Exercícios sobre Mônadas

Prof.: Carlos Camarão

29 de Maio de 2017

1 Funtores

Funtores em Haskell são construtores de tipos que são instâncias da classe `Functor`, definida no `Prelude`, para os quais é definida a função chamada em Haskell de *fmap*, que permite aplicar uma função internamente a todos os elementos de uma estrutura de dados, mantendo alterada essa estrutura:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Por exemplo, as funções `map` e `mapTree` com os tipos a seguir devem ser definidas de modo a respectivamente aplicar uma função fornecida como argumento a cada um dos elementos de uma lista ou árvore:

```
map :: (a -> b) -> [a] -> [b]
mapTree :: (a -> b) -> Tree a -> Tree b
```

A função `map` é uma instância de `fmap` para listas (i.e. uma instância na qual `f` é o construtor de tipos `[]`) e, analogamente, `mapTree` é uma instância de `fmap` para árvores (i.e. uma instância na qual `f` é igual ao construtor de tipos `Tree`). A definição do tipo `Tree` pode ser feita por exemplo como a seguir:

```
data Tree a = Folha a | Nodo (Tree a) (Tree a)
```

As definições das instâncias de `Functor` para listas e árvores podem ser feitas como a seguir:

```
instance Functor [] where
  fmap = map

instance Functor Tree where
  fmap f (Folha a) = f a
  fmap f (Nodo t t') = Nodo (fmap f t) (fmap f t')
```

Essencialmente, um funtor permite aplicar uma função internamente ao construtor de valor da estrutura de dados que é instância da classe *Functor*, sem modificar a estrutura.

Por exemplo, `fmap (+1) [1,2,3]` retorna `[2,3,4]`. A ação `fmap (fmap toUpper) getLine` retorna uma ação (de tipo `IO String`) que lê uma linha do dispositivo de entrada padrão e transforma as letras dessa linha em letras maiúsculas.

1.1 Exercícios Resolvidos

1. Defina instância de `Functor` para o construtor de tipo `Maybe`.

   Solução:

   ```
   instance Functor Maybe where
     fmap _ Nothing  = Nothing
     fmap f (Just a) = Just (f a)
   ```

2. Implemente instâncias de `Functor` para os construtores de tipos algébricos `Par`, `Q a`, `T a`, definidos abaixo:

```
data Par a = Par a a
data Q a b = Q a a b
data T a b = T a b b
```

Solução:

```
instance Functor (Par a) where
  fmap f (Par a a') = Par (f a) (f a')

instance Functor (Q a) where
  fmap f (Q a b) = Q a a (f b)

instance Functor (T a) where
  fmap f (T a b b') = T a (f b) (f b')
```

3. Considere as propriedades de funtores a seguir:

```
id_funtor :: (Functor f, Eq (f a)) => f a -> Bool
id_funtor x = fmap id x == x

comp_funtor::(Functor f, Eq(f c))=>(a->b)->(b-> c)->f a-> Bool
comp_funtor f g x = fmap g (fmap f x) == fmap (f . g) x
```

Prove estas propriedades para os construtor de tipo algébrico `Par` definido no exercício acima.

Solução:

```
    fmap id (Par a a')
= { def. fmap para Par }
    Par (id a) (id a')
= { def. id }
    Par a a'

    fmap (f . g) (Par a a')
= { def. fmap para Par }
    Par ((f . g) a) ((f . g) a')
= { def. (.) }
    Par (f (g a)) (f (g a'))
= { def. fmap para Par }
    fmap f (Par (g a) (g a'))
= { def. fmap para Par }
    fmap f (fmap g (Par a a'))
```

1.2 Exercícios

1. Implemente instâncias de `Functor` para os construtores de tipos algébricos `Id`, `Tres a b`, definidos abaixo:

```
data Id a = Id a
data Tres a b c = Tres a b c
```

2. Prove as propriedades `id_funtor` e `comp_funtor`, definidas no exercício resolvido 3 da subseção anterior, para os construtores de tipos algébricos definidos no exercício acima.

3. Considere os seguintes tipos algébricos:

(a) `data AouB = A | B`

(b) `data F f a = F (f a)`

(c) `data G a f = G (f a)`

(d) `data X a b = A a | B b`

Para cada construtor de tipo `AouB`, `F f`, `G a`, `X a`, indique se existe instância de `Functor` para ele e, se existir, defina-a, caso contrário justifique porque não.

Dicas:

(a) em 3b considere a possibilidade de `f` ser da classe `Functor`;

(b) em 3c, considere os super-tipos (*kinds*) de `f` e de `G a`.

