



CAMPUS DE FLORIANÓPOLIS
RELATÓRIO DO TRABALHO I DE PARADIGMAS DE PROGRAMAÇÃO - INE5416

João Paulo A. Bonomo
Gabriel Lima Jacinto
Rodrigo Santos de Carvalho

Florianópolis
2023



Departamento de
Informática e Estatística
CTC • UFSC



CENTRO TECNOLÓGICO
Universidade Federal de Santa Catarina

RELATÓRIO DO TRABALHO I DE PARADIGMAS DE PROGRAMAÇÃO - INE5416
PROFESSOR MAICON RAFAEL ZATELLI

Resumo

Este relatório é referente ao primeiro trabalho da disciplina de **Paradigmas de Programação - INE5416**, do trio João Paulo Bonomo, Gabriel Lima Jacinto e Rodrigo Santos de Carvalho. O trabalho em questão é um resolvedor do puzzle Vergleichssudoku, usando a linguagem de programação puramente funcional Haskell.

Sumário

1	INTRODUÇÃO AO PROBLEMA PROPOSTO	1
1.1	Introdução	1
1.2	Regras e Descrição do Jogo	1
2	DESCRIÇÃO DETALHADA DO PROGRAMA	4
2.1	Desenvolvimento da solução	4
2.2	Técnica de programação escolhida	4
2.3	Modelagem do tabuleiro	4
2.4	Aplicação	4
3	<i>INPUT E OUTPUT</i>	9
3.1	<i>Input</i>	9
3.2	<i>Output</i>	10
4	ORGANIZAÇÃO DO GRUPO	13
4.1	Comunicação do Grupo	13
4.2	Gerenciamento do Código	13
4.3	Ambiente de Desenvolvimento	13
5	DIFICULDADES ENCONTRADAS	14

1 Introdução ao Problema Proposto

1.1 Introdução

No arquivo de descrição do Trabalho I , o professor deixou a cargo dos grupos que escolhessem dentre três opções de puzzle:

1. Kojun;
2. Makaro;
3. **Vergleichssudoku.**

Nosso grupo optou pela terceira opção por uma questão de maior proximidade com o sudoku tradicional, e também pela maior simplicidade das regras (o que julgamos que simplificaria nossa implementação em Haskell, uma linguagem que ambos os 3 tínhamos pouco domínio).

1.2 Regras e Descrição do Jogo

Sucintamente, as regras do **Vergleichssudoku** são praticamente as mesmas do Sudoku tradicional, sendo acrescentada a presença de sinais de comparação ('>' e '<') entre as células do tabuleiro, o que torna o **Vergleichssudoku** mais fácil do que o próprio Sudoku, uma vez que diminui os números possíveis para cada uma das células no tabuleiro.

Um exemplo de tabuleiro 9x9 inicial de jogo pode ser visualizado na Figura 1, e a solução para esse exemplo pode ser vista na Figura 2.

Todos os tabuleiros que usamos para aprender a jogar, e também como "inputs" para testar nosso código foram retirados do site janko.at. Portanto, nosso trabalho consiste em resolver tabuleiros de tamanho 4x4, 6x6 e 9x9.

