Modelling and analysis of cyber-physical systems now with monads

Etienne Costa
Rui Azevedo
Universidade do Mihho
Escola de Engenharia
Braga

(June 15, 2021)

Abstract

O presente trabalho é referente a resolução de problemas reais recorrendo a maquinaria dos mónades.O problema a ser abordado é sobre o caixeiro viajante tendo sindo feitas algumas implementações recorrendo ao mónade das listas, IO e duração, sendo possível constatar que as soluções não são obtidas de maneira eficiente.

1 First Part

1.1 Lambda-Calculus

)

$$\frac{}{[\![x,y : \mathbb{N} \vdash y : \mathbb{N}]\!] = \pi_2} \qquad \frac{}{[\![x,y : \mathbb{N} \vdash y : \mathbb{N}]\!] = \pi_2}$$

$$\frac{}{[\![x : \mathbb{N} \vdash x - 1 : \mathbb{N}]\!] = pred} \quad \frac{}{[\![x : \mathbb{N} \vdash x + 1 : \mathbb{N}]\!] = succ} \qquad \frac{}{[\![x,y : \mathbb{N} \vdash y - 1 : \mathbb{N}]\!] = pred} \quad \frac{}{[\![x,y : \mathbb{N} \vdash y + 1 : \mathbb{N}]\!] = succ}$$

$$\frac{[\![x : \mathbb{N} \vdash_c return(x-1) : \mathbb{N}]\!] \; [\![x : \mathbb{N} \vdash_c return(x+1) : \mathbb{N}]\!]}{} \qquad \frac{[\![x,y : \mathbb{N} \vdash_c return(y-1) : \mathbb{N}]\!] \; [\![x,y : \mathbb{N} \vdash_c return(y+1) : \mathbb{N}]\!]}{}$$

$$\frac{}{[\![x : \mathbb{N} \vdash_c choice(return\ (x-1), return(x+1)) : \mathbb{N}]\!] = f} \qquad \frac{}{[\![x,y : \mathbb{N} \vdash_c choice(return(y-1), return(y+1)) : \mathbb{N}]\!] = g}$$

$$\frac{}{[\![x : \mathbb{N} \vdash_c y \leftarrow choice(return(x-1)\ , return(x+1))\ ;\ choice(return(y-1), return(y+1)) : \mathbb{N}\ ]\!] = g^* \cdot str \cdot \langle id, f \rangle}$$

$$\begin{cases} f\ x \;=\; [\![\ choice\ ]\!] \cdot \langle\ \eta \cdot prod,\ \eta \cdot succ\ \rangle \\ g(x,y) \;=\; [\![\ choice\ ]\!] \cdot \langle\ \eta \cdot prod \cdot \pi_2\ , \eta \cdot succ \cdot \pi_2\ \rangle \end{cases}$$

$$\left(\ g^* \cdot str \cdot \langle\ id\ , f \rangle\ \right) 0$$

$$\equiv\ \{\ Def - Comp\ ;\ Def - Split\ \}$$

$$\left(\ g^* \cdot str\ \right)\ (\ id\ 0\ ,\ f\ 0\ )$$

$$\equiv\ \{\ Def - id\ ;\ Def - f\ ;\ Def - comp\ \}$$

$$g^*\ (\ str(\ 0,\ [-1\ ,\ 1]\ )\ )$$

$$\equiv\ \{\ Def - str\ \}$$

$$g^*\ [\ (\ 0, -1)\ ,\ (0,1)\ ]$$

$$\equiv\ \left\{ Def - g^* \right\}$$

$$g(0,-1)\ +\!+\ g(0,1) \equiv\ [-2,0]\ +\!+\ [0,2]\ \equiv\ [-2,0,0,2]$$

1.2 Computation of Hamiltonian cycles

O problema do caixeiro viajante é baseado em conceitos importantes da teoria de grafos, concretamente **caminho hamiltoniano** e **ciclo hamiltoniano**. Um caminho hamiltoniano é um caminho que permite passar por todos os vértices de um grafo G, não repetindo nenhum, ou seja, passar por todos uma e uma só vez . Caso esse caminho seja possível descrever um ciclo, este é denominado **ciclo hamiltoniano** em G.
E, um grafo que possua tal circuito é chamado de grafo hamiltoniano.