2 Funtores Aplicativos

A função `fmap` permite aplicar uma função `f` a um valor que está encapsulado em um construtor de uma estrutura de dados. Considere que o construtor não é conhecido para o usuário da estrutura de dados (não foi exportado pelo módulo que define a estrutura, digamos para não fornecer detalhes da implementação da estrutura). Para definir uma função `f :: Int -> X -> X` de modo que `f y` incremente valor `x` interno a `X` para que fique, internamente à estrutura de dados `X`, igual a `x+y`, podemos definir:

```
f y = fmap (+y)
```

No entanto, suponha que queremos definir `g :: X -> X -> X` sem usar construtores internos da estrutura de dados `X` de modo a retornar a soma, encapsulada em `X`, dos valores internos aos argumentos de `g`. Não há como usar `fmap`. Podemos no entanto encapsular a função a ser aplicada internamente em `X`, usando a classe `Applicative`:

```
class Functor f => Applicative f where
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

`pure` encapsula a função `+` e `(<*>)` aplica funções obtidas internamente a seus argumentos:

```
g xy xx = pure (+) <*> xy <*> xx
```

ou:

```
g xy xx = fmap (+) xy <*> xx
```

`(<*>) ff fx` difere de `fmap` devido ao fato de que a função aplicada (`ff`) também está encapsulada; o efeito de aplicar a função ao valor encapsulado, obtendo o resultado encapsulado é similar.

Entretanto, o que não permite que um funtor aplicativo possa ser usado é o fato de existir alguma informação de estado gerada por `ff` que necessite ser usada para determinar o comportamento de `fx` (como, por exemplo, o número de vezes que `ff` foi anteriormente chamada). Nestes casos, é necessário o uso de mônadas, que é o assunto abordado na seção 3.

2.1 Exercícios

1. Escreva instâncias de `Applicative` para os seguintes tipos:

```
data Pair a   = Pair a a
data Two a b = Two a b
```

2. Considere a definição:

```
sequenceAL:: Applicative f => [f a] -> f [a]
sequenceAL []     = pure []
sequenceAL (fa:fx) = pure (:) <*> fa <*> sequenceAL fx
```

Uma outra forma de escrever a segunda equação de sequenceAL é

```
sequenceAL (fa:fx) = fmap (:) fa <*> sequenceAL fx
```

ou, usando a forma infixada `(<$>)` de `fmap`:

```
sequenceAL (fa:fx) = (:) <$> fa <*> sequenceAL fx
```

Usando funções monádicas, podemos também escrever:

```
sequenceAL []     = return []}
sequenceAL (fa:fx) = return (:) `app` fa `app` sequenceAL fx
```

onde:

```
app:: Monad m => m (a->b) -> m a -> m b
app mf mx = do f <- mf
               x <- mx
               return (f x)
```

Considere o seguinte programa:

```
import Control.Applicative

sequenceAL:: Applicative f => [f a] -> f [a]
sequenceAL []     = pure []
sequenceAL (fa:fx) = (:) <$> fa <*> sequenceAL fx

main = sequenceAL [putStr "a", putStrLn "b", putStrLn "c"]
```

A execução de main produz:

```
ab
c
```

Um outro exemplo:

```
f x y = (x ++ " -- arg1      ") ++ (y ++ " -- arg2")
xy    = return f <*> getLine <*> getLine
main  = xy >>= putStrLn
```

Use funções da classe Applicative para ler duas cadeias de caracteres que são números inteiros e retornar a soma desses inteiros.

3. Considere o registro e a função definidas a seguir:

```
data InfoPessoa = InfoP { nome::String,
                          ocup::String,
                          area::String }

getInfoP:: [(String,Maybe String)] -> Maybe InfoPessoa
getInfoP x = case lookup "nome" x of
               Just (Just nome@(_:_)) ->
                 case lookup "ocup" x of
                   Just (Just ocup@(_:_)) ->
                     case lookup "area" x of
                       Just (Just area@(_:_)) ->
                         Just (InfoP nome ocup area)
                       _ -> Nothing -- falta area
                   _ -> Nothing -- falta ocup
               _ -> Nothing -- falta nome
```

Redefina `getInfoP` usando a instância de `Maybe` para `Applicative`, de modo a evitar o código com testes seguidos acima, um após o outro. Use a função `liftA3`.

4. Considere o seguinte programa, que ilustra o uso de `(<*>)` da classe `Applicative`:

```
import Control.Applicative ((<*>))
import Data.Map            (Map,findWithDefault,empty,insert)

type Env = Map String Int

data Exp = Var String | Val Int | Add Exp Exp

eval:: Exp -> Env -> Int
eval (Var x)   e = fetch x e
eval (Val i)   _ = i
eval (Add p q) e = eval p e + eval q e
----------------------------------------
eval':: Exp -> Env -> Int
eval' (Var x)   = fetch x
eval' (Val i)   = const i
eval' (Add p q) = const (+) <*> eval p <*> eval q

fetch:: String -> Env -> Int
fetch v = findWithDefault
            (error ("undef. var. " ++ show v ++ "\n")) v

ex_eval1  = eval  (Add (Val 1) (Val 2)) empty
ex_eval1' = eval' (Add (Val 1) (Val 2)) empty
ex_eval2  = eval  (Add (Val 1) (Var "x")) mx
ex_eval2' = eval' (Add (Val 1) (Var "x")) mx

