Universidade Federal de Santa Catarina

Campus de Florianopolis

Relatório Vergleichssudoku - resolução em LISP

João Volpato

Thiago Bewiahn

Rafael Vargas

Florianópolis/SC

2022

Relatório Vergleichssudoku Prof. Maicon Zatelli

Resumo

O problema escolhido para ser resolvido neste trabalho foi o Vergleichssudoku. Ele é semelhante ao sudoku tradicional, com as mesmas restrições de não repetir os números em linhas, colunas e quadrados. Pórem a entrada do problema escolhido não contém números mas sim comparadores entre as células do tabuleiro. Com base nessas comparações (maior que e menor que), os números devem ser preenchidos no tabuleiro.

Sumário

1 Solução

1.1 Introdução

Para visualizar a solução, é necessário executar o arquivo main.lisp, ou compilá-lo pelo comando "clisp -c main.lisp"e executar o comando "clisp main.fas". A segunda alternativa terá melhor performance. O programa imprimirá "NIL"no caso de não haver solução ou, caso contrário, as linhas da matriz-resposta.

1.2 Estruturas

Como o problema possui dois elementos principais, o proprio tabuleiro do sudoku e a entrada de comparadores, decidimos criar duas estruturas separadas, uma para representar o tabuleiro e outra a entrada dos comparadores.

Para o tabuleiro, a estrutura utilizada foi uma lista de listas de inteiros (matriz de inteiros). Devido à tipagem dinâmica do LISP, não foi necessário utilizar uma estrutura mais complexa (como o Monad Maybe em Haskell utilizado no Trabalho I), visto que para indicar que uma solução era inválida podíamos simplesmente retornar NIL.

A entrada dos comparadores foi organizada da mesma foram: uma matriz de strings. Cada posição corresponde aos comparadores da célula correspondente na matriz de inteiros. A ordem dos comparadores em cada célula segue o sentido horário. Ex.: .<>. O primeiro caractere representa o comparador com o topo (O ponto representa a ausência de comparador), o segundo representa a comparação com a direita, terceiro o de baixo e quarto o da esquerda. Neste caso: (TOPO ., DIREITA <, BAIXO >, ESQUERDA .). As matrizes de comparadores estão hardcoded, representadas linha a linha como uma String e o | representa a divisão de cada célula. Os | são eliminados ao construir a matriz e a String é quebrada em Strings menores (1 String por célula).

```
(defconstant row1size4 ".<>.|..<>|.<<.|..>>|")
(defconstant row2size4 "<<..|>..>|>>..|<..<|")
(defconstant row3size4 ".>>.|..<<|.<<.|..>>|")
(defconstant row4size4 "<>..|>..<|>>..|<..<|")
```

1.3 Comparação: Haskell x LISP

A principal vantagem do LISP foi a tipagem dinâmica. Conseguimos escrever código bem mais rápido sem precisar se preocupar em declarar a assinatura de cada método e resolver os erros de tipagem do Haskell, que eram constantes. Além disso, apesar de os erros levantados pelo compilador e interpretador do LISP serem bem mais difíceis de compreender, em LISP nós conseguimos com facilidade adicionar 'prints' ao código, o que tornou a depuração mais rápida. Em Haskell foi necessário encontrar uma função de uma biblioteca à parte (por algum motivo não estava havendo compatibilidade entre os Monads Maybe e IO), o que levou algum tempo e travou o desenvolvimento do trabalho.

