Trabalho 1 - *Solver* em Haskell para o jogo Makaro

INE5416 - Paradigmas de Programação

Matheus Dhanyel C. Roque e Pedro H. Aquino Silva

Março 2021

1 Análise do problema

O trabalho consiste em desenvolver um programa em linguagem Haskell que encontre uma solução para um *puzzle* do tipo Makaro.

Este jogo é um quebra-cabeças com números, que tem como base um tabuleiro $n \times n$, dividido em regiões — ou *tectons* — e com algumas posições bloqueadas com a cor preta, ou com uma seta. Nas posições bloqueadas, não é possível inserir números, e nas posições com setas, a regra afirma que a posição vizinha apontada pela seta deve conter o maior número dentre os vizinhos ortogonais da posição da seta. Além disso, dentro dos *tectons*, as posições livres devem ser preenchidas com cada número entre 1 e a quantidade de posições do *tecton*, sendo que dois números iguais não podem ser vizinhos ortogonais. Algumas posições são já preenchidas no tabuleiro de entrada.

2 Descrição da solução

2.1 Visão geral

De maneira geral, o funcionamento da solução pode ser separada em 3 tópicos:

- **Estruturas e entrada de dados:** Primeiramente é definido como os dados serão armazenados e como se estrutura a entrada de dados para que eles possam ser devidamente manipulados.
- **Checagem de valores válidos:** Em seguida, é necessário definir funções para o acesso dessas estruturas e para fornecer quais números podem ser alocados em cada posição.
- **Solução por tentativa e erro:** Agora que conseguimos obter os movimentos válidos para cada posição, podemos executar um algoritmo de tentativa e erro, até a completude da matriz de tabuleiro.

2.2 Estruturas e entrada de dados

Primeiramente foi decidido como seria feita a modelagem do *puzzle* Makaro em nosso programa. Após pesquisar e analisar as regras do puzzle, decidimos por uma abordagem que representa o puzzle como a junção de 3 elementos: uma matriz com os valores de cada espaço (tomando como 0 como espaço vazio e -1 para os espaços que não podem ser preenchidos, ou seja, bloqueados ou com setas); uma lista de todas as regiões do puzzle, sendo que cada região é uma lista de posições e uma lista contendo a localização e direção de todas as setas, com cada seta sendo representada como uma tupla de sua posição e a direção para onde aponta. Além disso foi criado um tipo `Puzzle` para trabalhar mais facilmente com coordenadas no código.

```haskell
type Pos = (Int, Int)
type Tectons = [Pos]
type Arrow = (Pos, Char)
type Data = [[Int]]
type Puzzle = (Data, [Tectons], [Arrow])
```

Dessa forma, um Puzzle pode ser inicializado no programa criando apenas algumas listas, como segue no exemplo de um simples problema de Makaro 3×3:

```haskell
let matriz = [  [0, 0, 0],
                [0, -1, 2],
                [1, 0, 0]]
let areas = [   [(0,0), (0,1), (1,0)],
                [(0,2), (1,2), (2, 2)],
                [(2, 0), (2, 1)]]
let setas = [   ((1, 1), 'n')]
let puzzle = (matriz, areas, setas)
```

Foi-se decidido que a instância de Makaro a ser solucionada deveria ser definida no módulo `Main` do trabalho, de modo que a entrada de dados é *hardcoded* no programa Haskell.

2.3 Checagem de movimentos válidos

Quanto à checagem de valores válidos, existem 3 critérios que tem de ser analisados. As posições válidas são limitadas: pelas regiões, pelos vizinhos ortogonais e pelas possíveis setas adjacentes. Para que o código fosse mais compreensível e menos inclinado a erros difíceis de rastrear esses 3 elementos foram computados separadamente e em seguida unidos numa única função:

```haskell
validMoves :: Puzzle -> Pos -> [Int]
validMoves puz pos =
    validMovesArrow puz pos \\ (invalidMovesOrto puz pos `union` invalidMovesTecton puz pos)
```

Dessa forma, a função `validMoves` retorna uma lista de todos os valores possíveis naquela posição. Na linha 3 do trecho acima podemos ver que a lista de valores em um espaço não preenchido corresponde a diferença entre a lista de valores possíveis em relação às setas com a união dos valores proibidos pelas regiões e vizinhos ortogonais.