mx  = insert "x" 2 empty
```

Redefina o programa de modo a importar `Data.List` em vez de `Data.Map`.

3 Mônadas

Existem dois *combinadores monádicos* principais, **return** e `>>=`: **return** faz com que se entre em uma mônada (encapsula um valor), e `>>=` (costuma-se dizer, em inglês, *bind*; leia "o sequenciador", ou "combinador de composição monádica") permite sequenciar ações monádicas. Esses combinadores constituem a ferramenta principal usada na manipulação de de valores monádicos

e na estruturação de programas para tornar a modificação de programas em Haskell restrita à definição da mônada. A estruturação de programas com mônadas é abordada na seção 3.5.

A característica fundamental de mônadas é que é possível, usando o sequenciador, obter o valor não monádico resultante da expressão monádica anterior de modo a modificar o processamento do valor monádico seguinte.

Apesar de ser permitido sequenciar ações monádicas, cada ação dependendo do valor (desencapsulado) retornado por valores encapsulados em ações monádicas anteriores, para mônadas que são ações de entrada e saída, não há como "sair da mônada", isto é, obter um valor final não monádico. Por exemplo, não é possível obter valor de algum tipo t a partir de um valor monádico $IO\ t$ (no entanto, já falamos anteriormente de *unsafePerformIO*).

Em Haskell, uma mônada é uma classe definida como a seguir:

```
class Monad m where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a
```

Dado um tipo monádico (instância da classe `Monad`) `m t`, vamos chamar de `t` o tipo-alvo do tipo monádico. Valor monádico é um valor de tipo monádico e expressão monádica é uma expressão de tipo monádico.

Em qualquer instância da classe `Monad`, a composição de valores monádicos é feita com o operador `(>>=)`, sendo a combinação feita, em geral, de modo que o valor resultante do processamento do primeiro valor monádico é passado como argumento do segundo.

Isso permite que a computação com valores do tipo monádico seja feita sequencialmente, porque um valor resultante do processamento de um valor monádico é passado como argumento de uma função que recebe esse valor para processamento do próximo valor monádico. É essa característica, de controle de como o processamento sequencial de valores monádicos é feita internamente na definição do combinador `(>>=`, em vez de explicitamente nos usos do valor monádico em programas, que proporciona uma forma de estruturar programas com mônadas, descrita na seção 3.5.

A composição sequencial de dois valores também pode ser feita com o operador `(>>)`, mas nesse caso o valor resultante do processamento do primeiro valor monádico é descartado (em vez de ser passado para o segundo). O operador `(>>)` tem definição default em termos de `(>>=)`:

```
(m >> m') = m >>= \ _ -> m'
```

A função **return** apenas transforma o argumento em um valor monádico, em geral simplesmente etiquetando o argumento com um construtor do tipo-alvo do tipo monádico.

A função `fail` receve uma cadeia de caracteres e retorna um valor de tipo monádico que indica uma situação de erro. Essa função não é parte da definição matemática de mônada, é usada com a notação **do** (veja seção 3.2 a seguir), quando ocorre falha em casamento de padrões.

3.1 Ações

A solução adotada em *Haskell* para permitir o uso de ações com efeito colateral é explicada a seguir. Chamamos de *ação* uma construção da linguagem cuja execução provoca uma "mudança de estado", como modificar o valor armazenado em uma variável ou realizar alguma operação de entrada e saída.

Observação: *ação com efeito colateral* é um nome mais ortodoxo, uma vez que a palavra *ação* é usada em Haskell para denotar qualquer valor monádico — isto é, qualquer valor de tipo $m\ a$, para qualquer tipo a e qualquer construtor de tipo m que é instância da classe `Monad`. No entanto, o uso do termo "ação" para qualquer valor monádico é indevida: se não provocam efeito colateral, esses valores são valores como outros quaisquer. Neste livro, usamos consistentemente "ação" com o sentido que é usualmente o sentido de "ação com efeito colateral".

Dizemos também que uma ação é *executada*, mas um valor monádico (que pode ou não ser uma ação) é *processado* (executado se for uma ação e avaliado se não for).

A solução adotada em Haskell é baseada na seguinte ideia, bastante simples:

Um construtor é usado para distinguir uma ação do resultado que ela produz.

Em Haskell, valores de tipo *IO a*, para um tipo `a` qualquer, são ações. O tipo que é instância de `a` é o tipo do valor retornado pela execução da ação.

Valores do tipo `ST s a` também são ações. Neste tipo, a variável de tipo `a` pode ser instanciada, como qualquer variável de tipo, mas a variável de tipo `s` tem uma característica especial: ela não pode ser instanciada em argumentos de `runST` (veja explicação a seguir).

Nesses argumentos, a variável `s` existe apenas para identificar ou dar um nome a, digamos, um "fluxo de execução". O termo "fluxo de execução" é apenas um nome dado ao argumento de `runST` (uma sequência de ações quaisquer), de tipo `ST s a` (para algum `a` e, como vamos ver, para um `s` que não pode ser instanciado). Tipos (`STRef s a`) de variáveis que podem ser modificadas são "etiquetadas" com esse `s`. Isso é feito para que ações que modificam o valor dessas variáveis só possam existir nesse argumento (diz-se "nesse fluxo de execução"), isto é, para que uma variável etiquetada não pode ser usada em outro fluxo de execução. No entanto, o resultado obtido na execução desse argumento (nesse "fluxo de execução") pode ser usado em outro fluxo de execução.

É possível em Haskell, dessa forma, modificar o valor armazenado em uma "variável", no sentido de "variável" em linguagens e programas imperativos. Em Haskell, tais entidades são chamadas de referências e devem ter tipo `STRef s a`, para algum `a` e para algum `s`. (O tipo `IORef` também pode ser usado para modificação do valor armazenado em variáveis, mas não carrega etiqueta, e por isso o valor armazenado não pode "sair" da mônada `IO`). O tipo `STRef s a` indica que a referência (variável, no sentido imperativo) é local ao argumento de `runST` (diz-se "é local a um determinado 'fluxo de execução'); essa etiqueta não pode "sair" da mônada `ST`, ou seja, não pode ser usada em nenhum outro argumento a `runST`. As operações abaixo, que manipulam valores de tipo `STRef s a`, têm resultado de tipo `ST s a` (para algum `a`,`s`):

