UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE CIÊNCIAS DA COMPUTAÇÃO

ALEX DAVIS NEUWIEM DA SILVA
LUAN DINIZ MORAES
LUCAS CASTRO TRUPPEL MACHADO

Relatório Trabalho 2 Paradigmas de Programação

Florianópolis, 2023.

1. Introdução

O puzzle escolhido para ser resolvido é o Kojun. O código foi feito na linguagem Elixir utilizando backtracking como método de resolução.

Esse puzzle foi o que escolhemos para resolver com Haskell no trabalho 1, portanto já sabíamos a lógica que deveria ser implementada e a dificuldade se deu na tradução do código de Haskell para Elixir.

2. Análise do Problema

No Kojun, o jogador recebe inicialmente um tabuleiro com alguns números preenchidos e deve completar os espaços vazios com números de acordo com algumas regras. Além disso, o tabuleiro é dividido em algumas regiões, que são relevantes para resolução.

Existem 3 regras para preencher os espaços em branco:

1. Uma região precisa ter todos os números de 1 a N sem repetição, sendo N a quantidade de células na região.
2. Números em células ortogonalmente adjacentes devem ser diferentes.
3. Se duas células estiverem adjacentes verticalmente na mesma região, o número na célula superior deve ser maior que o número na célula inferior.

3. Entrada e Saída

As matrizes de entrada devem ser especificadas na parte inicial do código. Uma matriz representando os números iniciais das células e outra as regiões do tabuleiro. Já a saída é impressa no terminal, sendo a matriz resolvida ou a string “Nao ha solucao” caso não haja solução.

Obs. : O arquivo “docs/exemploEntrada.txt” possui 3 exemplos de entrada que podem ser copiados em “elixir/elixir.exs”

4. Descrição da Solução

Foi utilizada a técnica backtracking. Existem um módulo principal, Kj, definindo as funções utilizadas na solução e algumas variáveis declaradas fora desse módulo para serem passadas como argumento.

A função principal da solução é a *kojun*, que é chamada ao menos uma vez para cada posição da matriz. O método analisa se a posição é válida e, se ela estiver vazia, chama a função *avaliarNumeros*, que vai buscar um número para colocar na posição analisada.

A função *avaliarNumeros* recebe uma posição “i j” e um número “num” para preenchê-la. Ela utiliza a função *numeroEhpossivel*, que verifica se o número atende às 3 regras do jogo. Caso seja um número válido, atualiza a matriz com o método *atualizarMatriz* e chama *kojun* para a próxima posição. Caso seja um número inválido, chama a função *avaliarNumeros* novamente com o próximo número.

O algoritmo começa testando o maior número possível da região e, em caso erro, subtrai 1 e repete o teste. Essa estratégia foi escolhida baseando-se na regra 3, como os testes começam pela parte de cima da matriz, a probabilidade de acertar um número é maior começando do maior número.

```elixir
def kojun(i, j, numerosMatriz, regioesMatriz, regioes, tamanhoMatriz) do
  cond do
    # Percorreu a matriz inteira sem achar erros
    (i == tamanhoMatriz - 1) and (j == tamanhoMatriz) ->
      {True, numerosMatriz}

    # Terminou a linha
    (j == tamanhoMatriz) ->
      kojun((i+1), 0, numerosMatriz, regioesMatriz, regioes, tamanhoMatriz)

    # Posição já está ocupada
    (Enum.at(Enum.at(numerosMatriz, i), j) > 0) ->
      kojun(i, (j+1), numerosMatriz, regioesMatriz, regioes, tamanhoMatriz)

    # Posição válida e vazia, procura um número para ocupá-la
    true ->
      maxNum = tamanhoRegiao(regioes, Enum.at(Enum.at(regioesMatriz, i), j))
      # Começa avaliando a partir do maior número possível da região (segundo parâmetro)
      avaliarNumeros(maxNum, i, j, numerosMatriz, regioesMatriz, regioes, tamanhoMatriz)
  end
end
```

```elixir
def avaliarNumeros(num, i, j, numerosMatriz, regioesMatriz, regioes, tamanhoMatriz) do
  # Tentou todos os números e não encontrou nenhum válido
  if (num <= 0) do
    {false, numerosMatriz}
  else
    # Número é válido para posição i j
    if numeroEhpossivel(num, i, j, numerosMatriz, regioesMatriz, regioes, tamanhoMatriz) do
      matrizAtualizada = atualizarMatriz(num, i, j, numerosMatriz)
      # Tenta preencher a próxima posição da matriz, chamando a função "kojun"
      {resultado, matriz} = kojun(i, (j+1), matrizAtualizada, regioesMatriz, regioes, tamanhoMatriz)

      # Se o resultado do teste da próxima posição é válido, a posição atual também é válida.
      if (resultado) do
        {resultado, matriz}
      #T este da próxima posição retornou inválido, tentar outro número para posição atual.
      else
        avaliarNumeros((num - 1), i, j, numerosMatriz, regioesMatriz, regioes, tamanhoMatriz)
      end
    # úmero não é válido, tentará o próximo número
    else
      avaliarNumeros((num - 1), i, j, numerosMatriz, regioesMatriz, regioes, tamanhoMatriz)
    end

  end
end
```

Caso nenhum número seja válido, pelo menos um número que não faz parte da solução correta foi colocado em alguma posição da matriz de números, então a função retornará false. Como o algoritmo é recursivo, ele voltará para a posição anterior, que tentará outro número e, se não conseguir também, retornará false, voltando para posição anterior e assim o processo se repete até que sejam encontrados números válidos para todas casas. Desse modo, utilizando o backtracking, o algoritmo consegue eliminar muitas possibilidades inválidas, sendo muito mais eficiente do que uma solução de força bruta pura.