Quanto às desvantagens, enxergamos duas: menor número de bibliotecas e funções nativas, e sintaxe. O menor número de bibliotecas motivou a criação de um novo arquivo "utils.lisp"para guardar funções mais genéricas criadas para a manipulação dos dados. Em relação à sintaxe, Haskell nos pareceu mais enxuto e expressivo. A única vantagem que enxergamos nesse quesito foi a presença de um loop 'for' nativo, o que facilitou na escrita de algumas funções. Consideramos como negativo no LISP:

- Falta de list comprehension;
- Excesso de parênteses (às vezes dificultava a leitura do código);
- Notação préfixada (todas as outras linguagens com as quais tivemos contato utiliza notação infixa);
- Falta da cláusula 'where' (permitia organizar melhor o código);

Exemplificando o que foi dito sobre sintaxe, segue uma comparação da implementação da função getPossibleOptions para ambas as linguagens:

Em LISP:

```
(defun isPossible (grid comparatorsGrid row column value)
  (if (not (null (find value (getX grid row))))
    NIL
    (if (not (null (find value (mapa (lambda (l) (getX l column)) grid))))
      NIL
      (if (not (null (find value (getSquare grid row column))))
      NIL
      (if (null (find value (getCompare grid comparatorsGrid row column)))
        NIL
        T
      )
      )
```

```
    )
  )
)
```

```
(defun getPossibleOptions (grid comparatorsGrid row column)
  (progn
  (setq possiblesList '())
  (loop for i from 1 to sudokuSize do
    (if (isPossible grid comparatorsGrid row column i)
      (setq possiblesList (cons i possiblesList))
    )
  )
  possiblesList
  )
)
```

Em Haskell:

```
getPossibleOptions :: Maybe [[Int]] -> [[[Char]]] -> Int -> Int -> [Int]
getPossibleOptions sudokuGrid sudokuGridChars x y = [a | a <- [1..sudokuSize],
    notInRow a, notInCol a, notInSquare a, inCompareOptions a]
    where
        notInRow a = a `notElem` getRow sudokuGrid x y
        notInCol a = a `notElem` getCol sudokuGrid x y
        notInSquare a = a `notElem` getSquare sudokuGrid x y sudokuSize
        inCompareOptions a = a `elem` getCompare sudokuGrid sudokuGridChars x y
```

1.4 Algoritmos

O principal algoritmo que resolve o problema do sudoku são as duas funções solveSudoku e solveSudokuWithValues:

```
(defun solveSudoku (grid comparatorsGrid row column)
  (cond ((and (= row sudokuSize) (= column 0)) (printGrid grid))
        ((= column sudokuSize) (solveSudoku grid comparatorsGrid (+ row 1) 0))
        ((> (getXY grid row column) 0) (solveSudoku grid comparatorsGrid row
           (+ column 1)))
        (t (solveSudokuWithValues grid comparatorsGrid row column
           (getPossibleOptions grid comparatorsGrid row column) 0))
  )
```

```
)
```

A função solveSudoku recebe de entrada o Grid do sudoku, o Grid dos comparadores, a linha, e a coluna. O primeiro teste é o caso de fim do backtracking onde achamos uma solução valida, então a função printGrid é chamada. Ela imprime a matriz no formato descrito na seção de Introdução, e também retorna a matriz, para encerrar a recursão. Depois a função testa se a chegamos ao final da linha no tabuleiro e caso positivo passa para a próxima linha, caso contrário verificamos se o valor atual da célula já foi definido. Se já está definido, passamos para a próxima célula, se não está chamamos a função getPossibleOptions para calcular a lista com os possíveis valores que aquela célula pode assumir, e chamamos a função solveSudokuWithValues.

```
(defun solveSudokuWithValues (sudokuGrid comparatorsGrid row column possibles
    index)
(if (>= index (length possibles))
      '()
       (if (null (solveSudoku (setXY sudokuGrid row column (getX possibles
           index)) comparatorsGrid row (+ column 1)))
         (solveSudokuWithValues (setXY (setXY sudokuGrid row column (getX
            possibles index)) row column 0) comparatorsGrid row column
            possibles (+ index 1))
         (setXY sudokuGrid row column (getX possibles index))
       )
  )
)
```

A função solveSudokuWithValues está escrita de forma semelhante à função em Haskell devido a dificuldades com a sua implementação utilizando um loop 'for'. Porém ela age de forma semelhante: é chamada para cada elemento da lista de possíveis valores. Se em algum momento index >= tamanho da lista, sabemos que todas as possibilidades foram testadas e nenhuma satisfaz o sudoku. Logo, retornamos uma lista vazia (equivalente a NIL). Para aplicar o backtracking de fato, a função solveSudoku é chamada novamente, porém desta vez preenchendo a posição atual com um dos possíveis valores, e prosseguindo para a próxima posição. Se o retorno for falso, tentamos preencher com outro valor, chamando solveSudokuWithValues para o próximo valor da lista de possibilidades. Senão, conseguimos preencher o sudoku com um valor válido, e apenas retornamos o sudoku com o valor preenchido.