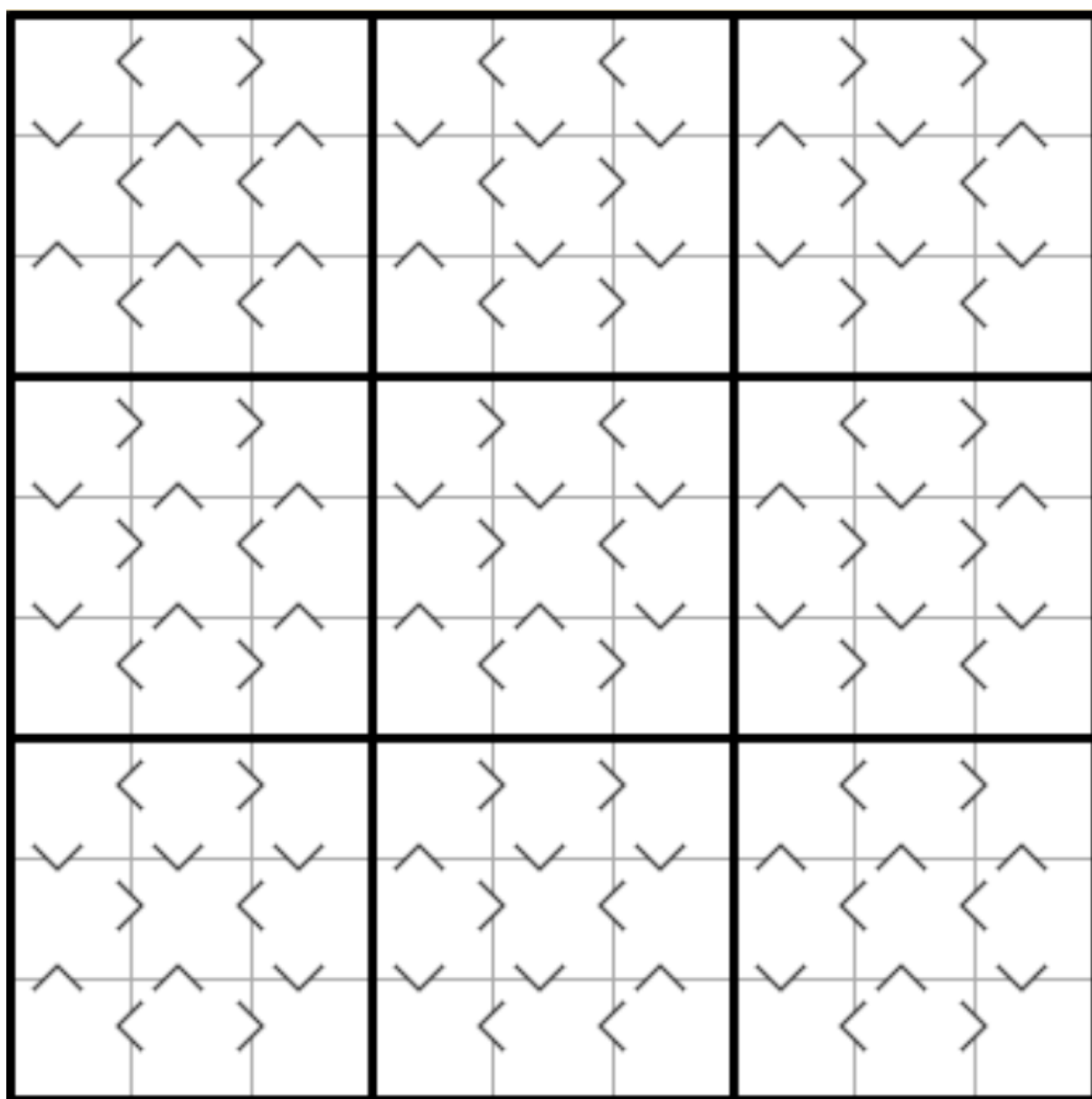


Figura 1 – Exemplo de tabuleiro 9x9 - retirado de janko.at/Raetsel/Sudoku/Vergleich/011.

3 < 4 > 2 ∨ ^ ^ 1 < 5 < 6 ^ ^ ^ 7 < 8 < 9	6 < 8 < 9 ∨ ∨ ∨ 3 < 7 > 2 ^ ∨ ∨ 4 < 5 > 1	7 > 5 > 1 ^ ∨ ^ 8 > 4 < 9 ∨ ∨ ∨ 3 > 2 < 6
9 > 2 > 1 ∨ ^ ^ 8 > 3 < 4 ∨ ^ ^ 6 < 7 > 5	5 > 4 < 7 ∨ ∨ ∨ 2 > 1 < 6 ^ ^ ∨ 8 < 9 > 3	6 < 8 > 3 ^ ∨ ^ 9 > 7 > 5 ∨ ∨ ∨ 2 > 1 < 4
4 < 9 > 8 ∨ ∨ ∨ 2 > 1 < 7 ^ ^ ∨ 5 < 6 > 3	7 > 6 > 5 ^ ∨ ∨ 9 > 3 < 4 ∨ ∨ ^ 1 < 2 < 8	1 < 3 > 2 ^ ^ ^ 5 < 6 < 8 ∨ ^ ∨ 4 < 9 > 7

Figura 2 – Resolução da Figura 1 - retirado de janko.at/Raetsel/Sudoku/Vergleich/011.

2 Descrição Detalhada do Programa

2.1 Desenvolvimento da solução

Para o desenvolvimento de nossa solução, nos baseamos no post [Solving ‘Greater-than sudoku’ with python and z3](#). O post traz uma solução em *Python* para o **Vergleichssudoku**, então nos bastou adaptar parte do código para *Haskell* e fazer algumas alterações na lógica para que funcionasse com tabuleiros de tamanhos diferentes de 9x9.

2.2 Técnica de programação escolhida

Apesar de inicialmente tentarmos usar algumas heurísticas em um algoritmo que estávamos desenvolvendo em *Python* para resolver o puzzle, trazê-las ao *Haskell* seria extremamente complicado pela quantidade de laços de repetição que usamos nesse algoritmo inicial. Portanto, após algumas tentativas frustradas de replicar essas heurísticas em *Haskell*, optamos por utilizar apenas a técnica de *backtracking* ("tentativa e erro") no desenvolvimento do projeto.

2.3 Modelagem do tabuleiro

A modelagem do tabuleiro pode ser vista de forma mais detalhada mais adiante na seção [3.1](#), onde tratamos do "input" do usuário no programa.

2.4 Aplicação

Ao arquivo **VergleichssudokuSolver.hs** cabe a função de resolver o puzzle. Tratemos abaixo mais detalhadamente de cada uma das funções desse módulo.

