição de uma célula da região. Para células pretas (vazias) deve ser colocado um "X" na posição e para as setas, devem ser usadas letras que representem o sentido para o qual apontam: L - esquerda (left), R - direita (right), U - cima (up), D - baixo (down).
- As *n* linhas seguintes devem conter os números de cada célula, utilizando 0 para células vazias (brancas ou pretas) e para setas.

Exemplo de entrada para um tabuleiro 8x8:

8
1 D 2 3 3 4 4 4
5 5 2 6 D U 4 U
R 5 5 6 7 7 8 8
9 5 U 10 7 11 L 12
9 10 10 10 11 11 11 12
9 X D 13 U 11 X 12
14 14 13 13 15 15 15 D
16 16 16 X 17 17 15 15
0 0 0 2 0 0 3 4
0 5 2 0 0 0 0 0
0 0 3 0 3 0 0 0
0 0 0 1 2 0 0 3

0 3 2 4 0 5 0 0
3 0 0 0 0 0 0 1
0 2 0 0 0 4 0 0
2 3 0 0 2 0 3 0
(puzzle disponível em: https://www.janko.at/Raetsel/Makaro/001.a.htm)

O puzzle resolvido é apresentado na forma da impressão no terminal da matriz *n*x*n* contendo os números da solução.

Organização:

Gabriela trabalhou mais na parte de implementação inicial no Haskell, criando as estruturas de dados necessárias (*Matrix*, *Position*) e desenvolvendo as funções que realizam a leitura do arquivo e mapeamento das regiões. Além disso, implementou o módulo de certezas e auxiliou em algumas outras atividades.

Samantha implementou as validações em Haskell, além de ter desenvolvido a lógica e o código da parte que realiza o backtracking propriamente dito, com uma série de funções que permitem resolver o puzzle.

A resolução foi feita majoritariamente em chamadas de áudio no aplicativo Discord com o auxílio da extensão LiveShare do VSCode que permite desenvolvimento colaborativo em tempo real.

Dificuldades e soluções

Inicialmente, não estava claro como seria feito o backtracking e algumas funções de forma recursiva pela baixa experiência com Haskell e o paradigma funcional, então fizemos a resolução em Python, que facilitou os testes e *debug* para obter uma lógica sólida do backtracking e das funções auxiliares. Entretanto, mesmo na resolução em Python, muitos erros foram encontrados, mas superados com tempo de desenvolvimento.

Na transição para Haskell, houve muitos erros de sintaxe, de tipos e de passagem de argumentos, já que as “variáveis” são imutáveis, e não há um “estado” no paradigma funcional, e sim a passagem de argumentos alterados por outras funções. Era comum passar, por engano, um argumento inalterado quando queríamos passar um modificado por uma função específica. Para encontrar esses problemas, utilizamos o *debug* do *ghci* (adicionando *breakpoints*) e análise de código, com muitas comparações com a resolução de Python, e também a ferramenta “*ChatGPT*” para entender as mensagens de erro (em tempo de execução e compilação) e dúvidas de sintaxe específicas do Haskell.

Os puzzles testados foram de tamanho 8x8, 12x12 e 15x15, com resolução aparentemente instantânea.