Relatório de Implementação do jogo Kakuro

Alek Frohlich & Nicolas Goeldner, 06 de Outubro de 2019,

Executando o programa:

Detalhes para como configurar os casos de teste e como executar a aplicação se encontram no README do repositório.

Evoluíndo uma implementação em C++ para Haskell:

Inicialmente, nos aproximamos do jogo implementando-o na linguagem C++. Esta implementação nos ajudou a encontrar os casos de falha de um árvore de computação no modelo de backtracking que construíamos. O código de referência, em C++, pode ser visto na figura abaixo.

```
/*
 * Solves a given kakuro board.
 */
int solve(int i, int j)
{
    for (int w = 1; w < 10; w++)
    {
        board[i][j].first = w;
        if (can_continue(i, j, w))
        {
            idx p = next_pos(((j == LAST_J)? i+1: i), (j+1) % WIDTH);

            if (has_finished(p.x, p.y) && (j == LAST_J ||
board[LAST_I][LAST_J].c == BLACK))
                return SOLVED;

            if (solve(p.x, p.y) == SOLVED)
                return SOLVED;
        }
    }

    return UNSOLVABLE;
}
```

Figura 1. Solução em C++.

Antes da execução do procedimento **solve(int,int)**, assumimos a existência de uma matriz de posições correspondente ao tabuleiro kakuro e uma estrutura **pos**, responsável por representar uma posição do tabuleiro. Uma posição pode ser preta ou branca e contém dois valores: first e second, como vistas abaixo na figura 2.

```
/*
 * Black/White position.
 */
struct pos {
    int c;
    int first;
    int second;
};
```

Figura 2. struct pos.

Na figura 1, podemos ver a iteração sobre todos os possíveis valores de uma posição branca (1-10), alterando o valor da presente posição para o mesmo. Após alterar o valor da posição branca, testamos para ver se podemos continuar ou se o valor recém colocado já coloca o tabuleiro em um estado inconsistente. Estado inconsistente, neste contexto, significa um estado em que as regras do Kakuro estão feridas. As regras são:

1. Não podem haver elementos repetidos entre duas casas pretas na mesma coluna ou linha.
2. Uma linha ou uma coluna completa deve ter a soma de seus elementos igual a dica contida na casa preta que a guia.
3. Casas pretas guiam as casas brancas abaixo e/ou a sua direita. Podemos ver na figura 3 abaixo que as casas brancas de índice (1,1) e (1,2), as quais contém os elementos 9 e 8, respectivamente, têm seus valores ditados pelas casas pretas de índices (0,1), (0,2) e (1,0), no sentido que 9 e 8 tem que somar 17, 9 e 3 tem que somar 12 e 8,2,1,3 e 7 tem que somar 21.

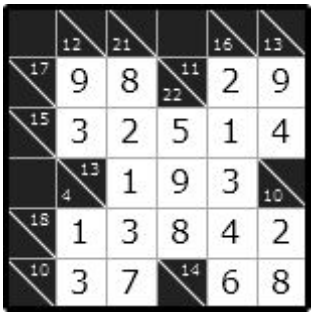

Figura 3. Um tabuleiro de Kakuro.

Estas regras são independentes de implementação e são garantidas nos procedimentos can_continue e canContinue de ambas as implementações.

```
static bool can_continue(int i, int j, int value)
{
    if (line_repeats(i, j, value) || col_repeats(i, j, value))
        return false;

    if ((((j == LAST_J) || ((j != LAST_J) && board[i][j+1].c == BLACK))
&& !check_line(i, j)))
        return false;
```

```
    if ((((i == LAST_I) || ((i != LAST_I) && board[i+1][j].c == BLACK))
&& !check_col(i, j)))
        return false;

    return true;
}
```

Figura 4. Teste de estado inconsistente em C++.

```
-- Check if branch of computation can procede.
canContinue (i,j) board w = do
    ifM ((lineRepeats (i,(j-1)) board w) ||^ (colRepeats ((i-1),j) board
w))
        (return False)
        (do
            ifM ((horizContTest (i,j) board) ||^ (vertContTest (i,j)
board))
                (return False)
                (return True))
```

Figura 5. Teste de estado inconsistente em Haskell.