1. A Figura [3](#) trata das seguintes funções:

- A função *getXY* recebe uma matriz de elementos, um valor *x* e um valor *y*, e retorna o valor na posição (*x*, *y*) da matriz;
- A função *setXY* recebe uma matriz de elementos, um valor *r*, um valor *c* e um novo valor *val*, e retorna uma nova matriz que é a mesma que a matriz original, exceto que o valor na posição (*r*, *c*) é atualizado para *val*;

- A função *getVergleichssudokuGrid* recebe um tamanho de tabuleiro como entrada e retorna uma matriz de tamanho *sizeBoardXsizeBoard* inicializada com todos os valores como 0. Isso é usado para criar o tabuleiro inicial do Sudoku;
- A função *getRow* recebe uma matriz de elementos, um valor x e um valor y, e retorna a linha na posição x da matriz;
- A função *getCol* recebe uma matriz de elementos, um valor x e um valor y, e retorna a coluna na posição y da matriz;
- A função *getRegion* recebe uma matriz de elementos, um valor x, um valor y e o tamanho do tabuleiro como entrada, e retorna a região (submatriz) correspondente à posição (x, y) na matriz. O tamanho da região é calculado com base nas configurações incluídas pelo usuário no arquivo **SizeConfig.hs**.

2. Já a Figura 4 trata das seguintes funções:

- *compareBigger* e *compareSmaller* são funções auxiliares que comparam o valor em uma determinada posição com o valor em uma posição vizinha, de acordo com a direção fornecida como argumento. Essas funções retornam *True* se a comparação for verdadeira ou se a posição vizinha estiver vazia (valor = 0);
- Já a função *executeComparison* é responsável por executar uma comparação específica com base no símbolo de comparação fornecido. Se o símbolo for '?', a função retorna *True*. Se for '>', a função chama *compareBigger*. Se for '<', a função chama *compareSmaller*;
- Por fim, *getCompare* é responsável por obter todas as possíveis comparações que podem ser aplicadas em uma determinada célula/posição do tabuleiro. Ele obtém o comparador da posição correspondente no tabuleiro de comparações e, em seguida, para cada valor possível, executa todas as comparações possíveis chamando *executeComparison*. E então, a função retorna uma lista dos valores que podem ser colocados na célula correspondente.

3. A Figura 5, por sua vez, diz respeito às três seguintes funções:

- A função *getPossibleOptions* recebe uma matriz de **Vergleichssudoku** representada por uma lista de listas, uma matriz de caracteres que representa a mesma matriz de **Vergleichssudoku**, e as coordenadas de uma posição na matriz. A função retorna uma lista com os números possíveis para serem colocados nessa posição;
- Já a *getValueInList* recebe dois argumentos: uma lista e um inteiro. Ela retorna o valor do elemento i-ésimo da lista. Se o valor do inteiro for maior ou igual ao comprimento da lista, um erro é retornado;

- Por último, a função *getListLenght* apenas retorna o tamanho da lista de entrada.
4. Na Figura 6 temos uma das funções mais importantes do código: *solveVergleichssudoku*.
- A função é responsável por resolver o tabuleiro do puzzle. Ela recebe como entrada a grade do jogo *vergleichssudokuGrid* e a grade de comparadores *comparatorsGrid*, a linha e a coluna atual;
 - A função começa verificando se a célula atual é a última do tabuleiro. Se for, a solução é encontrada e a grade completa é retornada. Caso contrário, ela verifica se chegou ao final da linha e passa para a próxima linha. Em seguida, verifica se o valor da célula já foi definido e passa para a próxima célula, caso contrário, ela chama a função *solveVergleichssudokuWithValues* (tratada no tópico abaixo e visualizada na Figura 7, que testa os valores possíveis para a célula atual.
5. Por último, temos a função *solveVergleichssudokuWithValues*, que está representada na Figura 7.
- A função é responsável por testar os valores possíveis para a célula atual;
 - Se a lista de possíveis valores estiver vazia, não há solução possível para o tabuleiro de entrada. Se o índice ultrapassar o tamanho da lista de possíveis valores, não há solução possível para a célula atual. Em seguida, a função seta o valor encontrado na lista de possibilidades no índice recebido e chama a função *solveVergleichssudoku* para testar a próxima célula;
 - Se *solveVergleichssudoku* retornar *Nothing*, isso significa que não houve solução para a célula atual com o valor encontrado, então a função reseta o valor da célula atual e chama a si mesma para testar o próximo valor na lista de possíveis valores. Se *solveVergleichssudoku* retornar uma solução, ela é retornada pela função *solveVergleichssudokuWithValues*.

```

7 -- Retorna o valor de uma matriz em uma determinada posição.
8 getXY :: Show a => Maybe [[a]] -> Int -> Int -> a
9 getXY Nothing _ _ = error "getXY: Nothing" -- Caso base: matriz vazia.
10 getXY (Just grid) x y = grid !! x !! y -- Caso recursivo: retorna o valor da posição (x, y) da matriz.
11
12 -- Altera o valor de uma matriz em uma determinada posição (x, y).
13 setXY :: Maybe [[a]] -> Int -> Int -> a -> Maybe [[a]]
14 setXY Nothing _ _ _ = error "setXY: Nothing" -- Caso base: matriz vazia.
15 setXY (Just grid) r c val = Just (take r grid ++ [take c (grid !! r) ++ [val] ++ drop (c + 1) (grid !! r)] ++ drop (r + 1) grid)
16
17 -- Retorna a matriz inicial do tabuleiro, com o tamanho configurado pelo usuário em SizeConfig.hs.
18 getVergleichssudokuGrid :: Int -> Maybe [[Int]]
19 getVergleichssudokuGrid sizeBoard = Just (replicate sizeBoard (replicate sizeBoard 0))
20
21 -- Retorna a linha (x) na posição (x, y) do tabuleiro.
22 getRow :: Maybe [[Int]] -> Int -> Int -> [Int]
23 getRow Nothing _ _ = []
24 getRow (Just grid) x y = grid !! x
25
26 -- Retorna a coluna (x) na posição (x, y) do tabuleiro.
27 getCol :: Maybe [[Int]] -> Int -> Int -> [Int]
28 getCol Nothing _ _ = []
29 getCol (Just grid) x y = map (!! y) grid
30
31 -- A função getRegion recebe uma matriz de elementos, um valor x, um valor y e o tamanho do tabuleiro como entrada.
32 -- Ela retorna a região correspondente à posição (x, y) na matriz.
33 -- O tamanho da região é calculado com base nas dimensões fornecidas pelo usuário em SizeConfig.hs.
34 getRegion :: Show a => Maybe [[a]] -> Int -> Int -> Int -> [a]
35 getRegion Nothing _ _ _ = [] -- Caso base: matriz vazia.
36 getRegion (Just grid) x y sizeBoard =
37     [getXY (Just grid) i j | i <- [startRow..startRow + sizeRowRegion - 1], j <- [startColumn..startColumn + sizeColumnRegion - 1]]
38     where
39         startRow = x - x `mod` sizeRowRegion -- Linha inicial daquela região.
40         startColumn = y - y `mod` sizeColumnRegion -- Coluna inicial daquela região.
41