Para a resolução do primeiro problema proposto foram implementadas duas funções :

- **addtoEnd**
- **hCycles**

sendo que ambas foram implementadas à custa do mónade das listas .
A estrutura inicial do nosso problema é a seguinte :

```
Representa a estrutura fundamental para representar um grafo, contendo
    um tipo Node e um conjunto de construtores {A,B,C,D,E,F} do mesmo .

-- 1. Graph structure: nodes and adjacency matrix (i.e. the edges)
data Node = A | B | C | D | E | F | deriving (Show,Eq,Ord)

De seguida são representadas as nossas arestas a custa de uma função :

adj :: (Node,Node) -> Bool
adj p = case p of
  (A,B) -> True
  (A,C) -> True
  (A,F) -> True
  (B,A) -> True
  (B,C) -> True
  (B,E) -> True
  (B,F) -> True
  (C,A) -> True
  (C,B) -> True
  (C,D) -> True
  (D,C) -> True
  (D,E) -> True
  (E,B) -> True
  (E,D) -> True
  (E,F) -> True
  (F,A) -> True
  (F,B) -> True
  (F,E) -> True
  (_,_) -> False
```

```
Um caminho por sua vez é representado por um array ordenado de nodos .

type Path = [Node]
```

Listing 1: Estrutura do grafo

Relembrando o mónade das listas :

```
instance Monad [] where
    m >>= f = concat map f m
    return x = [x]
    fail s  = []
```

Listing 2: Mónade das listas

Tirando partido do **bind**(»=) do mónade das listas, definimos as seguintes funções:

```
addtoEnd :: Path -> [Node] -> [Path]
addtoEnd p ns = ns >>= (\n -> if not(elem n p) && adj(n,last p) then
    [choice(p,[n])] else [] )

A mesma pode ser traduzida para a linguagem natural :

Para cada nodo 'n' pertencente a 'ns', se o nodo 'n' não se encontra no
    caminho 'p' e o último nodo do caminho 'p' for adjacente do nodo
    'n' então é criado um caminho novo concatenando ao caminho 'p' o
    nodo 'n'.
```

Listing 3: addtoEnd

```
hCycles :: Node -> [Path]
hCycles n = do
            p <- return(return n)
            p1 <- addtoEnd p allNodes
            p2 <- addtoEnd p1 allNodes
            p3 <- addtoEnd p2 allNodes
            p4 <- addtoEnd p3 allNodes
            p5 <- addtoEnd p4 allNodes
            return p5 >>= (\p -> if (adj(last p,n)) then
                [choice(p,[n])] else [] )


Esta implementação permite-nos calcular todos os ciclos hamiltonianos a
    partir de um determinado nodo .
```

```
A última linha do código é responsável por fechar os ciclos, fazendo uma
    verificação de adjacência do último nodo de cada caminho com o nodo
    inicial .
```

Listing 4: hCycles

1.3 Resultados Obtidos

```
Representa o conjunto de ciclos obtidos a partir do nodo 'A' .

$ hCycles A

[
[A,B,C,D,E,F,A],
[A,B,F,E,D,C,A],
[A,C,D,E,B,F,A],
[A,C,D,E,F,B,A],
[A,F,B,E,D,C,A],
[A,F,E,D,C,B,A]
]
```