```
newSTRef :: a -> ST s (STRef s a)
readSTRef :: STRef s a -> ST s a
writeSTRef :: STRef s a -> a -> ST s ()
```

A função `newSTRef` recebe um valor de um tipo *a* e retorna uma ação que, quando executada, retorna uma referência para esse valor.

A função `readSTRef` recebe uma referência a um valor de tipo `a` e retorna uma ação que, quando executada, retorna o valor armazenado no endereço denotado por essa referência.

A função `writeSTRef` recebe uma referência a um valor de tipo `a` e um valor de tipo `a` e retorna uma ação que, quando executada, armazena esse valor no endereço denotado pela referência.

Agora você vai poder entender mais precisamente o que dissemos acima. O único modo de usar um valor produzido pela execução de uma ação de tipo `ST s a` (para algum `a`) em um contexto que não envolve o construtor `ST`, é por meio do uso da função `runST`, que tem um tipo peculiar:

```
(forall s. ST s a) -> a
```

Esse é um tipo diferente dos tipos que vimos até agora. É chamado de um tipo polimórfico de *grau 2* (em inglês, *rank 2*). Em todos os tipos que vimos até agora, o quantificador `forall` ocorria externamente, antes do tipo não quantificado. Tais tipos têm *grau 1*. Um tipo de grau 2, como o de `runST`, *requer* que o argumento tenha um tipo no qual o estado (`s`) não seja instanciado.

Considere por exemplo a diferença entre os tipos

$t_1 =$ `forall` a`.` a `-> Int` e

$t_2 =$ `(forall` a`.` a`) -> Int`

O único valor de tipo `forall` a`.` a (tipo do domínio de t_2) é `undefined`. No entanto, o tipo t_1 pode ser instanciado para inúmeras instâncias. Por exemplo, `Int -> Int` é instância de t_1.

O tipo de `runST` evita o uso de `runST` em expressões como:

```
let t = runST (newSTRef True)
    f = runST (writeSTRef t False)
in runST (readSTRef t)
```

O motivo é que essa expressão não é tipável: note que `newSTRef True` tem tipo:

```
ST s (STRef s Bool)
```

A expressão `runST (newSTRef Bool)` teria então que ter tipo `STRef s Bool`, ou seja, uma variável de tipo `a` teria que ser unificada com `STRef s Bool`. No entanto, isso modificaria o tipo requerido por `runST`, porque o tipo do resultado de `runST (STRef s Bool)` passa a depender de `s` (é comum dizer: a variável de tipo *s* não pode "escapar" para o tipo do resultado).

A presença da variável `s` em um valor de tipo `STRef s t` (para um `t` qualquer) explica porque existem dois tipos distintos de referências em Haskell, construídas com `STRef` e com `IORef`. O construtor `IORef` não tem tal variável de estado; um valor é construído apenas com um valor do tipo que desejado (para qualquer `t`, o tipo `IORef t` é um tipo de referência para valor de tipo `t`). Em vez de existir uma condição de uso de `s` (que requer que `s` não possa ser instanciada e não possa, assim, ser usada em tipo de outro argumento para `runST`, requer-se que não haja, ou pelo menos não seja usada em geral, nenhuma função que "saia da mônada" `IO` (ou seja, requer-se que `unsafePerformIO` seja usada somente quando não há problema em identificar uma ação com o resultado de sua execução).

O exercício resolvido 3 ilustra o uso da mônada `ST`.

Para ações de entrada e saída de dados, é usado o construtor `IO`. Por exemplo, a ação de ler um caractere, `getChar`, tem tipo `IO Char`. Como já mencionado, o uso do construtor `IO` indica que se trata de uma ação, e distingue a ação do resultado obtido em sua execução, que é um valor de tipo `Char`.

Note:

Uma ação de entrada e saída tem tipo `IO t`; `t` é o tipo do resultado de sua execução.

Essa característica se apoia na condição de que não seja possível definir uma função de tipo `IO a -> a`, para qualquer tipo *a*. Note no entanto que existe uma função, disponível na versão de Haskell implementada pelo compilador GHC [2] com esse tipo: a função `unsafePerformIO`. Mas, como o próprio nome indica, seu uso não é em geral seguro, ela só deve ser usada em circunstâncias em que a mudança de estado provocada pela ação de entrada ou saída não tem efeito indesejado, ou seja, pode ser usada sem destruir a relevância da distinção entre a ação e o valor retornado em sua execução.

As condições sobre o uso de ações em Haskell são essenciais para garantir que a linguagem seja uma linguagem funcional pura, com *transparência referencial*.

Em Haskell, a função `main`, que representa o efeito de executar o programa resultante de uma compilação, é uma ação, de tipo `IO()` (o parâmetro `()` do construtor `IO` indica que não há valor útil retornado como resultado da execução dessa ação). A execução de toda ação em um programa Haskell é iniciada a partir da execução da ação definida na função `main`.

Existem ações básicas de leitura e escrita, como por exemplo `getChar` e `putChar` para leitura e escrita de um caractere, e `getLine` e `putStr` para leitura e escrita de uma lista de caracteres, que serão abordadas na seção 3.4.

Na seção seguinte é apresentada uma introdução a mônadas em Haskell. Em Haskell, ações e valores monádicos em geral são usados como argumentos de "combinadores" (funções de ordem superior, não locais a outras funções, geralmente usadas na construção e combinação de valores de um determinado tipo), que proveem uma maneira de estruturar programas, como abordado nas seções a seguir.

3.1.1 Exercícios Resolvidos

1. Escreva função `sequence_` que recebe uma lista de valores monádicos e retorne o resultado de processá-los sequencialmente (um após o outro), desprezando o resultado obtido com seu processamento.

   Solução:

```
sequence_ :: Monad m => [m a] -> m ()
sequence_ = foldr (>>) (return ())
```

Nota: o uso de `_` como sufixo do nome de uma função é usado em geral em Haskell para indicar que os resutado do valor monádico é `()`, sendo desconsiderados resultados de argumentos resultantes de processamento de valores monádicos.

2. Escreva função `sequence` que recebe uma lista de valores monádicos e retorne resultado que consiste em processá-los sequencialmente, retornando a lista dos resultados obtidos com seu processamento.

```
sequence :: Monad m => [m a] -> m [a]
sequence = foldr consRes (return [])
  where consRes ma m = do {a <- ma; x <- m; return (a:x)}
```

3. Escreva função `for_` que recebe um inteiro n e um valor monádico m e retorna resultado de processar m exatamente n vezes, desconsiderando os resultados obtidos pelos processamentos.

   Solução:

```
for_ :: Monad m => Int -> m a -> m ()
for_ n = sequence_  . replicate n
```

4. Escreva função `mapM_` que recebe uma função de tipo `a -> m b`, uma lista de valores de tipo `a` e retorne o resultado de processar sequencialmente os valores obtidos pela aplicação da função a cada elemento da lista.

```
mapM_ f = sequence_ . map f
```

5. Escreva função `mapM` que recebe uma função de tipo `a -> m b`, uma lista de valores de tipo `a` e retorne retorne uma lista dos resultados obtidos com os valores monádicos resultantes da aplicação da função a cada elemento da lista.

```
mapM f = sequence . map f
```

6. Escreva função **while** que recebe uma ação `b` com resultado booleano (i.e. de tipo `IO Bool`), um comando `c`, de tipo `IO a`, e executa `b` e em seguida `c` repetidamente, `c` se e somente se o resultado da execução de `b` seja igual a `True`, terminando a execução de **while** `b c` quando a execução de `b` retornar resultado `False`.

```
while:: IO Bool -> IO a -> IO ()
while b c = do rb <- b
               if rb then c >> while b c
                     else return ()
```

3.2 Notação do

A notação **do** pode ser usada, em vez de `(>>=)`, para combinação de valores monádicos. A notação é definida pelas seguintes equações (que funcionam como regras de tradução), onde `m` é uma expressão monádica, `cls` é uma sequência não vazia de cláusulas, componentes da notação, que termina com uma expressão monádica (i.e. a última cláusula não é da forma `v <- m`), sendo uma cláusula ou uma expressão monádica ou uma expressão da forma `v <- m` (onde `v` é um nome de variável), e *cls'* a tradução de *cls*:

```
do { m }           = m
do { m; cls }      = m >> cls'
do { v <- m; cls } = m >>= \ v -> cls'
```

Note que, por exemplo, as expressões:

```
do { c <- getChar }
```

e

```
do { c <- getChar; c' <- getChar}
```

não são sintaticamente corretas: a última cláusula em uma notação **do** tem que ser um valor monádico, i.e. não pode ser da forma `c <- m`.

3.2.1 Exercícios resolvidos

1. Escreva um programa que usa `getChar` para ler dois caracteres, um depois do outro, e imprimir uma cadeia contendo os dois caracteres lidos, na ordem em que foram lidos, usando `putStr`, usando combinador monádico `(>>=)`. Reescreva usando a notação **do**.