```

Figura 3 – Primeiras funções de /src/VergleichssudokuSolver.hs.

```

42 -- Função responsável pelas comparações de MAIOR QUE.
43 compareBigger :: Maybe [[Int]] -> Int -> Int -> Int -> Int -> Bool
44 compareBigger Nothing _ _ _ _ = False
45 compareBigger (Just grid) x y value 0 = value > getXY (Just grid) (x-1) y || getXY (Just grid) (x-1) y == 0
46 compareBigger (Just grid) x y value 1 = value > getXY (Just grid) x (y+1) || getXY (Just grid) x (y+1) == 0
47 compareBigger (Just grid) x y value 2 = value > getXY (Just grid) (x+1) y || getXY (Just grid) (x+1) y == 0
48 compareBigger (Just grid) x y value 3 = value > getXY (Just grid) x (y-1) || getXY (Just grid) x (y-1) == 0
49 compareBigger (Just grid) x y value n
50     | n < 0 || n > 3 = error "compareBigger: n is not in range 0..3"
51     | otherwise = compareBigger (Just grid) x y value n
52
53 -- Função responsável pelas comparações de MENOR QUE.
54 compareSmaller :: Maybe [[Int]] -> Int -> Int -> Int -> Int -> Bool
55 compareSmaller Nothing _ _ _ _ = False
56 compareSmaller (Just grid) x y value 0 = value < getXY (Just grid) (x-1) y || getXY (Just grid) (x-1) y == 0
57 compareSmaller (Just grid) x y value 1 = value < getXY (Just grid) x (y+1) || getXY (Just grid) x (y+1) == 0
58 compareSmaller (Just grid) x y value 2 = value < getXY (Just grid) (x+1) y || getXY (Just grid) (x+1) y == 0
59 compareSmaller (Just grid) x y value 3 = value < getXY (Just grid) x (y-1) || getXY (Just grid) x (y-1) == 0
60 compareSmaller (Just grid) x y value n
61     | n < 0 || n > 3 = error "compareSmaller: n is not in range 0..3"
62     | otherwise = compareSmaller (Just grid) x y value n
63
64 -- Função responsável por executar as comparações.
65 executeComparison :: Maybe [[Int]] -> Char -> Int -> Int -> Int -> Int -> Bool
66 executeComparison vergleichssudokuGrid comparator x y value operatorType
67     | comparator == '.' = True
68     | comparator == '>' = compareBigger vergleichssudokuGrid x y value operatorType
69     | comparator == '<' = compareSmaller vergleichssudokuGrid x y value operatorType
70     | otherwise = error "executeComparison: Invalid comparator"
71
72 -- Pega as comparações de uma determinada posição.
73 getCompare :: Maybe [[Int]] -> [[Char]] -> Int -> Int -> [Int]
74 getCompare vergleichssudokuGrid comparatorsGrid x y = [a | a <- [1..sizeBoard], canFitComparators a]
75     where
76         comparators = getXY (Just comparatorsGrid) x y
77         canFitComparators a = all (==True) [executeComparison vergleichssudokuGrid (comparators !! index) x y a index | index <- [0..3]]
78