Listing 5: hCycles A

2 Second Part

2.1 Lambda-Calculus

Por uma questão de simplificação foi colocado o resultado da árvore de derivação e a respectiva prova pretendida :

$$[\![\ x\ :\ A\ \vdash_c\ (\ \lambda_y\ :\ A\ \rightarrow\ A.wait_2\ (y\ x)\ )\ (\lambda_z\ :\ A.wait_1\ (\ return\ z\ ))\ :\ A]\!]$$
$$=\ [\![\ x :\ A\ \vdash_c\ \ wait_3\ (return\ x)\ :\ A]\!]$$

$$\equiv\ \{\ Igualdade\ Extensional;\ swap\ =<\pi_2,\pi_1>\ \}$$

$$([\![\ wait_2\ ]\!]\ \cdot\ app\ \cdot\ swap\cdot\ <id,\lambda([\![wait_1]\!]\ \cdot\ \eta\ \cdot\ \pi_2)>\ )\ x\ =([\![wait_3]\!]\ \cdot\ \eta)\ x$$

$$\equiv\ \{\ Def-Split\ ;\ Def-Comp\ ;\ Def-\eta\ ;\ Def-wait\ ;\ Def-id\ \}$$

$$(\ [\![wait_2]\!]\ \cdot\ app\ \cdot\ swap\ )\ (x,\ \lambda(\ [\![wait_1]\!]\ \cdot\ \eta\ \cdot\ \pi_2)\ x)\ =\ (3,x)$$

$$\equiv\ \{\ Def-Swap\ ;\ Def-Comp\ \}$$

$$([\![wait_2]\!]\ \cdot\ app)\ (\lambda\ ([\![wait_1]\!]\ \cdot\ \eta\ \cdot\ \pi_2)\ x\ ,\ x)\ =\ (3,x)$$

$$\equiv\ \{\ Def-Comp\ ;\ Def-App\ \}$$

$$[\![wait_2]\!]\ (\lambda\ (\ [\![wait_1]\!]\ )\ \cdot\ \eta\ \cdot\ \pi_2)\ x\ x\ )\ =\ (3,x)$$

$$\equiv\ \{\overline{\lambda}\ f\ a\ b\ =\ f(a,b)\ \}$$

$$[\![wait_2]\!]\ (\ ([\![wait_1]\!]\ \cdot\ \eta\ \cdot\ \pi_2)\ (x,x)\ )\ =\ (3,x)$$

$$\equiv\ \{Def-Comp\ ;\ Def-\pi_2\ ;\ Def-\eta\ ;\ Def-wait_1\ \}$$

$$[\![wait_2]\!]\ (1,x)\ =\ (3,x)$$

$$\equiv\ \{Def-wait_2\ \}$$

$$(3,x)\ =\ (3,x)$$

$$\equiv\ \{\ Trivial\ \}$$

$$True$$

2.2 Computation of Hamiltonian Cycles Cost

Nesta fase foi feita a implementação real do problema do caixeiro viajante, isto é introduzindo custos à cada uma das arestas . Feito isso é possível fazer uma idealização diferente do problema inicial considerando cada nodo uma cidade e os custos associados a cada aresta o tempo de deslocação entre as mesmas. Tirando partido dos ciclos hamiltonianos é possível inferir o menor tempo para viajar por todas as cidades.

A estrutura inicial do nosso problema é a seguinte :

```
Representa a estrutura fundamental para representar um grafo, contendo
    um tipo Node e um conjunto de construtores {A,B,C,D,E,F} do mesmo .

-- 1. Graph structure: nodes and adjacency matrix (i.e. the edges)
data Node = A | B | C | D | E | F | deriving (Show,Eq,Ord)

De seguida são representados os custos de cada aresta a custa de uma
    função :

adjT :: (Node,Node) -> Maybe Int
adjT p = case p of
          (A,B) -> Just 2
          (A,C) -> Just 3
          (A,F) -> Just 6
          (B,A) -> Just 30
          (B,C) -> Just 0
          (B,E) -> Just 4
          (B,F) -> Just 3
          (C,A) -> Just 60
          (C,B) -> Just 3
          (C,D) -> Just 50
          (D,C) -> Just 2
          (D,E) -> Just 3
          (E,B) -> Just 1
          (E,D) -> Just 3
          (E,F) -> Just 2
          (F,A) -> Just 4
          (F,B) -> Just 5
          (F,E) -> Just 3
          (_,_) -> Nothing

Um caminho por sua vez é representado por um array ordenado de nodos .

type Path = [Node]
```