Além disso, o código usa uma estrutura para facilitar a verificação das células pertencentes a uma região. Essa estrutura é uma lista de listas em que cada lista interna representa uma região. O index de cada lista interna é o identificador da região, determinado na matriz de regiões. Cada lista de região é composta por tuplas {i, j} que representam as posições que pertencem a aquela região.

Por exemplo:

```
matrizRegioes = [[0 ,0 ,1 ,1 ,1 ,2],
                 [3 ,3 ,3 ,3 ,3 ,2],
                 [4 ,5 ,5 ,5 ,3 ,6],
                 [4 ,4 ,4 ,5 ,6 ,6],
                 [7 ,7 ,8 ,9 ,9 ,9],
                 [10,10,8 ,8 ,9 ,9]]
```

Resultaria na lista de regiões:

```
[
  [{0, 0}, {0, 1}],
  [{0, 2}, {0, 3}, {0, 4}],
  [{0, 5}, {1, 5}],
  [{1, 0}, {1, 1}, {1, 2}, {1, 3}, {1, 4}, {2, 4}],
  [{2, 0}, {3, 0}, {3, 1}, {3, 2}],
  [{2, 1}, {2, 2}, {2, 3}, {3, 3}],
  [{2, 5}, {3, 4}, {3, 5}],
  [{4, 0}, {4, 1}],
  [{4, 2}, {5, 2}, {5, 3}],
  [{4, 3}, {4, 4}, {4, 5}, {5, 4}, {5, 5}],
  [{5, 0}, {5, 1}]
]
```

Essa estrutura é criada dentro da função *definirRegioes*:

```elixir
def definirRegioes(regioesMatriz, qtdadeRegioes, tamanhoMatriz) do
  regioes = List.duplicate([], qtdadeRegioes)
  atualizarRegioes(regioesMatriz, regioes, tamanhoMatriz)
end

# Cria uma lista com todas coordenadas (i, j) e, com a função Enum.reduce do Elixir, pega cada posição (i, j)
#e aplica a função "atualizarRegiao", usando a lista de regioes como valor inicial.

def atualizarRegioes(regioes_matriz, regioes, tamanho_matriz) do
  coordenadas = for i <- 0..(tamanho_matriz-1), j <- 0..(tamanho_matriz-1), do: {i, j}
  Enum.reduce(coordenadas, regioes, &atualizarRegiao(regioes_matriz, &1, &2))
end

# Adiciona uma posição (i, j) na lista de sua região. Depois adiciona a lista da região atualizada na lista de regioes.
def atualizarRegiao(regioes_matriz, {i,j}, regioes) do
  idRegiao = Enum.at(Enum.at(regioes_matriz,i),j)
  regiao_atualizada = Enum.at(regioes, idRegiao) ++ [{i, j}]
  List.replace_at(regioes, idRegiao, regiao_atualizada)
end
```

Um exemplo de uso da estrutura é na função *numeroEhpossivel*, que avalia se o número “num” dado de parâmetro é válido na posição “i j”. A função acessa a lista de região “regiao” dentro da lista “regioes” a partir do identificador de região presente na posição “i j” da matriz “numerosMatriz”. Após isso, o método *verificarRegiao* analisa cada tupla {i, j} presente na lista “regiao” para verificar se existe algum número igual a “num” na região. Caso exista, “num” não é um número válido, pois viola a regra 1.

5. Haskell vs Elixir

Durante a implementação percebeu-se algumas diferenças entre essas linguagens. A primeira delas é que ,diferentemente do Haskell, Elixir não é uma linguagem puramente funcional, o que o torna um pouco mais flexível. Além disso, por trás dos panos, ele executa de forma concorrente e isso impacta diretamente no tempo de execução:

- Haskell: 5.667s
- Elixir: 1.098s

Importante ressaltar que esses tempos foram medidos utilizando o comando “time”, no terminal. Ambos os programas foram executados no mesmo ambiente (uma Virtualbox rodando Linux Mint) e resolvendo o exemplo 3 do nosso “exemplosEntrada.txt”. (Tabuleiro 17x17). (Observação: Esse resultado depende muito do ambiente em que é executado, pois também foram feitos alguns testes em outras máquinas, sem virtualização, e o Haskell foi mais rápido. Uma análise mais minuciosa é necessária para tirar conclusões concretas nesse aspecto.)