```

Figura 4 – Funções que tratam das comparações entre as células do tabuleiro.

```

78
79 -- Retorna uma lista com as opções possíveis para uma determinada posição.
80 getPossibleOptions :: Maybe [[Int]] -> [[[Char]]] -> Int -> Int -> [Int]
81 getPossibleOptions vergleichssudokuGrid vergleichssudokuGridChars x y = [a | a <- [1..sizeBoard], notInRow a, notInCol a, notInSquare a, inCo
82   where
83     notInRow a = a `notElem` getRow vergleichssudokuGrid x y
84     notInCol a = a `notElem` getCol vergleichssudokuGrid x y
85     notInSquare a = a `notElem` getRegion vergleichssudokuGrid x y sizeBoard
86     inCompareOptions a = a `elem` getCompare vergleichssudokuGrid vergleichssudokuGridChars x y
87
88 -- Retorna o valor de uma lista em um determinado índice.
89 getValueInList :: [a] -> Int -> a
90 getValueInList [] _ = error "getValueInList: index too large" -- Caso base: lista vazia.
91 -- Caso recursivo: retorna o valor da cabeça da lista caso o índice seja 0, caso contrário, chama a função novamente com a cauda da lista e o
92 getValueInList (x:xs) i
93   | i == 0 = x -- Caso base: índice 0.
94   | otherwise = getValueInList xs (i - 1) -- Caso recursivo: chama a função novamente com a cauda da lista e o índice decrementado.
95
96 -- Retorna o tamanho de uma lista
97 getListLength :: [a] -> Int
98 getListLength [] = 0 -- Caso base: lista vazia.
99 getListLength (_:xs) = 1 + getListLength xs -- Caso recursivo: soma 1 ao tamanho da cauda da lista.
100

```

Figura 5 – Algumas das funções auxiliares de /src/VergleichssudokuSolver.hs.

```

101 -- Função responsável pela solução do tabuleiro.
102 -- Recebe o tabuleiro, o tabuleiro de comparações, a linha atual e a coluna atual.
103 solveVergleichssudoku :: Maybe [[Int]] -> [[[Char]]] -> Int -> Int -> Maybe [[Int]]
104 solveVergleichssudoku vergleichssudokuGrid comparatorsGrid row column = do
105   -- Retorna o tabuleiro caso tenha chegado na última célula.
106   if row == (sizeBoard - 1) && column == sizeBoard then trace "Found the solution: " vergleichssudokuGrid
107   -- Verifica se chegou ao fim de uma linha, caso tenha chegado, salta para a próxima.
108   else if column == sizeBoard then solveVergleichssudoku vergleichssudokuGrid comparatorsGrid (row + 1) 0
109   -- Verifica se o valor da célula atual já foi definido, caso tenha sido, passa para a próxima célula.
110   else if getXY vergleichssudokuGrid row column > 0 then trace "Value already defined" solveVergleichssudoku vergleichssudokuGrid comparators
111   else do
112     -- Checa os valores possíveis para a célula atual.
113     possibles <- Just (getPossibleOptions vergleichssudokuGrid comparatorsGrid row column)
114     -- Após validar a posição e adquirir os possíveis números chama a função recursiva que testa cada um deles.
115     solveVergleichssudokuWithValues vergleichssudokuGrid comparatorsGrid row column possibles 0
116

```

Figura 6 – Funções que tratam das comparações entre as células do tabuleiro.

```

116
117 -- A partir de uma lista de possíveis números para uma posição específica testa cada um deles até terminar a lista ou algum funcionar
118 solveVergleichssudokuWithValues :: Maybe [[Int]] -> [[[Char]]] -> Int -> Int -> [Int] -> Int -> Maybe [[Int]]
119 -- Caso a lista de possibilidades esteja vazia, não há solução para o tabuleiro de entrada.
120 solveVergleichssudokuWithValues vergleichssudokuGrid comparatorsGrid row column possibles index = do
121   -- Verifica se o index ultrapassou o tamanho da lista, nesse caso não há solução possível para o tabuleiro de entrada.
122   if index >= getListLength possibles then Nothing
123   else do
124     -- "Seta" a grid com o valor encontrado na lista de possibilidades no index recebido.
125     vergleichssudokuGrid <- setXY vergleichssudokuGrid row column (getValueInList possibles index)
126     -- Continua o "flow" para a célula seguinte e verifica o retorno.
127     case solveVergleichssudoku (Just vergleichssudokuGrid) comparatorsGrid row (column + 1) of
128       -- Caso seja Nothing, não houve uma solução.
129       Nothing -> do
130         -- Reseta o valor da célula atual.
131         setXY (Just vergleichssudokuGrid) row column 0
132         -- Chamada recursiva para testar o próximo valor da lista de possibilidades.
133         solveVergleichssudokuWithValues (Just vergleichssudokuGrid) comparatorsGrid row column possibles (index + 1)
134       -- Retorna a solução.
135       Just n -> Just n
136

```

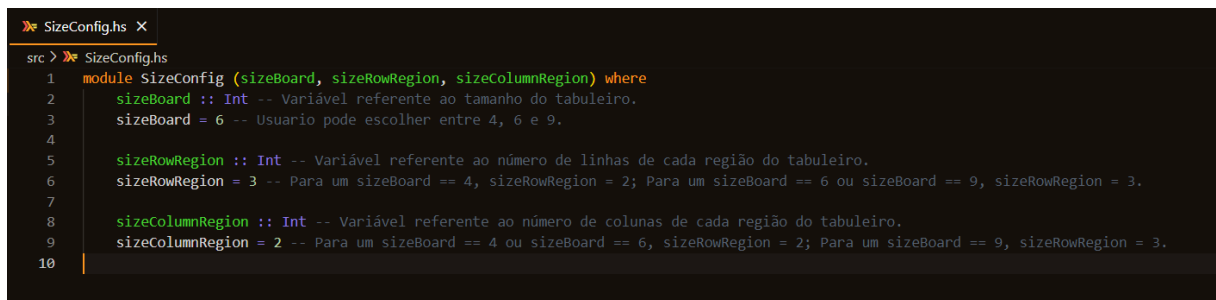
Figura 7 – Funções que tratam das comparações entre as células do tabuleiro.