Listing 6: Estrutura do grafo

Com base na solução do problema anterior é possível afirmar que podemos calcular os ciclos hamiltonianos à custa do mónade das **listas** e os custos associados aos caminhos por sua vez podem ser calculados através do mónade da **duração**. Tirando partido de ambos é possível implementar uma solução para o problema do caixeiro viajante .

```
tadjacentNodes :: Node -> [Node] -> [Duration Node]
tadjacentNodes n ns = ns >>= (\n' -> if (adj(n,n')) then [ Duration
    (fromJust (adjT(n,n')), n')] else [] )


A mesma pode ser traduzida para a linguagem natural do seguinte modo :

Para cada nodo n' pertencente a ns, se o nodo n' for adjacente a n,
    então é feita
a construção do tempo necessário extraindo o tempo de viagem atravé s do
    fromJust(adjT(n,n')). De ressaltar que adjT tem como tipo de
    retorno o Maybe Int daí a necessidade de tirarmos partido da função
    fromJust que permite extrair o valor inteiro do contexto Maybe .
```

Listing 7: tadjacentNodes

```
taddToEnd :: Duration Path -> [Duration Node] -> [Duration Path]
taddToEnd p ns = ns >>= (\n -> if not (elem (getValue n) (getValue p))
    && adj( getValue n,(last.getValue) p) then [ Duration (getDuration
    n + getDuration p, choice(getValue p ,[ getValue n ])) ] else [] )


A implementação da função taddToEnd é muito similar a função addtoEnd, o
    grande diferencial está no cálculo dos custos/tempos associados a
    cada deslocação.
Tirou-se partido das funções implementadas no módulo DurantionMonad.

data Duration a = Duration (Int, a) deriving (Show,Eq)

getDuration :: Duration a -> Int
getDuration (Duration (d,x)) = d

getValue :: Duration a -> a
getValue (Duration (d,x)) = x
```

Listing 8: taddToEnd

```
hCyclesCost :: Node -> [Duration Path]
hCyclesCost n = do
                dp <- return (Duration (0,[n]))
                p1 <- f dp
                p2 <- f p1
                p3 <- f p2
                p4 <- f p3
                p5 <- f p4
                return p5 >>= (\p -> if adj((last.getValue) p, n) then
                    [Duration ( getDuration p + (fromJust $ adjT(
                    (last.getValue) p, n)) ,choice(getValue p,[n]))]
                    else [] )
           where f durationP = taddToEnd durationP (tadjacentNodes
               (last(getValue durationP)) allNodes)
```

Listing 9: hCyclesCost

2.3 Resultados Obtidos

```
Representa o conjunto conjunto de ciclos com o seu custo associado a
    partir do nodo 'A' .

*Problem2> hCyclesCost A

[
Duration (61,[A,B,C,D,E,F,A]),
Duration (73,[A,B,F,E,D,C,A]),
Duration (64,[A,C,D,E,B,F,A]),
Duration (93,[A,C,D,E,F,B,A]),
Duration (80,[A,F,B,E,D,C,A]),
Duration (47,[A,F,E,D,C,B,A])
]


*Problem2> tsp A

Duration (47,[A,F,E,D,C,B,A])
```

Listing 10: hCyclesCost tsp

3 Why Monads Matter

3.1 Contextualização

Na programação funcional, o conceito de mónade é usado para sintetizar a ideia de computação.

