Tabuleiro de Vergleichssudoku em Lisp

Maykon Marcos Junior - 22102199 e Tiago Faustino de Siqueira - 22102193

Novembro 2023

Contents

1 Introdução

1.1 Solução Adotada pelo Grupo

1.1.1 Modelagem do Tabuleiro

O código original em Haskell utiliza tipos de dados para representar células e tabuleiros. Ao transpor isso para Lisp, utilizamos estruturas de dados (`defstruct`) para garantir a integridade dos dados. A função `deep-copy-cell` assegura que modificações em uma célula não afetem outras.

O formato da quíntupla continua o mesmo da implementação em Haskell para Lisp, assim como observado abaixo:

" {num} {left} {up} {right} {down} "

Figure 1: Formatação das células

1.1.2 Algoritmos Utilizados

No código Haskell, a função *isValid* verifica a validade de um número em uma posição. A tradução para Lisp, mantendo a mesma lógica, foi crucial devido à sua importância na lógica do algoritmo de resolução do jogo Vergleichssudoku.

Função isValid

```
(defun isValid (board num row col)
  (and
    (isRowValid board num row)
    (isColValid board num col)
    (isBoxValid board num row col)
    (isComparativeValid board num row col)
  )
)
```

1.1.3 Entrada e Saída de Dados

A consistência na leitura do tabuleiro a partir do arquivo (`"../tabuleiro.txt"`) entre Haskell e Lisp é mantida. Da mesma forma, a apresentação dos resultados na saída padrão segue um padrão uniforme. Ao aderir a um padrão consistente na exibição dos resultados na saída padrão, facilitamos a comparação direta dos resultados obtidos pelas duas implementações. Isso simplificou a análise dos resultados.

Função principal

```
(defun main ()
  (let ((content (with-open-file
                  (stream "../tabuleiro.txt"
                          :direction :input)
                         (read-line stream))))
    (let ((board (readBoard content)))
      (let ((solution (solveComparative board)))
        (if solution
            (printBoard (first solution))
            (princ "No solution found")
        )
      )
    )
  )
)
```

2 Vantagens e Desvantagens entre Haskell e Lisp

2.1 Vantagens

2.1.1 Haskell

A implementação em Haskell se destaca pelo uso de tipos de dados e padrões claros, o que resulta em código altamente legível. A expressividade da linguagem facilita a compreensão da lógica do programa.

2.1.2 Lisp

Comparado a Haskell, é muito mais fácil depurar o código.

2.2 Desvantagens

2.2.1 Haskell

A curva de aprendizado íngreme do Haskell pode representar um desafio para alguns desenvolvedores. Além disso, a menor popularidade pode limitar o suporte da comunidade.

2.2.2 Lisp

Ao migrar do Haskell para Lisp, a sintaxe peculiar de Lisp pode ser inicialmente desafiadora. A falta de bibliotecas modernas em Lisp pode demandar mais esforço na implementação de certas funcionalidades em comparação com Haskell.

3 Mudanças na Implementação/ Estratégia

Adaptar a lógica funcional de Haskell para a flexibilidade de Lisp exigiu ajustes específicos. A função *tryNumber* e *solveComparative* tiveram modificações para se adequar à sintaxe de Lisp.

Função solveComparative

```
(defun solveComparative (board)
  (let ((emptyCell (findEmpty board 0 0)))
    (if (null emptyCell)
        (list board)
        (let ((row (car emptyCell))
              (col (cdr emptyCell)))
          (tryNumber board 1 row col)
        )
    )
  )
)
```

4 Organização do Grupo

O trabalho em grupo envolveu comunicação efetiva, compartilhamento de ideias e resolução colaborativa de desafios. Cada membro contribuiu para a tradução da lógica Haskell para Lisp.

5 Dificuldades Encontradas e Soluções Adotadas

5.1 Diferença na Manipulação de Dados e Estruturas

Durante a migração do código de Haskell para Lisp, uma das principais dificuldades enfrentadas pelo grupo foi lidar com a diferença na manipulação de tipos de dados e estruturas. O Haskell é estritamente tipado, enquanto o Lisp é dinamicamente tipado e utiliza listas de forma extensiva.

Relacionamento com o Código: A função *deep-copy-cell* e a manipulação de estruturas de dados no código Lisp refletem a necessidade de acomodar a tipagem dinâmica, um desafio que não estava presente na implementação original

em Haskell.