```
main = getChar >>=
          (\ c1 -> getChar >>=
          \ c2 -> getChar -> putStr [c1,c2]))
```

   Usando a notação **do**:

```
main = do c1 <- getChar
          c2 <- getChar
          putStr [c1,c2]
```

2. Considere as duas definições abaixo:

```
copiaAteLinhaVazia = do lin <- getLine
                        let copia = do
                              if (lin == "")
                                then return ()
                                else do putStrLn lin
                                        lin <- getLine
                                        copia
                        copia
```

```
copiaAteLinhaVazia = do
  l <- getLine
  if null l then return ()
            else putStrLn l >> copiaAteLinhaVazia
```

   Uma delas não copia linhas da entrada até que seja lida uma linha vazia. Qual e porquê?

   Solução: A primeira.

   A primeira função lê uma linha e só termina se esta linha for uma linha vazia; caso contrário, qualquer chamada a `copiaAteLinhaVazia` não termina: a cláusula `lin <- getLine` usada após o teste de `(lin == "")`, na notação **do**, não modifica o valor denotado pela variável `lin` usada nesse teste. Essa cláusula cria nova variável, de nome `lin`, que tem o mesmo nome de variável que já existe (em escopo mais externo; a variável mais externa fica invisível devido à criação da variável com mesmo nome em escopo mais interno). Podemos reescrever a primeira função `copiaAteLinhaVazia` sem usar a notação **do** para tornar isso explícito:

```
copiaAteLinhaVazia = getLine >>= \lin ->
                        let copia
                             | (lin=="") = return ()
                             | otherwise =
                                putStrLn lin >>
                                getLine >>= \lin->copia
                        in copia
```

3. Escreva uma definição de uma função que receba um inteiro positivo `n` e retorne o `n`-ésimo número de Fibonacci, usando a mônada `ST` para evitar chamadas recursivas de funções.

```
fib :: Int -> ST s Integer
fib n = do a <- newSTRef 0
           b <- newSTRef 1
           for_ n (do x <- readSTRef a
                      y <- readSTRef b
                      writeSTRef a y
                      writeSTRef b (x+y)
                  )
           readSTRef a
```

3.3 O que é uma mônada

Regras monádicas expressam que tipos monádicos devem ter cláusulas que podem ser simplificadas como se espera. As simplificações significam que **return** é um elemento identidade à direita e à esquerda, e que `>>=` "se assemelha a um combinator associativo":

$$
\begin{aligned}
m \texttt{ >>= } return &= m \\
return\ x \texttt{ >>= } f &= f\ x \\
(m \texttt{ >>= } f) \texttt{ >>= } g &= m \texttt{ >>= (\ } x \texttt{ -> (} f\ x \texttt{ >>= } g\texttt{))}
\end{aligned}
$$

As regras podem ser escritas também, de modo mais simétrico, usando o seguinte combinador:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> (a -> m c)
(f >=> g) x = f x >>= g
```

Esse combinador é semelhante ao operador de composição de funções, mas os tipos dos resultados dos argumentos (funcionais) são monádicos, e a ordem dos argumentos é inversa à ordem do operador de composição de funções (i.e. o argumento é passado para a primeira função, e o resultado é passado para a segunda função, ao contrário do que ocorre no caso do operador de composição de funções usual).

Esse combinador é chamado de composição-Kleisli, e está definido na biblioteca `Control.Monad`, como:

```
(f >=> g) x = f x >>= g
```

As regras monádicas expressas por meio do combinador de composição-Kleisli indicam que o combinador é associativo com identidade **return** — ou seja, é um monóide (um monóide é um conjunto de valores com uma operação binária associativa e um elemento identidade). Assim, uma mônada é um monóide se considerarmos o combinador de composição-Kleisli como o operador de composição e **return** como o elemento identidade. No entanto, o uso de `>>=` em vez de `>=>` facilita a programação, e é preferido como combinador básico da classe `Monad`.

3.3.1 Exercícios Resolvidos

1. Expresse as regras monádicas usando a notação **do**.

```
do {x <- p; return x} = do {p}

do {x <- return e; f x} = do {f e}

do {y <- do {x <- p; f x}; g y}
= do {x <- p; do {y <- f x; g y}}
= do {x <- p; y <- f x; g y}
```

2. Suponha que `(>=>)` esteja definido e defina `(>>=)` em termos de `(>=>)`.

   Solução:

```
(m >>= f) = (id >=> f) m
```

   Alternativamente:

```
(>>=) = flip (id >=>)}
```

3. Mostre que `(f >=> g) . h = (f . h) >=> g`.

   Solução:

```
 ((f >=> g) . h) x
=   { def. (.) }
 (f >=> g) (h x)
=   {def. (>=>) }
 f (h x) >>= g
=   {def. (.)}
 (f . h) x >>= g
=   {def. (>=>)}
 ((f . h) >=> g) x
```

3.3.2 Exercícios

1. Defina a versão dual da combinador de composição-Kleisli, de tipo:

```
(<=<):: Monad m => (b->m c)->(a->m b)->(a->m c)
```

2. Reescreva as regras monádicas usando o combinador de composição-Kleisli.

3.4 Mônada *IO*

As seguintes operações são definidas em Haskell para entrada e saída de dados.