Uma computação é vista como algo que se passa dentro de uma **"Caixa Preta"** e da qual conseguimos apenas ver os resultados .

3.2 Motivação

Digamos que temos como desafio a modelação de um tipo de dados que nos permite representar expressões aritméticas, com constantes, variáveis, produtos, somas, subtrações e divisões inteiras .

```
data Exp = Const Int | Var String | Prod Exp Exp | Add Exp Exp | Sub Exp
    Exp | Div Exp Exp
   deriving Show
```

Listing 11: Data Type e Construtores

Tendo este desafio ultrapassado, existe a necessidade de calcular o valor de uma determinada expressão, para isto, é proposto a implementação de um novo tipo de dados "Dict" que por sua vez armazena numa lista o valor associado a uma determinada variável e implementa-se a função "eval" responsável por calcular o valor de uma determinada expressão .

```
type Dict = [(String,Int)]

eval :: Dict -> Exp -> Int
eval _ (Const n) = n
eval dict (Var x) = fromJust (lookup x dict)
eval dict (Prod e d) = (eval dict e) * (eval dict d )
eval dict (Add e d) = (eval dict e) + (eval dict d )
eval dict (Sub e d) = (eval dict e) - (eval dict d )
eval dict (Div e d) = div (eval dict e) (eval dict d )
```

Listing 12: Eval definition

Aos olhos de um bom observador ressaltam logo dois possíveis problemas :

```
A divisão de uma determinada expressão por 0 lançaria uma "exception",
    pois não é possível efectuar esta operação.

     Exemplo :
             exceptionDiv = Div (Const 4) (Const 0)
             eval [] exceptionDiv
             *** Exception: divide by zero
```

Listing 13: Primeiro problema

```
O cálculo de uma determinada operação cuja variável não se encontra
    definida no nosso dicionário .

     Seja o dict = [("x",0)]
     exceptionDict = Div (Const 4) (Var "y")
     eval dict exceptionDict
     *** Exception: Maybe.fromJust: Nothing
```

Listing 14: Segundo problema

A razão destes problemas está fortemente relacionada com o facto de estarmos a definir uma função parcial, isto é, uma função que não está definida para todos os valores do seu domínio.

Uma solução possível passa por declarar o construtor de tipos **Maybe** como instância da classe **Monad**, sendo muito útil para trabalhar com computações parciais, pois permite fazer a propagação de erros.

```
instance Monad Maybe where
    return x = Just x
    (Just x) >>= f = f x
    Nothing >>= _ = Nothing
    fail _ = Nothing
```

Listing 15: Monad Maybe

Redefinindo a nossa solução inicial temos o seguinte :

```
eval' ::Dict -> Exp ->Maybe Int
eval' _ (Const n) = return n
eval' dict (Var x) = lookup x dict

eval' dict (Prod e d) = do
                        esq <- eval' dict e
                        dir <- eval' dict d
                        return (esq*dir)

eval' dict (Add e d) = do
                        esq <- eval' dict e
                        dir <- eval' dict d
                        return (esq+dir)

eval' dict (Sub e d) = do
                        esq <- eval' dict e
                        dir <- eval' dict d
                        return (esq-dir)

eval' dict (Div e d) = do
                        esq <- eval' dict e
                        dir <- eval' dict d
                        if dir==0 then Nothing
                                  else return (div esq dir)
```

Listing 16: Eval Definition

```
A divisão de uma determinada expressão por 0 lançaria uma exception,
    visto que não é possível efectuar esta operação.

       Exemplo :
           exceptionDiv = Div (Const 4) (Const 0)
           eval' [] exceptionDiv
           Nothing
```

Listing 17: Primeira solução

```
O cálculo de uma determinada operação cuja variável não se encontra
    definida no nosso dicionário .

            Seja o dict = [("x",0)]
            exceptionDict = Div (Const 4) (Var "y")
            eval' dict exceptionDict
            Nothing
```