Voltando o foco para a implementação em Haskell, vemos logo de primeira a presença de funções auxiliares como o ifM. Tais funções são detalhadas na última seção. Fora isso, vemos também a presença de quatro testes:

1. Elemento repete na linha? (lineRepeats)
2. Elemento repete na coluna? (colRepeats)
3. Soma dos elementos horizontais conformante com a dica da casa preta à esquerda? (horizContTest)
4. Soma dos elementos verticais conformante com a dica da casa preta acima? (vertContTest)

Na próxima figura, são mostradas suas implementações.

```
-- Check if line sums up to hint.
checkLine (i,j) board sum = do
    (color, value, final) <- board ! (i,j)
    if color == white
        then (checkLine (i, (j-1)) board (sum + value))
        else return (sum == final)

-- Check if collumn sums up to hint.
checkCol (i,j) board sum = do
    (color, value, _) <- board ! (i,j)
```

```
    if color == white
        then (checkCol ((i-1), j) board (sum + value))
        else return (sum == value)

-- Check if element repeats on given line.
lineRepeats (i,j) board w = do
    (color, value, _) <- board ! (i,j)
    if color == black
        then return False
        else if value == w
            then return True
            else (lineRepeats (i, (j-1)) board w)

-- Check if element repeats on given collumn.
colRepeats (i,j) board w = do
    (color, value, _) <- board ! (i,j)
    if color == black
        then return False
        else if value == w
            then return True
            else (colRepeats ((i-1), j) board w)

-- Test second if.
horizContTest (i,j) board = do
    checkL <- (checkLine (i,j) board 0)
    if (j == last_j) && (not checkL)
        then return True
        else do
            if (j /= last_j)
                then do
                    (color,_,_) <- board ! (nextPos (i,j))
                    if ((color == black) && (not checkL))
                        then
                            return True
                        else
                            return False
                    else
                        return False

-- Test third if.
vertContTest (i,j) board = do
    checkC <- (checkCol (i,j) board 0)
    if (i == last_i) && (not checkC)
        then return True
        else do
```

```
        if (i /= last_i)
            then do
                (color,_,_) <- board ! ((i+1),j)
                if ((color == black) && (not checkC))
                    then
                        return True
                    else
                        return False
                else
                    return False
```

Figura 6. Implementações dos testes em Haskell.

As funções checkLine, checkCol, lineRepeats e colRepeats são triviais, elas simplesmente recorrem pelas suas respectivas linhas/colunas, testando suas condições. Podemos ver da figura 4 que chamamos checkLine e checkCol nos índices anteriores a posição que estamos atualmente, fazemos isto para evitar testar se a casa que estamos é igual a ela mesma.

As funções horizContTest e vertContTest podem, à primeira vista, parecer complicadas, mas, na prática, só testam se estamos visitando a última casa branca da linha/coluna. Se estivermos, verifica se a soma dos elementos da linha/coluna está de acordo com a casa preta que as guia. Podemos ver nas mesmas funções que invocamos as funções checkLine e checkCol passando 0 como o terceiro argumento, sum. Fazemos isso para indicar que a soma dos elementos da linha/coluna começa em zero, sendo então acrescida aos valores pertencentes à sequência de posições visitadas pela recursão.

Após entendermos como a implementação de Haskell testa por estados inconsistentes, podemos retornar ao processo de entender a função **solve(int,int)** implementada em C++. Podendo continuar, procuramos a próxima posição branca a partir da próxima (a partir da próxima para evitar visitarmos a mesma casa que estamos). A função responsável por fazer isso em Haskell é a nextWhitePos, ilustrada abaixo.

```
{-
 Returns next white position on board,
 if there isn't none, returns last position
 instead.
 -}
nextWhitePos (i,j) board = do
    if (i > last_i)
        then do
            return (last_i,last_j)
        else do
            (color, _, _) <- board ! (i,j)
            if (color == white)
                then return (i,j)
                else (nextWhitePos (nextPos (i,j)) board)
```

Figura 7. Busca de próxima posição branca implementada em Haskell.

A função nextWhitePos recorre sobre as posições do tabuleiro, evitando casas pretas. A função chama outra auxiliar, nextPos, que retorna o próximo índice do tabuleiro seguindo a ordem de visitação row-major. Caso estejamos na última casa branca ao invocarmos esta função, ela nos retornará a última posição do tabuleiro (last_i, last_j).

O código de C++ prossegue testando se chegou no último índice, caso o has_finished retorne true é propagado pelas recursões a resposta de solved, caso contrário continua testando outra possibilidade até testar todas e então retorna unsolvable.