3 Input e Output

3.1 Input

A princípio, tínhamos a intenção de permitir que o usuário "inputasse" os comparadores do tabuleiro diretamente pelo terminal, mas por dificuldades impostas pela linguagem *Haskell*, que serão comentadas posteriormente, optamos pela inserção dessas relações diretamente no código fonte.

Para inserir um *input*, primeiramente, o usuário precisa alterar as linhas 3, 6 e 9 do arquivo **SizeConfig.hs**, que está na pasta **/src/**. Os comentários nas linhas, que podem ser vistos na Figura 8, são o suficiente para que o usuário não cometa nenhum engano.

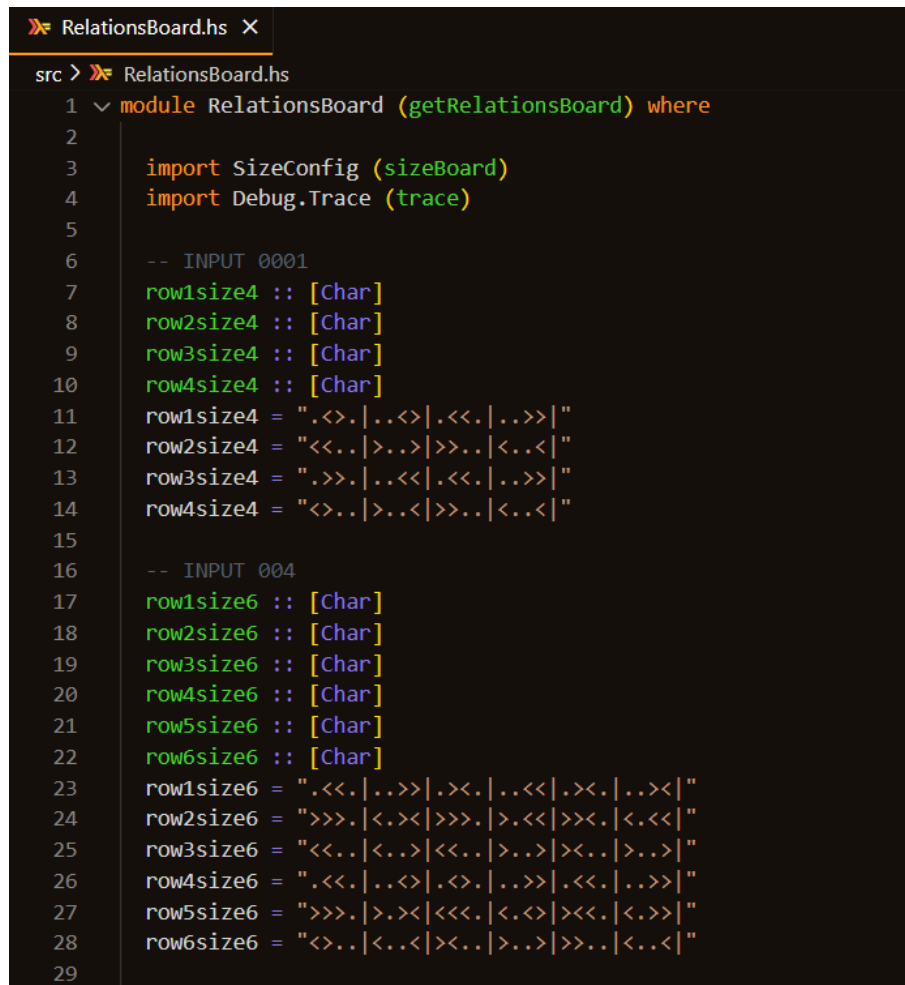


```
src > SizeConfig.hs
1 module SizeConfig (sizeBoard, sizeRowRegion, sizeColumnRegion) where
2   sizeBoard :: Int -- Variável referente ao tamanho do tabuleiro.
3   sizeBoard = 6 -- Usuario pode escolher entre 4, 6 e 9.
4
5   sizeRowRegion :: Int -- Variável referente ao número de linhas de cada região do tabuleiro.
6   sizeRowRegion = 3 -- Para um sizeBoard == 4, sizeRowRegion = 2; Para um sizeBoard == 6 ou sizeBoard == 9, sizeRowRegion = 3.
7
8   sizeColumnRegion :: Int -- Variável referente ao número de colunas de cada região do tabuleiro.
9   sizeColumnRegion = 2 -- Para um sizeBoard == 4 ou sizeBoard == 6, sizeRowRegion = 2; Para um sizeBoard == 9, sizeRowRegion = 3.
10
```

Figura 8 – Arquivo **/src/SizeConfig.hs**.