Sentimos que o Elixir possui uma variedade maior de funções prontas e muitas mais alto nível que as do Haskell. Existe uma facilidade maior para declarar variáveis (embora jamais sejam globais), tanto que na nossa solução final, enfatizamos isso declarando as matrizes fora de qualquer função e passando ao *Kj.kojun()* como argumento.

Todas as funções que você criar em Elixir precisam estar dentro de um módulo, porém as definições têm certa semelhança ao Python, devido a sua simplicidade. Não sendo necessário declarar os tipos dos argumentos, por exemplo.

Ademais, as diferenças que percebemos foram em relação à sintaxe. Elixir tem mais de uma opção para atribuir com a cabeça e a cauda de listas ([head | tail] ou hd() e tl()). Para acessar um índice específico de uma lista é necessário usar Enum.at(matriz, indice), o que é mais prático no haskell: matriz !! indice.

No começo, estranhamos algumas notações, como por exemplo a função Enum.any?(arg1, arg2), que tem uma interrogação no nome e também a forma alternativa de escrever funções anônimas:

- fn(a,b) -> a + b end

ou

- &(&1 + &2)

significam exatamente a mesma coisa para o Elixir.

Gostaríamos também de dar um destaque ao Pipe Operator (|>) que pode ser muito útil para deixar o código mais legível:

- funcao3(funcao2(funcao1(arg)))

ou

- arg |> funcao1() |> funcao2 |> funcao3

têm mesmo significado para o Elixir.

Em resumo, nesse problema em específico, o Elixir foi mais eficiente, além dele também ter sido mais flexível que o Haskell.

6. Organização do Grupo

O trabalho foi feito utilizando principalmente a extensão Live Share do VSCODE e utilizando GitHub. Todos os membros participaram da tradução do código, não havendo destaques específicos.

7. Dificuldades Encontradas

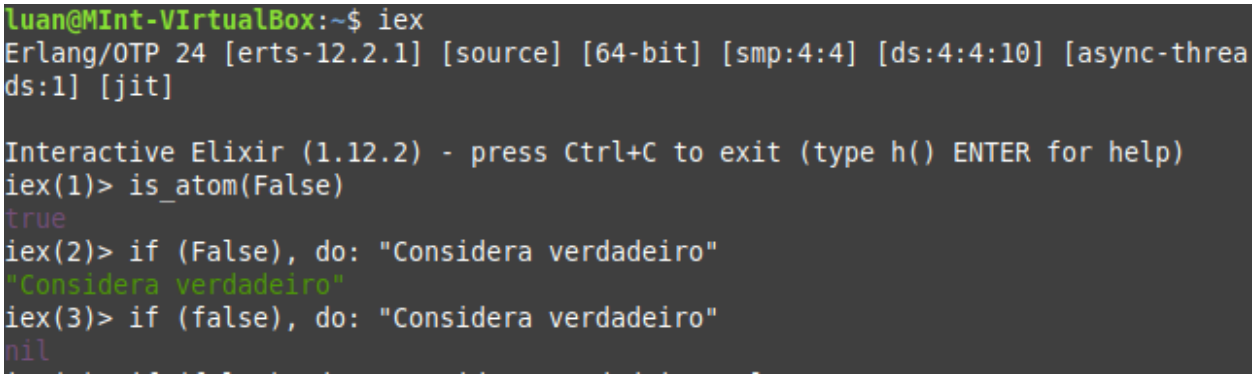
Sem dúvidas as maiores dificuldades estavam relacionadas a sintaxe. Para acessar um índice específico de uma lista em elixir é necessário usar Enum.at(matriz, índice), o que causou confusões algumas vezes já que estávamos trabalhando com matrizes e sem usar alguma biblioteca específica para isso. Outro exemplo é a tupla, que se representa como {i, j} e não como o habitual (i, j).

Embora não fosse difícil achar funções nativas do Elixir análogas ou até mais abrangentes que as do Haskell, o processo de testar elas consumiu bastante tempo, já que por vezes apareciam erros devido a ordem dos parâmetros serem diferentes nas duas linguagens, por exemplo.

Por fim, relataremos o último erro encontrado no trabalho antes dele funcionar como deveria. Após a tradução do código para Elixir e a correção de todos os erros de sintaxe e argumentos que apareceram, o resultado do programa não estava correto. Nossas suspeitas, corretas, estavam sobre a função *avaliarNumeros*. Logo percebeu-se uma nuance que não tínhamos considerado: a função em algum momento retornava {False, numerosMatriz} ao invés de {false, numerosMatriz}.

O Elixir interpreta quase tudo como true, com exceção das palavras chave false (átomo :false) e nil (átomo :nil). Ao escrever False, com a letra f maiúscula, o elixir o interpretou como um átomo (:False) e, como :False != :false, ele é considerado como

verdadeiro.



Essa interpretação como átomos faz com que erros de sintaxe ou compilação não sejam gerados e podem resultar em pequenas complicações como essa.