- **Funções de leitura do dispositivo de entrada padrão**
  - `getChar:: IO Char`: Lê um caractere.
  - `getLine:: IO String`: Lê uma linha.
  - `getContents:: IO String`: Retorna todo o conteúdo da entrada padrão como uma lista de caracteres, que é lida de fato à medida que essa lista é usada.
  - `interact:: (String -> String) -> IO()`: Recebe uma função, digamos f, como argumento, e passa toda a entrada como uma lista de caracteres para f, sendo a cadeia retornada como resultado da aplicação de f escrita no dispositivo de saída padrão (a entrada é lida de fato à medida que os dados são usados por f).
- **Funções de escrita no dispositivo de saída padrão**
  - `putChar:: Char -> IO()`: Escreve o argumento (um caractere).
  - `putStr:: String -> IO()`: Escreve o argumento (uma lista de caracteres).
  - `putStrLn:: String -> IO()`: Escreve o argumento seguido de um caractere de terminação de linha.
  - `print:: Show a => a -> IO()`: Escreve o resultado de converter o argumento em uma lista de caracteres, usando a função `show`, e adicionar no final um caractere de terminação de linha.

3.4.1 Referências na mônada *IO*

O módulo `Data.IORef` contém definições para suporte ao uso de referências (variáveis no sentido de programação imperativa, i.e. que armazenam um valor que pode ser modificado durante a execução de um programa) em Haskell, na mônada `IO`.

As funções definidas no módulo `Data.IORef` são:

- `newIORef:: a -> IO (IORef a)`: Recebe um valor e retorna uma referência para esse valor.
- `readIORef:: IORef a -> IO a`: Recebe uma referência e retorna o valor apontado pela referência.
- `writeIORef:: IORef a -> a -> IO()`: Recebe uma referência e um valor e retorna uma ação que armazena o valor na referência.
- `modifyIORef:: IORef a -> (a->a) -> IO()`: Recebe uma referência e uma função f e retorna uma ação que armazena o valor resultante de aplicar f ao valor corrente armazenado na referência. Ou seja, modifica o valor v armazenado na referência pelo valor dado por $f\ v$.

3.5 Mônadas para estruturamento de código

Mônadas permitem estruturar programas de modo a colocar o controle sobre como sequenciar o processamento de valores monádicos internamente à mônada, mais especificamente na definição do combinador `(>>=)`.

Essa capacidade pode ser entendida como uma forma de permitir que "`;`"s sejam programados (sendo "`;`" o delimitador usado comumente em linguagens imperativas para indicar o sequenciamento de comandos). Uma metáfora também usada para expressar o comportamento do combinador monádico `(>>=)` é o de uma esteira rolante em uma linha de montagem de uma fábrica, na qual o material trasportado de uma unidade para a seguinte pode ser controlado com filtragem e modificação.

É a propriedade de controlar a sequenciação de valores monádicos internamente na mônada (ou seja, internamente na definição do combinador `(>>=)`) que faz com que modificações possam ser feitas apenas localmente, em vez de nos usos dos valores monádicos ao longo de todo um programa.

3.5.1 Mônadas para evitar repetidos testes de erro

Podemos usar mônadas para evitar ter que testar condições de erro ou de ocorrência anormal. Por exemplo, o exemplo a seguir ilustra o caso de uma função com 3 parâmetros de tipo `Maybe Int` que retorna a soma do resultado dos valores contidos em seus parâmetros, se nenhum deles for `Nothing`, caso contrário retorna `Nothing`:

```
soma Nothing _ _ = Nothing
soma _ Nothing _ = Nothing
soma _ _ Nothing = Nothing
soma (Just a) (Just b) (Just c) = Just (a+b+c)
```

Podemos simplificar essa função, concentrando a verificação de se o valor é ou não é igual a `Nothing` no operador `>>=` da mônada `Maybe`:

```
soma ma mb mc = ma >>= \a ->
                mb >>= \b ->
                mc >>= \c -> return (a+b+c)
```

O tipo `Either` (`data Either a b = Left a | Right b`) é similar: em vez de apenas `Nothing` o tipo `Either` em geral usa `Left v`, que contém o valor `v` que permite indicar alguma informação sobre o erro (`Right r` sendo análogo a `Just r`). Por convenção, é comum usar `Left` para indicar erro e `Right` para indicar sucesso.