A função validMovesArrow, que apenas chama a função auxiliar validMovesArrowChecker, avalia os valores possíveis pegando como base um vetor com todos os números que poderiam ser colocados naquela posição caso o tecton que a posição pertence estivesse vazio e em seguida aplicando filtros com base em quaisquer setas ortogonais a posição, isso por meio da função arrowMovesFilter.

```haskell
    --Filter a list of valid moves based on an Arrow
    arrowMovesFilter :: Puzzle -> Pos -> Pos -> [Int] -> [Int]
    arrowMovesFilter puz cellPos arrowPos values
        -- if it's an arrow that points to the cell we're evaluating
        | arrow && cellPos == direction = filter (>ortoMax puz arrowPos) values
        -- if it's an arrow that points somewhere else that's been filled
        | arrow && cellPos /= direction && arrowHeadValue > 0 = filter (< arrowHeadValue) values
        -- if it's an arrow that points somewhere empty (will be evaluated later) OR if it's not an arrow
        | otherwise = values
        where (arrow, direction, arrowHeadValue) = (isArrow puz arrowPos, getArrowHead (getArrow puz arrowPos), getValue puz (getArrowHead (getArrow puz arrowPos)))
```

Essa função de filtragem aplica 4 possíveis filtros: Se estamos checando uma seta que aponta para a célula inicial são mantidos apenas os valores maiores que o máximo ortogonal à seta (dado pela função ortoMax); Se a seta aponta para outro valor são mantidos apenas valores menores do que o apontado; Se a seta aponta para uma casa vazia são mantidos os valores menores ou iguais ao maior ao redor da seta; e por fim, se não se tratava de uma seta nenhum filtro é aplicado.

Com o retorno de validMovesArrow apenas é retirado deles todos os valores que existem ortogonalmente ou existem num mesmo tecton e enfim é retornado um vetor com todos os valores válidos para a dada posição.

2.4 Solução por tentativa e erro

A implementação realizada no trabalho é mais parecida com uma busca em largura dos possíveis desenvolvimentos de um tabuleiro que uma solução por *backtracking* no paradigma imperativo, por exemplo, seria. Para facilitar a explicação, podemos entender a computação da solução como uma árvore de sub-problemas. Para coordenar as tentativas e erros, temos a função `backtrack`:

```
backtrack :: Puzzle -> [Puzzle]
backtrack s
    -- solution
    | solved s = [s]
    -- deadtrail and can't develop further
    | invalid s = []
    -- concatenate recursively children nodes that can be developed
    | otherwise = concatMap backtrack $ children s
```

Aqui, temos três situações possíveis. O nodo atual avaliado contém uma solução válida, ou o nodo encontrou um caminho sem saída e a solução parcial avaliada não pode ser desenvolvida, ou o nodo ainda pode ser desenvolvido em mais um passo de computação, e neste caso, fazemos a chamada da função `backtrack` em cada filho gerado por `children`:

```
children :: Puzzle -> [Puzzle]
children s
    | pos == (-1,-1) = [s]
    | otherwise = [update s pos opt | opt <- validMoves s pos]
    where pos = nextOpenPosition s
```

Notavelmente, as funções `solved` e `invalid` são bastante simples. `solved` somente checa se a matriz do tabuleiro está completa, uma vez que a cada passo de computação do algoritmo de *backtracking* garantimos que só testamos posições válidas em dada posição. Enquanto `invalid` somente checa se na próxima posição a ser preenchida existem valores possíveis, isto é, se `validMoves` retorna uma lista vazia.

Esta implementação retorna uma lista com todas as soluções possíveis de uma instância de Makaro, contudo, não só não podemos garantir que exista uma ou mais soluções, como caso exista só queremos uma resposta. Portanto, podemos avaliar somente o primeiro item da lista retornada. E caso não exista solução (lista vazia), recebemos um Puzzle vazio.

3 Detalhes sobre a execução do trabalho

3.1 Divisão de tarefas

A parte inicial de pesquisa foi feita em conjunto, e envolveu o entendimento do problema, pesquisa sobre e formulação da solução, bem como a decisão de como nossos dados seriam armazenados e que tipos teriam de ser declarados para tal.

Após essa fase, a carga de implementação foi dividida de forma que ambos pudessem trabalhar independentemente, um responsável pelas funções relativas à checagem de que números são válidos em cada espaço vazio e o outro pelas funções relativas à resolução do puzzle através de *backtracking*.

Para facilitar o trabalho assíncrono entre as duas partes foi criado um repositório no Github para colocar os arquivos referentes ao trabalho. Houve comunicação de ambas as partes ao longo da implementação, de modo que erros que surgiram foram identificados e solucionados rapidamente. Desta forma, consideramos que a divisão de tarefas e cargas de trabalho foi satisfatória para ambos os membros.

3.2 Dificuldades

No geral, não encontramos grandes dificuldades no desenvolvimento do trabalho. As ideias utilizadas no trabalho, como as estruturas utilizadas para representar os *tectons*, e a estrutura geral do algoritmo de tentativa e erro foram extraídos de mais de uma fonte durante a pesquisa inicial de compreensão do problema,

Concluindo, fora o esforço inicial de estudo, o processo de implementação seguiu de forma relativamente tranquila.