Em Haskell, **solve(int,int)** é representado por:

```
-- Solves Kakuro boards!
solve (i,j) board = do
    (try (i,j) board 1)
```

Em que try abstrai a iteração do loop **for (int w = 1; w < 10; w++)** da implementação em C++:

```
-- Try one iteration of work.
try (i,j) board w = do
    if (w == limit)
        then return unsolvable
        else do
            ifM ((work (i,j) board w) ==^ (1 solved))
                (return solved)
                ((try (i,j) board (w+1)))
```

Na implementação de Haskell, realiza toda a business logic do programa: realizar o teste de consistência para então avançar na árvore de computação ou realizar o backtracking; Testar se terminou e retornar solved ou unsolvable:

```
-- Skip impossible board states, also propagate finished state.
work (i,j) board w = do
    (writeArray board (i,j) (white, w, 0))
    ifM (canContinue (i,j) board w)
        (hasFinished (i,j) board)
        (return continue)
```

Por fim, resta mostrar como a aplicação de Haskell testa suas condições de término com a função hasFinished:

```
-- Test finish conditions. Calls next position to try.
hasFinished (i,j) board = do
    nextP <- (nextWhitePos (nextPos (i,j)) board)
    (color,_,_) <- board ! (last_i,last_j)
    if (isLastPos nextP) && (j == last_j || color == black)
        then return solved
        else do
            ifM ((solve nextP board) ==^ (1 solved))
                (return solved)
                (return continue)
```

A função testa se o programa está na última casa branca, se não estiver, avança na árvore de computação invocando solve na próxima casa branca. É importante lembrar que o programa só invoca hasFinished caso canContinue tenha retornado True, indicando que o tabuleiro está em um estado consistente. Portanto, se hasFinished retornar True, temos garantia que o tabuleiro foi resolvido.

Após resolvido (ou não, caso exista um tabuleiro não resolvível), imprimimos na tela a matriz representando o resultado (código da main):

```
    -- 3. Solve the puzzle.
    nex <- (nextWhitePos (0,0) board)
    sol <- (solve nex board)

    -- 4. Print solution.
    if (sol == solved)
        then sequence_ $ do
                i <- [0..last_i]
                j <- [0..last_j]
                return $ do
                (color, first, second) <- board ! (i,j)
                putStr $ if color == black
                        then "*"
                        else (show first)
                if j == last_j
                        then putChar '\n'
                        else return ()
        else
            print ("Unsolvable!")
```

Estruturas de dados:

Para este trabalho, optamos por utilizar a estrutura IOArray, contida no módulo Data.Array.IO. Ela fornece operações de leitura e escrita:

1. readArray board (i,j) :: IO Pos. (aliased para o operador (!) ao decorrer do código).

2. writeArray board (i,j) pos :: IO ().

Instanciamos a estrutura na main:

```
board <- newArray ((0,0), (height,width)) undefined
let _ = board :: Board
```

Onde especificamos o índice inicial (0,0), o tamanho (height, width) e o tipo Board:

```
type Board = IOArray (Int, Int) Pos
```

A presença de (Int, Int) no tipo indica que a estrutura será indexada na forma de uma matriz, através dos índices (i,j).

A maneira em que os testes são trazidos para o namespace da aplicação e o tabuleiro é inicializado utilizam ideias de programação generativa, cujos detalhes fogem do escopo deste relatório.

Funções auxiliares:

De maneira geral, as funcionalidades dos auxiliares estão descritas em forma de comentário acima de suas declarações. Entretanto, é importante explicar o funcionamento das operações monádicas: Como trabalhamos com a estrutura IOArray, todas as funções que interagem com o tabuleiro, ou seja, a grande maioria, opera dentro da mônada de IO, resultando em vários bindings, prejudicando a legibilidade do código. Para resolver este problema, declaramos diversas funções, como a ifM (if monádico), de modo a operar diretamente com os valores monádicos. Nos inspiramos em algumas funções existentes no módulo Control.Monad.

```
{-- START OF AUXILIARES --}

-- Alias for acessing kakuroBoard.
(!) = readArray

-- Checks if index is last of board.
isLastPos :: Index -> Bool
isLastPos (i, j) = (i == last_i) && (j == last_j)

-- Return next Position on board.
nextPos :: Index -> Index
nextPos (i,j)
        | (j == last_j) = ((i+1), (mod  (j+1) width))
        | otherwise = (i, (mod  (j+1) width))

-- Compare monadic values.
cmpM m1 m2 = do
    a <- m1
```

```
    b <- m2
    if a == b then (return True) else (return False)

-- Monadic if.
ifM act t e = do
    b <- act
    if b then t else e

-- Monadic or.
(||^) a b = ifM a (return True) b

-- Monadic and.
(&&^) a b = ifM a b (return False)

-- Monadic compare.
(==^) m1 m2 = cmpM m1 m2

-- Alias for return (used in boolean expressions to avoid cluttering).
l m = return m

{-- END OF AUXILIARES --}
```

Figura 8. Funções Auxiliares.