Universidade Federal de Santa Catarina
INE5416 - Paradigmas de Programação

# RELATÓRIO TRABALHO I - SUGURU

Murillo Cordeiro Guindani
Vitor Silvestri

Florianópolis, 12 de Dezembro de 2021

# Análise

O suguru é um jogo de tabuleiro que consiste em preencher células com números seguindo duas regras. A primeira, dois números iguais não podem ser vizinhos horizontal, diagonal e verticalmente. A segunda, um bloco pode ser composto apenas de números que não se repetem entre 1 e o tamanho do bloco. O tabuleiro possui blocos com diferentes formatos e números de células sendo estas sempre adjacentes em algum grau. Ao iniciar o jogo, já existem células com algum valor nos blocos, o que reduz as opções do bloco e dos vizinhos da célula.

Nesse sentido, resolver o jogo a partir das células com maiores restrições deve ser a maneira mais lógica para soluções que se baseiam em tentativa e erro. Isso pois, células podem pertencer a um bloco com poucas escolhas disponíveis ou estar em uma vizinhança com poucas alternativas, dois fatores que tornam iterações arbitrárias ineficientes. Essa ideia é uma abstração das quatro técnicas de solução sugeridas no website que descreve o jogo disponibilizado no enunciado. www.janko.at/Raetsel/Suguru.

# Solução

Foi utilizado o backtracking como estratégia de resolução do jogo. Para a desenvolver o código fonte de soluções para sudoku com backtracking foram analisados. O arquivo Solver.hs possui as 7 funções que realizam a solução do tabuleiro. Em geral, a monad Maybe foi utilizada para validar as funções. A seguir breves descrições das funções por tópico.

# Backtracking

Uma breve descrição do backtracking consiste nas interações entre **solve** e **solve'.** A função **solve** verifica se o tabuleiro já está resolvido e caso não esteja, faz uma chamada para **solve'** que recebe como parâmetros a posição e as opções de uma célula. Esta célula é selecionada pela função **getNextCell** que retorna a célula com maiores restrições do tabuleiro. Em seguida, **solve'** realiza o backtracking, caso receba uma lista vazia de opções retorna Nothing, senão, chama **writeBoard** que escreve o valor da cabeça da lista de valores possíveis e realiza uma chamada para **solve** com a matriz sobrescrita como parâmetro. Se **solve** tem Nothing como resultado, faz uma chamada recursiva excluindo o valor utilizado anteriormente, senão retorna o tabuleiro.

Abaixo trechos de **solve** e **solve'**:

```
solve :: Board -> Maybe Board
solve board
  | isSolved board = Just board
  | otherwise = do
                pos <- getNextCell board
                options <- getCellOptions board pos
                solve' board pos options
```

```
solve' :: Board -> Position -> Options -> Maybe Board
solve' board pos [] = Nothing
solve' board pos (value:options) = do
                                   b <- writeBoard board pos (Just value)
                                     case solve b of
                                     Nothing -> solve' board pos options
                                     b -> b
```

## Célula com maiores restrições

A procura pela célula com maiores restrições é realizada pelas funções **getNextCell** e **getNextCell'**. A função **getNextCell** Itera sobre todas as celulas do tabuleiro retornando as posições que ainda não possuem valor definido. Caso a lista *posList* esteja vazia retorna Nothing, senão, chama foldr1 tendo **getNextCell'** e a lista de retorno como parâmetro. A execução de **getNextCell'** sobre a lista de potenciais posições resulta na posição com menores possibilidades ou maiores retrições.

Abaixo, trechos de **getNextCell getNextCell':**

```
getNextCell :: Board -> Maybe Position
getNextCell board = do
        case posList of
          [] -> Nothing
          n -> foldr1 (getNextCell' board) n
          where
            posList = [Just (i, j) |
              i <- [1..(lb)],
              j <- [1..(lb)],
              isNothing (getValueFromPos board (i, j))]
            lb = length board

getNextCell' :: Board -> Maybe Position -> Maybe Position -> Maybe
Position
      getNextCell' board Nothing _ = Nothing
      getNextCell' board _ Nothing = Nothing
      getNextCell' board (Just p) (Just q) = do
        a <- getCellOptions board p
        b <- getCellOptions board q
        Just(if length a > length b then q else p)
```

## Levantamento de alternativas para celula

A função **getCellOptions** chamada por **solve** retorna os valores possíveis para a célula de posição desejada. Ela utiliza a função **getBlockOptions** para varrer o bloco da célula e verificar quais valores já foram tomados por outras células no mesmo bloco. Em seguida, compara o resultado com os vizinhos e retorna uma lista com as opções válidas ou Nothing se a lista retornada for vazia.