Listing 18: Segunda solução

Com base nisso é possível constatar que uma das principais vantagens na utilização de mónades é a conversão de funções parciais em totais sem grande esforço, conseguindo mapear todos os valores do seu domínio para algum valor do contradomínio.
Como se pode verificar na solução final obtida outra grande vantagem em declarar instâncias da classe "Monad" é que passamos a poder usar uma notação especial para escrever funções mónadicas, sendo que esta notação baseia-se na equivalência

```
d >>= \x -> e <=> do {x <- d; e }
```

Listing 19: Notação Especial

tornando explícita a noção de extrair um valor de uma computação antes de ser usado.

Em suma mónades não são apenas construções matemáticas da teoria das categorias, mas abstrações poderosas que permitem reutilizar código para resolver problemas do mundo real.

3.3 Curva de Aprendizagem

No que concerne aos mónades, a curva de aprendizagem acaba por ser o maior obstáculo para pessoas que procuram aprender o conceito, pois apesar de o mesmo ser muito útil ele carrega um certo grau de dificuldade e alguns membros da comunidade acabam, por divulgar certos artigos/tutoriais de baixa qualidade tornando mais confuso para quem procura perceber. Algumas pessoas mais supersticiosas acreditem que este facto está fortemente relacionado com a "maldição" de Douglas Crockford .

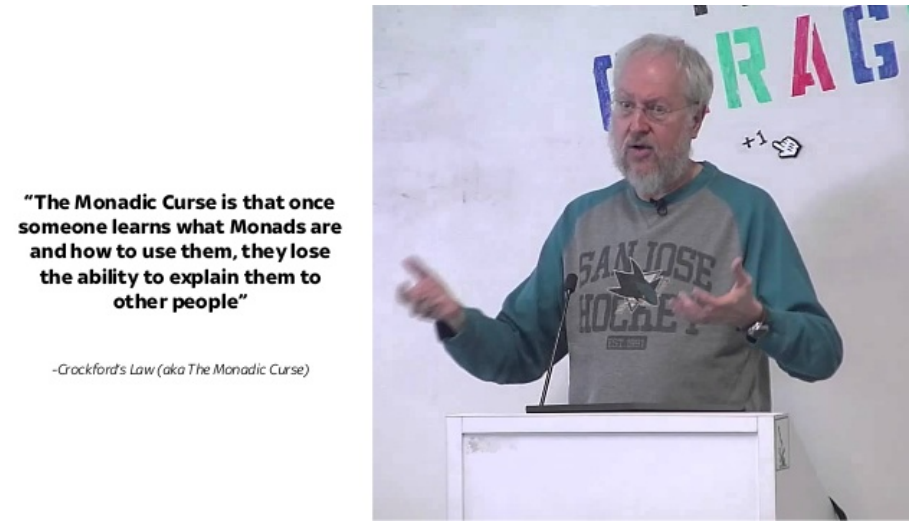


Figure 1: The Monadic Curse

4 Monads Among Other Things

Sendo esta uma fase mais exploratória o grupo acabou por focar-se na exploração dos mónades das listas e IO. Sendo assim achou-se relevante implementar as seguintes funcionalidades:

- Ser capaz de ler um grafo a partir de um ficheiro (IO Monad)
- Ser capaz de gerar dot files correspondendo a cada ciclo encontrado (IO Monad)
- Ser capaz de gerar pdfs que correspondem aos ciclos encontrados (IO Monad)
- E tornar a função genérica em relação ao problema inicial . (List Monad)

Visto que está a ser feita a reestruturação do problema do caixeiro viajante optou-se por definir a seguinte estrutura para um grafo :

```
Os nossos nodos passam a ser representados por um 'char' e as nossas
    arestas correspondem a uma lista de pares de nodos .

--  1. Graph structure:
type Node = Char
type Edge = (Node,Node)
type Edges = [Edge]

Um caminho continua a ser uma lista ordenada de nodos .
-- 2. A sequence of nodes its a path :
type Path = [Node]
```

