UNIVERSIDADE FEDERAL DE SANTA CATARINA – UFSC

CENTRO TECNOLÓGICO

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

CURSO DE GRADUAÇÃO EM CIÊNCIAS DA COMPUTAÇÃO

RELATÓRIO - TRABALHO 1 – PARADIGMAS DE PROGRAMAÇÃO

EDUARDO DIAS GUTTERRES

ARTHUR MOREIRA RODRIGUES ALVES

BRYAN MARTINS LIMA

FLORIANÓPOLIS, 05 DE OUTUBRO DE 2019

O programa desenvolvido apresenta uma solução para o jogo Wolkenkratzer, ou em inglês, Skyscraper. O jogo consiste em um tabuleiro quadrado dividido em posições, linhas e colunas. Em um tabuleiro 5x5, é necessário que os números de 1 a 5 apareçam em todas as linhas e colunas, representando as alturas das “torres”, não havendo repetições. Em volta do tabuleiro, ao lado de algumas das posições, pode haver um número, representando quantas “torres” é possível enxergar estando naquele lugar e olhando por aquela linha ou coluna. Por exemplo, se for possível visualizar 2 torres, uma solução possível para tal linha ou coluna é uma torre de altura 4 logo à frente e em seguida uma torre de altura 5, com as outras em qualquer ordem logo atrás, que não serão vistas pois são mais baixas que as duas primeiras.

O programa consiste em: dados os números existentes em volta do tabuleiro, apresentar uma solução para ganhar o jogo. Ele se dá por meio de dois arquivos Haskell, um para resolução de tabuleiros 5x5 e outro para 6x6. A entrada do programa se dá diretamente no código fonte, onde a pessoa informa os números em volta do tabuleiro através de quatro listas: Uma para os números parte de cima, na parte de baixo, ao lado esquerdo e ao lado direito. A saída do programa é uma representação em forma de matriz do tabuleiro resolvido, apresentada no console do sistema operacional.

Para o desenvolvimento do programa, foi utilizado e modificado um programa de resolução do jogo Sudoku, que funciona de forma semelhante ao Wolkenkratzer, mas com algumas diferenças.

O programa funciona da seguinte maneira: existe um tipo de dados Cell, representando uma posição do tabuleiro, podendo ser um Fixed Int ou um Possible [Int], representando uma lista dos números ainda possíveis de serem colocados naquela posição. Existe também tipo Row, representando uma linha do tabuleiro, sendo uma lista de Cell, e um tipo Grid, representando o tabuleiro em si, sendo uma lista de Row.

O programa então recebe o tabuleiro vazio e tenta preenchê-lo, de modo que todas as linhas e colunas contenham números de 1 a 5 sem nenhuma repetição. Até este ponto o algoritmo funciona de maneira muito semelhante ao Sudoku,

diferenciado apenas pelo tamanho do tabuleiro e pelas subdivisões do tabuleiro que existem no Sudoku, mas não existem neste jogo. Além disso, no Sudoku já existem algumas posições pré-fixadas, enquanto no Wolkenkratzer o tabuleiro começa vazio.

Para realizar este preenchimento, é utilizada uma técnica de backtracking. As posições, ou Cells, contém uma lista de possibilidades de números que as podem ocupar. Quando um número é assumido como “certo” em uma posição, ele se torna um Fixed Int, permitindo que o preenchimento do resto do tabuleiro seja feito em volta desta informação. Caso a solução não seja correta, outro número da lista é testado e colocado como Fixed. E assim o programa vai rodando, até que todas as posições sejam ocupadas de maneira correta.

O passo final é verificar as alturas das torres, pois não basta apenas que os números não se repitam em linhas e colunas, mas também que sejam vistos os números corretos de torres a partir das posições definidas. Para isso, foram criadas algumas funções que verificam se o número de torres vistas é o correto.

A primeira função é a validate_row:

```
validate_row :: Int -> [Int] -> Int

validate_row _ [] = 0

validate_row pivot (head:tail) =

    if (head > pivot) then

        1 + validate_row head tail

    else

        validate_row pivot tail
```

Esta função é uma função recursiva. O primeiro parâmetro que ela recebe é o maior número visto anteriormente em uma linha, e o segundo é o restante da linha a ser percorrida. Inicialmente, a função é chamada passando 0 e a linha a ser conferida. Sendo assim, a primeira torre a ser visitada será sempre avistada, contando +1 na resposta, e a partir daí o resto da linha é verificado. Cada vez que uma torre mais alta

que a torre anteriormente mais alta for visitada, o algoritmo soma +1 e se repete para o resto da linha, dando como resultado final o número de torres vistas naquela linha.

A função então é utilizada nas funções validate_rows e reverse_validate_rows. Estas, por sua vez, recebem uma das listas de torres a serem vistas e o tabuleiro. Elas verificam se todos os números correspondem ao número de torres vistas, retornando True se esta condição se satisfazer. A função validate_rows é chamada passando as listas da esquerda e de cima, enquanto a reverse_validate_rows é chamada passando as listas da direita e de baixo. Isso acontece devido ao fato de as listas precisarem ser percorridas da esquerda para a direita, enquanto as torres vistas de certas posições não devem ser vistas apenas com esta direção e sentido. As funções se diferenciam pela transformação do tabuleiro e da resposta final de modo a torná-la compatível com o problema. Finalmente, é chamada a função validate_grid, que verifica se todas as listas de números retornam True quando verificadas com o tabuleiro atual:

```
reverse_validate_rows :: [Int] -> Grid -> Bool
reverse_validate_rows [] [] = True
reverse_validate_rows (a:b) (head:tail) =
  if a == 0 then
    reverse_validate_rows b tail
  else
    if a == validate_row 0 (reverse [x | Fixed x <- head]) then
      reverse_validate_rows b tail
    else
      False


validate_rows :: [Int] -> Grid -> Bool
validate_rows [] [] = True
validate_rows (a:b) (head:tail) =
  if a == 0 then
    validate_rows b tail
  else
```

```
    if a == validate_row 0 [x | Fixed x <- head] then
      validate_rows b tail
    else
      False

validate_grid :: [Int] -> [Int] -> [Int] -> [Int] -> Grid -> Bool
validate_grid left right up down grid =
     validate_rows left grid &&
     reverse_validate_rows right grid &&
     validate_rows up (Data.List.transpose grid) &&
     reverse_validate_rows down (Data.List.transpose grid)
```

Caso isso aconteça, a solução está encontrada. Se não acontecer, o programa tenta encontrar outra solução, repetindo o processo.

A maior dificuldade para o desenvolvimento do programa foi justamente pensar na maneira que seria utilizada para a resolução do problema. Outra dificuldade foi o domínio do assunto de mônades, que é bastante utilizado na resolução do Sudoku e também utilizado nas funções criadas posteriormente. Após algumas tardes reunidos, finalmente surgiu um “estralo” e conseguimos pensar na solução de deixar o tabuleiro “pronto” e só então verificar as condições das alturas das torres.