5.2 Tipagem Fraca em Lisp

Pelo fato do Lisp ser dinamicamente tipado, a transição da tipagem forte e estática de Haskell para a tipagem fraca e dinâmica de Lisp demandou tempo significativo. A equipe teve que lidar com a natureza dinâmica dos tipos em Lisp, resultando em uma adaptação na representação de dados.

Relacionamento com o Código: A função *`deep-copy-cell`* e a manipulação de estruturas de dados no código Lisp refletem a necessidade de acomodar a tipagem dinâmica, um desafio que não estava presente na implementação original em Haskell.

5.3 Contextualização e Sistema de Escopo em Lisp

A diferença no sistema de escopo entre Lisp e Haskell foi uma dificuldade adicional. A capacidade de passar funções como referências e a falta de distinção clara entre paradigmas funcional e imperativo exigiram uma reavaliação da lógica.

Relacionamento com o Código: A adaptação do código para incorporar características específicas de Lisp, como o sistema de escopo, é observada na modificação de funções como *`solveComparative`*.

5.4 Falta de Documentação em Lisp

A falta de documentação adequada para Lisp complicou a compreensão do código existente. A equipe teve que recorrer a recursos externos e realizar experimentações para entender completamente a implementação.

5.5 Adaptação de Tipos de Dados

A lógica original em Haskell faz uso de tipos de dados específicos, como tuplas e listas, de forma intensiva. A tradução direta desses tipos para Lisp não seria eficiente. A equipe teve que repensar a representação das células e do tabuleiro para melhor se adequar à natureza dinâmica de Lisp.

Relacionamento com o Código: No trecho original em Haskell, a tupla `type Cell = (Int, Char, Char, Char, Char)` foi diretamente transposta para a estrutura em Lisp:

Função solveComparative

```
(defstruct cell
(first 0)
(second #\space)
(third  #\space)
(fourth #\space)
(fifth  #\space)
)
```

5.6 Estratégias de Cópia Profunda

A implementação de cópias profundas em Lisp, necessárias para garantir a integridade dos dados durante as operações, apresentou desafios adicionais. A necessidade de criar novas instâncias de estruturas de dados levou a discussões sobre eficiência e clareza do código.

Relacionamento com o Código: A função *deep-copy-cell* e *deep-copy-board* foi introduzida para criar cópias profundas, garantindo que modificações em uma célula ou no tabuleiro original não afetassem as cópias.

Função

```
(defun deep-copy-cell (cell)
  (make-cell
    :first   (cell-first   cell)
    :second  (cell-second  cell)
    :third   (cell-third   cell)
    :fourth  (cell-fourth  cell)
    :fifth   (cell-fifth   cell)
  )
)
```

5.7 Manipulação de Listas e Recursão

A transição da manipulação funcional de listas em Haskell para Lisp exigiu uma abordagem adaptativa. A linguagem Lisp é naturalmente orientada a listas, mas a mudança na mentalidade para pensar de maneira recursiva e funcional foi desafiadora para alguns membros da equipe.

Relacionamento com o Código: Funções como `mapcar` e `lambda` foram utilizadas extensivamente para manipular listas em Lisp, o que demandou uma compreensão profunda do paradigma funcional.

Função deep-copy-board

```
(defun deep-copy-board (board)
  (make-board :cells
    (mapcar
      (lambda
        (row)
        (mapcar #'deep-copy-cell row)
      )(board-cells board)
    )))
```

5.8 Discussões sobre Eficiência

A equipe enfrentou dilemas ao decidir sobre a eficiência versus a clareza do código. Estratégias para otimização de cópias profundas e iterações sobre listas foram alvo de discussões intensas para encontrar um equilíbrio adequado.

Relacionamento com o Código: A decisão de utilizar `mapcar` em vez de loops tradicionais levou a debates sobre legibilidade versus desempenho. Em síntese, as dificuldades enfrentadas durante a adaptação do código Haskell para Lisp não estiveram apenas na sintaxe, mas também na redefinição de estruturas de dados e na mudança de paradigma na manipulação de listas e tipos dinâmicos. As soluções foram alcançadas por meio de discussões detalhadas, pesquisa e experimentação contínua.

6 Conclusão

A migração bem-sucedida do resolvedor de Vergleichssudoku de Haskell para Lisp destaca a versatilidade de ambas as linguagens. A comparação entre Haskell e Lisp destaca nuances significativas, proporcionando uma experiência rica em aprendizado para o grupo.