Listing 20: Segunda solução

Tendo a nossa estrutura do grafo implementada , foi necessário implementar um conjunto de funções capazes de extrair do nosso ficheiro de entrada os nodos existentes bem com o conjunto de arestas .

```
Sendo intuitivo que a primeira linha corresponde aos nodos do nosso
    grafo e as restantes linhas ao conjunto de arestas .

A,B,C,D,E,F

(A,B)
(A,C)
(A,F)
(B,A)
(B,C)
(B,E)
(B,F)
(C,A)
(C,B)
(C,D)
(D,C)
(D,E)
(E,B)
(E,D)
(E,F)
(F,A)
(F,B)
(F,E)
```

Listing 21: Representação do Grafo

```
É feita a interação com o utilizador requisitando o ficheiro que contem
    o grafo
e o respectivo nodo inicial .

openFile :: FilePath -> IOMode -> IO Handle

Tirando partido da função 'openFile' foi possível obter o ficheiro e
    guardar num IO Handle. O Haskell por sua vez define operações que
    permitem ler e escrever carateres de e para ficheiros representados
    por valores do tipo Handle.


hGetContents :: Handle -> IO String
De modo a obter o conteúdo do ficheiro usou-se o hGetContents para de
    seguida usar as nossas funcções auxiliares .


main :: IO ()
main = do
   putStrLn "Insert the file name: "
```

```
    file <- getLine
    putStrLn "Insert the initial node: "
    node <- getLine
    handle <- openFile file ReadMode
    contents <- hGetContents handle
    let
        graph = splitAt 1 . words $ contents
        nodes = allNodes.splitOn "," . head . fst $ graph
        size = length nodes -1
        edges = listAdj.snd $ graph
        cycles = hCycles node nodes edges size

    mapM_ write cycles
    hClose handle
```

Listing 22: Leitura Ficheiro

```
De seguida foi implementada um função mais gé nerica, sabendo que o
    número de iterações para fechar um determinado ciclo é igual ao
    número de nodos - 1 .

hCycles ::[Node] -> [Node] -> Edges->Int -> [Path]
hCycles n nodes edges counter = do
          newpath <- addtoEnd n nodes edges
          if counter /= 1 then hCycles newpath nodes edges (counter - 1)
                          else
                               if (elem (last newpath) (adjacentNodes
                                    (head newpath) edges)) then
                                    [choice(newpath,[head n] )] else []


As restantes implementações acabam por ser muito similares ao que já
    tínhamos definido numa primeira instância do problema .

addtoEnd :: Path -> [Node] -> Edges -> [Path]
addtoEnd p ns edges = ns >>= (\n -> if not(elem n p) && (elem (last p)
    (adjacentNodes n edges)) then [choice(p,[n])] else [] )


adjacentNodes :: Node -> Edges -> [Node]
adjacentNodes n edges = edges >>= (\edge -> if n==fst edge then return
    (snd edge) else [] )
```