No lugar do list comprehension do Haskell, optamos por utilizar o loop 'for' nativo do LISP em funções como getSquare, que gera uma lista com os elementos do quadrado do sudoku:

```
(defun getSquare (grid row column)
  (progn
  (setq squareList '())
  (loop for i from (- row (mod row nSquare)) to (- (+ (- row (mod row
      nSquare)) nSquare) 1) do
    (loop for j from (- column (mod column nSquare)) to (- (+ (- column (mod
        column nSquare)) nSquare) 1) do
      (setq squareList (cons (getXY grid i j) squareList))
    )
  )
  squareList
  )
)
```

Uma lógica semelhante foi utilizada para gerar as listas na função getCompare, porém com sendo necessário apenas um loop.

Por fim, gostaríamos de destacar a função stringToList. Achamos curiosa a sua necessidade, visto que em LISP strings seriam listas de caracteres. Porém na prática percebemos que isso não era o caso, sendo necessária a conversão feita por esta função.

```
(defun stringToList (str)
  (stringToListRecursive str 0)
)

(defun stringToListRecursive (str i)
    (if (= (length str) i)
      ()
      (cons (char str i) (stringToListRecursive str (+ i 1)))
    )
)
```

1.5 Entrada e Saída

1.5.1 Entrada

Os arquivos dataset.lisp e config.lisp são responsáveis por guardar a entrada do programa (comparatorsGrid) e pela variavél global que guarda o tamanho do sudoku a ser resolvido, respectivamente.

Exemplo de entrada:

```
(defconstant row1size9 ".<>.|.><>|..<<|.<>.|.<>>|..>>|.><.|.>><|..<<|")
(defconstant row2size9 "<<<.|><<>|>.<>|<<<.|<>>>|<.><|>>>.|<<><|>.>>|")
(defconstant row3size9 "><..|><.>|>..>|><..|<>.>|<..<|<>..|<<.<|<..>|")
(defconstant row4size9 ".>>.|.><<|..<<|.>>.|.<><|..>>|.<<.|.>>>|..<<|")
(defconstant row5size9 "<>>.|><<<|>.<>|<><.|<<<<|<.>>|>>>.|<>><|>.><|")
(defconstant row6size9 "<<..|>>.>|>..<|><..|>>.>|<..<|<>..|<<.<|<..>|")
(defconstant row7size9 ".<>.|.>>>|..><|.><.|.>><|..><|.<<.|.><>|..<<|")
(defconstant row8size9 "<><.|<<<<|<.>>|>>>.|<<><|<.<>|><>.|><<>|>.>>|")
(defconstant row9size9 "><..|>>.>|<..<|<<..|<<.>|>..>|<<..|>>.>|<..<|")
```

1.5.2 Saída

A saída do programa (o tabuleiro do sudoku resolvido caso tenha achado solução ou NIL caso contrário) segue o seguinte padrão:

```
(0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0)
(0 0 0 0 0 0 0 0 0)
```

1.6 Organização

A comunicação do grupo foi feita com "reuniões"no tempo de aula e em um grupo de Whatsapp.

Nesses momentos e posteriormente fora de aula, a ideia foi traduzir as funções utilizadas para a solução em Haskell e aplicá-las para LISP.

1.7 Dificuldades

Desta vez não tivemos dificuldades em relação ao paradigma funcional devido à experiência prévia com Haskell. A principal dificuldade foi em relação à sintaxe, como citado anteriormente (notação pré-fixada, utilizar o loop 'for' corretamente).