Após definir as dimensões do tabuleiro, basta ao usuário definir as relações de comparação de cada uma das células do tabuleiro. Como isso pode ser um trabalho chato, deixamos alguns exemplos já formatados para o usuário na pasta **/inputs/**. Essa pasta está representada neste relatório na Figura 12. As relações devem ser redefinidas no arquivo **RelationsBoard.hs** (que pode ter seu trecho inicial visto na Figura 9), que está na pasta **/src/**. Caso o usuário tenha definido um tabuleiro 4x4 em **SizeConfig.hs**, o usuário deve alterar as linhas de 11 a 14. Para um tabuleiro 6x6, as linhas de 23 a 28, e para um tableiro 9x9, as linhas de 40 a 48.

Em relação à formatação dos *inputs*: cada linha representa uma linha do tabuleiro, contendo as 4 relações de cada célula daquela linha, separando as células por uma `'|'`. A ordem das relações para cada célula segue o sentido horário: **ACIMA, À DIREITA, ABAIXO, À ESQUERDA**. A Figura 10 demonstra como o exemplo da Figura 1 fica após a formatação do tabuleiro para o *input*.



```

src > RelationsBoard.hs
1  module RelationsBoard (getRelationsBoard) where
2
3  import SizeConfig (sizeBoard)
4  import Debug.Trace (trace)
5
6  -- INPUT 0001
7  row1size4 :: [Char]
8  row2size4 :: [Char]
9  row3size4 :: [Char]
10 row4size4 :: [Char]
11 row1size4 = ".<>|..<>|.<<|.>>|"
12 row2size4 = "<<..|>..|>>..|<..<|"
13 row3size4 = "..>..|..<<|.<<|.>>|"
14 row4size4 = "<>..|>..<|>>..|<..<|"
15
16 -- INPUT 004
17 row1size6 :: [Char]
18 row2size6 :: [Char]
19 row3size6 :: [Char]
20 row4size6 :: [Char]
21 row5size6 :: [Char]
22 row6size6 :: [Char]
23 row1size6 = ".<<|.>>|.><|.<<|.><|.><|"
24 row2size6 = ">>>|.<.<|>>>|.>.<<|>><|.<.<<|"
25 row3size6 = "<<..|<..>|<<..|>..>|><..|>..>|"
26 row4size6 = ".<<|.><>|.<.>|.>>|.<<|.>>|"
27 row5size6 = ">>>|.>.<|<<<|.<.<|><<|.<.>>|"
28 row6size6 = "<>..|<..<|><..|>..>|>>..|<..<|"
29

```

Figura 9 – Trecho inicial do arquivo /src/Relations.hs.

3.2 Output

A Figura 11 demonstra a saída de nosso programa para o *input* da Figura 10. Como pode ser visto, ele é compatível com a Figura 2. Por simplificação de código, nossa saída contém apenas os números ordenados de acordo com sua posição no tabuleiro, sem a presença de divisórias verticais ou horizontais entre as células ou regiões do tabuleiro. Apesar de simples, o formato do *output* não atrapalha ou dificulta a sua visualização e interpretação.

```
input011.txt X
inputs > 9x9 > input011.txt
1  .<>.|.><>|..<<|.<>|.<>>|..>>|.><|.>><|..<<|
2  <<<.|><<>|>.<>|<<<.|<>>>|<.><|>>>|.<<><|>.>>|
3  ><..|><.>|>..>|><..|<>.>|<..<|<>..|<<.<|<..>|
4  .>>.|.><<|..<<|.>>|.<><|..>>|.<<|.>>>|..<<|
5  <>>.|><<<|>.<>|<><|.<<<<|<.>>|>>>|.<>><|>.><|
6  <<..|>>.>|>..<|><..|>>.>|<..<|<>..|<<.<|<..>|
7  .<>.|.>>>|..><|.><|.>><|..><|.<<|.><|..<<|
8  <><|.<<<<|<.>>|>>>|.<<><|<.<>|><|.><<|>.>>|
9  ><..|>>.>|<..<|<<..|<<.>|>..>|<<..|>>.>|<..<|
```

Figura 10 – Exemplo da Figura 2 formatado.

```
WELLCOME TO VERGLEICHSSUDOKU SOLVER!
Found the solution:
3 4 2 6 8 9 7 5 1
1 5 6 3 7 2 8 4 9
7 8 9 4 5 1 3 2 6
9 2 1 5 4 7 6 8 3
8 3 4 2 1 6 9 7 5
6 7 5 8 9 3 2 1 4
4 9 8 7 6 5 1 3 2
2 1 7 9 3 4 5 6 8
5 6 3 1 2 8 4 9 7
```

Figura 11 – Output gerado por nosso programa para o *input* da figura 10.