Listing 23: Caixeiro Viajante

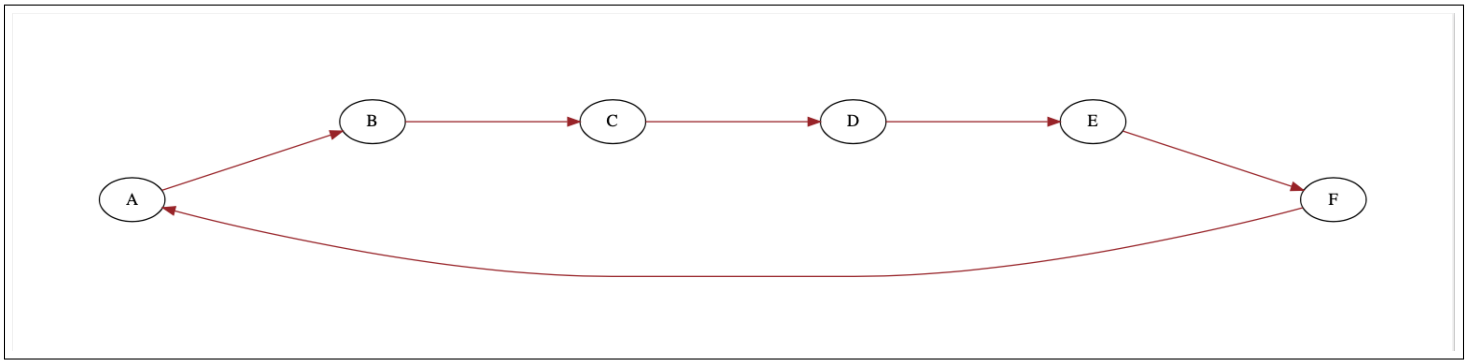
```
Definição do conjunto de cores para a coloração das arestas .

type Colors = [String]
colors::Colors
colors = ["black","gray","blue","red","green","yellow","purple","brown"]


Permite selecionar uma cor de forma aleatória .

randomColors :: [a] -> IO a
randomColors l = do
    i <- randomRIO (0, length l - 1)
    return $ (!!) l i



Funções responsáveis para fazer a geração da respectiva String para o
    formato DOT que de seguida é são criados os respectivos ficheiros à
    custa de system calls.

strToEdges::String->String-> String
strToEdges _ [] = []
strToEdges _ [x] = []
strToEdges color (origin:destiny:t) = [origin]++" ->
    "++[destiny]++"[style=solid, color="++color++"];\n"++ (strToEdges
    color (destiny:t))



write :: Path -> IO()
write path = do
    putStrLn ("Generating the"++path++".pdf")
    color <- randomColors colors
    let
        cycle = strToEdges color path
    appendFile ("./"++path++".dot") "digraph g {\n graph [pad=\"0.5\",
        nodesep=\"1\", ranksep=\"2\"] \nrankdir=LR\n"
    appendFile ("./"++path++".dot") cycle
    appendFile ("./"++path++".dot") "}"
    system ("dot -Tpdf "++path++".dot > "++path++".pdf") >>= \exitCode
        -> print exitCode
    system ("open "++path++".pdf") >>= \exitCode -> print exitCode
    print "File Created ..."
```

Listing 24: DOT Cycles e PDF Cycles

4.1 Resultados Obtidos

```
digraph g {
 graph [pad="0.5", nodesep="1", ranksep="2"]
rankdir=LR
A -> B[style=solid, color=brown];
B -> C[style=solid, color=brown];
C -> D[style=solid, color=brown];
D -> E[style=solid, color=brown];
E -> F[style=solid, color=brown];
F -> A[style=solid, color=brown];
}
```

Listing 25: DOT FILE




```
Representa o conjunto de ciclos obtidos a partir do nodo 'A' .
[
[A,B,C,D,E,F,A], --> Ciclo representado na imagem acima .
[A,B,F,E,D,C,A],
[A,C,D,E,B,F,A],
[A,C,D,E,F,B,A],
[A,F,B,E,D,C,A],
[A,F,E,D,C,B,A]
]
```

Listing 26: hCycles A

5 Conclusão

No que concerne aos mónades a nossa exploração foi direcionada aos mónades das listas e ao IO, pois achamos relevante a representação das soluções obtidas num formato mais amigável, podendo o utilizador final tirar partido destas soluções em futuras aplicações . Tendo implementado as funcionalidades que achamos relevantes recorrendo exclusivamente a estes mónades, a nossa parte de exploração acabou por estar limitada pois temos noção que o mónade de estados é mais preponderante e permite diversas aplicações em diferentes domínio, não obstante do grau de dificuldade que o mesmo acarreta .