A seguir, código fonte das funções:

```
getCellOptions :: Board -> Position -> Maybe Options
getCellOptions b p = do
  Cell block _ <- getAt b p
  blockOptions <- getBlockOptions b block
  case filter (`notElem` getNeighborValues b p) blockOptions of
    [] -> Nothing
    n -> Just n

getBlockOptions :: Board -> String -> Maybe Options
getBlockOptions board block =
  getBlock board block >>= (\c -> Just(filter
                                        (`notElem` mapMaybe
getCellValue c)
                                        [1..(length c)]
                                      ))
```

## Verificação de solução

A função **isSolved** é responsável por verificar se o tabuleiro foi resolvido e parar a recursão de solve'. Para tanto, ela é verdadeira somente se não existirem células com o valor *Nothing* no tabuleiro.

```
     isSolved :: Board -> Bool
     isSolved board = not (any (\(Cell _ value) -> isNothing value)
(concat board))
```

## Entradas e Saídas

Após iniciar o programa, o usuário deve inserir o tabuleiro desejado via teclado. Há opções de tabuleiros de dimensões  6x6, 8x8 e 10x10. Após escolhido, o programa imprime o nome do arquivo escolhido e o conteúdo do arquivo. Em seguida, ao terminar as computações, o programa imprime no terminal uma solução para o tabuleiro ou uma mensagem de erro.

## Sobre o grupo

O grupo optou pelo desenvolvimento assíncrono utilizando o GitHub para controle de versão e o Slack para comunicação. O desenvolvimento ocorreu sem a fase de projeto, isto é, através de tentativas e erros criou-se uma implementação que chegava mais perto de uma solução. Em geral, Vitor foi responsável pelas implementações com melhores resultados enquanto Murillo realizou algumas tentativas mas focou na pesquisa sobre eventuais métodos de solução e como os implementar utilizando monads disponíveis na linguagem Haskell.

## Dificuldades

Entre as dificuldades encontram-se questões de tipagem, imutabilidade de variáveis, entendimento do conceito de monads e mudança para o paradigma declarativo.

Com relação a tipagem, a tipagem forte e estática do Haskell dificultou a testagem dentro de funções. Em outras linguagens é possível simplesmente inserir um print() em qualquer lugar para entender o que se passa em determinada linha. Já a impressão na linguagem Haskel deve ser realizada em um monad IO, o que dificulta entender a manipulação de dados intra função. Embora não tenha sido utilizado pelo grupo, um debugger facilitaria muito na resolução de alguns destes problemas.

Também, a utilização de um sinônimo de tipo para definir a matriz como uma lista de listas, impediu a implementação de Show (type class responsável pela função “show”) para o tipo Matrix. Então, preferiu-se utilizar uma função customizada para realizar a impressão das matrizes.

Sobre a imutabilidade de variáveis, houve certa dificuldade ao modificar a matriz de células do tabuleiro, que precisaria ser copiada levando em conta os elementos modificados. Para isso, o grupo montou a função **setAt** que é responsável por copiar a matriz modificando apenas um elemento. Segue trecho da função:

```haskell
setAt :: Matrix t -> Position -> t -> Maybe (Matrix t)
setAt matrix (i, j) value
 | validPosition matrix (i-1, j-1) = Just (start++(x++value:ys):end)
 | otherwise = Nothing
 where
   (start, row:end) = splitAt (i-1) matrix
   (x, _:ys) = splitAt (j-1) row
```

Acerca das Monads, o entendimento do sequenciamento e das estruturas criadas por monóides demanda aprofundamento.  A implantação do backtracking para solução de computações não determinísticas, em particular, demandou bastante trabalho. Foram implementadas Monads de formas diferentes que em momentos nem compilaram. Afinal optou-se pelo uso do Monad Maybe por sua simplicidade. O paper *Reinventing Haskell Backtracking* e conteúdos do website *School of Haskell - Basics*, ajudaram a absorver melhor os conceitos relacionados as monads.

Por fim, no que se refere ao paradigma declarativo, houve dificuldade em entender por onde começar no problema. Foram estudadas soluções de jogos similares e lidos diversos papers para que se tivesse uma perspectiva. Alguns, como *Solving Suguru using an SMT (Integer) solver* propõe programar a célula com as restrições a fim de encontrar soluções. Outros, como *Sudoku Simple Solver* busca todas as possíveis soluções similar a uma abordagem *brute force*.