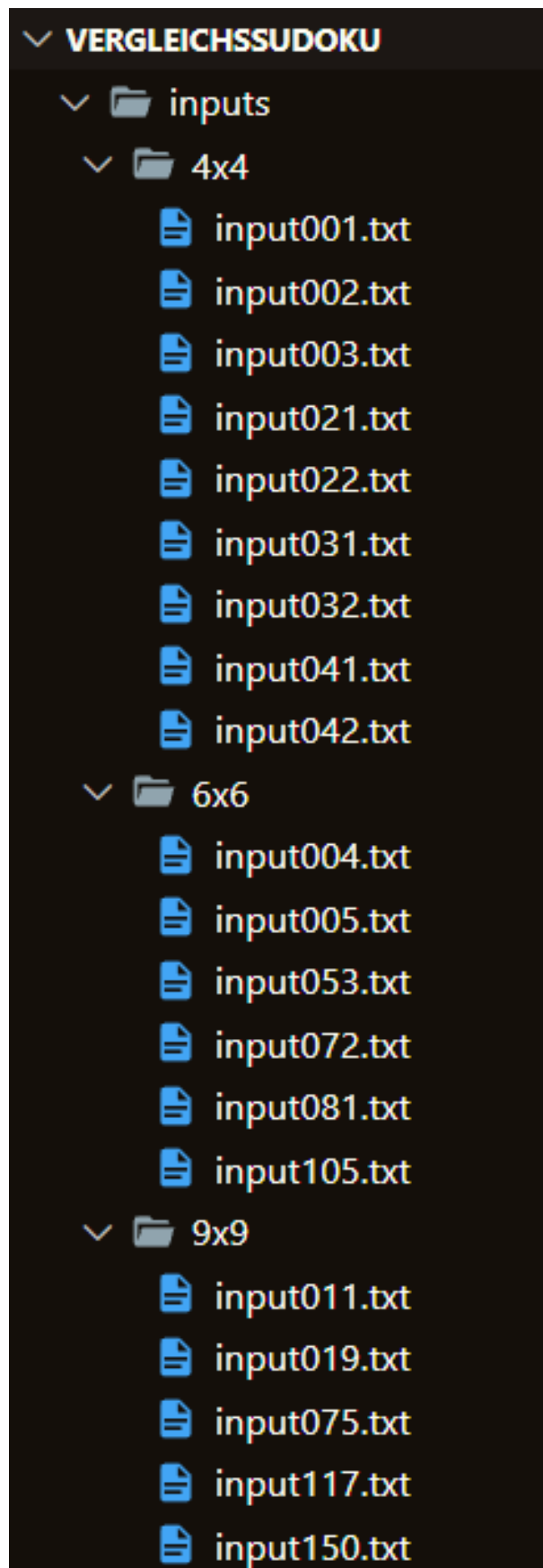


Figura 12 – Pasta de *inputs* de nosso repositório.

4 Organização do Grupo

4.1 Comunicação do Grupo

Para organização do trabalho, o grupo utilizou de algumas ferramentas de comunicação como:

- canal de voz do **Discord**;
- grupo de **Whatsapp**;
- aba de mensagens do **Moodle**.

4.2 Gerenciamento do Código

Para o gerenciamento do código do trabalho, utilizamos apenas o **GitHub**. Mantendo um repositório privado com apenas os 3 membros como colaboradores.

4.3 Ambiente de Desenvolvimento

A IDE (Integrated Development Environment) escolhida para o desenvolvimento do trabalho foi o **VSCode**, com o auxílio das extensões **Haskell**, **Haskell Syntax Highlighting** e **haskell-linter**.

5 Dificuldades Encontradas

No desenvolvimento do projeto, algumas dificuldades foram encontradas, especialmente por nenhum de nós ter experiência com *Haskell*. Abaixo estão listadas algumas dessas negativas.

- **Tipagem de variáveis do *Haskell*:** a tipagem de variáveis do *Haskell* foi um dos principais empecilhos em nosso trabalho. Como citado na Seção 3.1, a princípio tínhamos a ideia de aceitar inputs do usuário diretamente do terminal, mas por detalhes da linguagem como a diferença dos tipos `"Int"` e `"IO Int"`, desistimos dessa ideia;
- **Ausência de estruturas de repetição:** o *Haskell* não contém estruturas de repetição, como *while* e *for*, o que dificultou bastante a adaptação do código em *Python* desenvolvido, que continha bastante heurística e iterações sobre cada uma das regiões do tabuleiro. Esse código inicialmente desenvolvido em *Python* foi quase inteiramente descartado pela dificuldade de sua reprodução em *Haskell*;
- **Testes:** testar o código inicialmente era uma tarefa chata, tendo diversos problemas de compilação e, posteriormente, problemas com soluções erradas e/ou falta de solução para alguns dos tabuleiros (erros esses muitas vezes oriundos de um único comparador digitado incorretamente no arquivo de input).

Alguns outros problemas foram encontrados ao longo do projeto, mas foram menos marcantes/frustrantes, e não merecem a listagem aqui. No geral, a linguagem *Haskell* trouxe mais dificuldades ao desenvolvimento do projeto do que o puzzle em si.

Para nosso primeiro contato com uma linguagem puramente funcional, foi uma experiência interessante e enriquecedora. As dificuldades foram contornadas, e com certeza aprendemos bastante sobre linguagens funcionais nesse projeto.