Usando as instâncias de `Monad` para `Maybe` e `Either a`, podemos fazer chamadas a soma como:

```
soma (Just 1)  (Just 2)       (Just 3)  (retorna Just 6),
soma (Just 1)  Nothing        (Just 3)  (retorna Nothing),
soma (Right 1) (Right 2)      (Right 3) (retorna Right 6),
soma (Right 1) (Left "erro") (Right 3) (retorna Left "erro").
```

3.5.2 Exercícios Resolvidos

1. Considere o tipo de expressões `Expr` e a função de avaliação de expressões `eval` definidos nos módulos `Expr` e `Eval0`, respectivamente, a seguir:

```
module Expr (Expr(..), Name) where

data Expr = Var Name       | Const Integer
          | Add Expr Expr | Fun Name Expr | App Expr Expr
  deriving (Show)

type Name = String
```

```
module Eval0 where

import Expr
import Data.Map                (Map,(!))
import qualified Data.Map as M (insert)

data Val = I Integer | F (Val->Val)
type Store  = Map Name Val

eval:: Expr -> Store -> Val
eval (Var x)     s = s!x
eval (Const i)   _ = I i
eval (Add e e')  s = addExpr e e' s
eval (Fun x e)   s = F (\v-> eval e (M.insert x v s))
eval (App e e')  s = app (eval e s) (eval e' s)

app:: Val -> Val -> Val
app (F f) v = f v

addE:: Expr -> Expr -> Store -> Val
addE e e' s  = addVal (eval e s) (eval e' s)

addVal (I i) (I i') = I (i+i')
```

Reescreva `eval` de modo a usar código monádico, usando mônada identidade (que apenas introduz construtor `Identity`, transformando valores do tipo `a` em valores `Identity a`).

Use `Control.Monad.Identity`.

Use expressões como `exp1` abaixo para teste:

```
exp1 = Const 1 `Add` App (Fun "x" (Const 2 `Add` Var "x")) (Const 3)
```

Solução:

```
module EvalM0 where

import Expr
import Data.Map                  (Map,(!))
import qualified Data.Map as M (insert,lookup)
import Control.Monad.Identity  (Identity)

data Val = I Integer | F (Val->Identity Val) deriving (Show)
type Store = Map Name Val

eval:: Expr -> Store -> Identity Val
eval (Var x)     s = s ! x
eval (Const i)   _ = return $ I i
eval (Add e e')  s = addE e e' s
eval (Fun x e)   s = return $ F (\v-> eval e (M.insert x v s))
eval (App e e')  s = app e e' s

app:: Expr -> Expr -> Store -> Identity Val
app e e' s = eval e s >>= \(F f)-> eval e' s >>= \v-> f v

addE:: Expr -> Expr -> Store -> Identity Val
addE e e' s = eval e s >>= \(I i)-> eval e' s >>= \(I i')->
              return $ I (i+i')
```

2. Reescreva o código monádico acima para introduzir exceções (com indicação do erro ocorrido) que podem ser tratadas com `catchError` (de `Control.Monad.Except`).

   Não vamos usar transformadores de mônadas. O leitor interessado pode consultar, por exemplo, [1, 3] e [4, Cap. 18].

   Solução:

```
{-# LANGUAGE FlexibleContexts #-}
module EvalMExcept where

import Expr
import Data.Map (Map)
import qualified Data.Map as Map (insert,lookup)
import Control.Monad.Except      (throwError)

data Val   = I Integer | F (Val->M Val)
type Store = Map Name Val

instance Show Val where
  show (I i) = show i
  show (F f) = "function "

data EvalError = UnboundVar Name
               | IntExpectedButFound String
               | FunExpected

instance Show EvalError where
  show (UnboundVar x)          = "unbound var.: " ++ x
  show (IntExpectedButFound s) = "Integer expected but found: " ++ s
  show (FunExpected)           = "Functional value expected"

type M a = Either EvalError a

eval:: Expr -> Store -> M Val
eval (Var x)     s = s!x
eval (Const i)   _ = return $ I i
eval (Add e e')  s = evalBinOp add    e e' s
eval (App e e')  s = evalFun e e' s

s ! x = maybe (throwError (UnboundVar x)) return (Map.lookup x s)

evalBinOp:: (Val->Val->M Val) -> Expr -> Expr -> Store -> M Val
evalBinOp op e e' s =
  eval e s >>= \v -> eval e' s >>= \v' ->
    case (v, v') of
      (I _, I _) -> op v v'
      (v  , I _) -> throwError (IntExpectedButFound $ show e)
      (_  , v  ) -> throwError (IntExpectedButFound $ show e')

evalFun:: Expr -> Expr -> Store -> M Val
evalFun e e' s = eval e s >>= \ff -> eval e' s >>= \v ->
                 case ff of
                   F f -> f v
                   _   -> throwError FunExpected

add (I i) (I i') = return $ I (i+i')
```

3.5.3 Exercícios

1. Reescreva o módulo `EvalMExcept` da sub-seção anterior para incluir subtrações, multiplicações e divisões. Inclua uma alternativa na definição de `EvalError` para tratar divisões por zero. Inclua código para causar exceções por divisão por zero.

2. Reescreva o módulo `EvalMExcept` do exercício acima para incluir expressões condicionais (com condição booleana e expressões para cada caso correspondente a avaliação da condição retornar resultado verdadeiro ou falso, respectivamente.

   Inclua código para causar exceções de erros de tipo devido a tipo incorreto de expressões (já que agora expressões podem ser também booleanas).

   Argumente porque, em Haskell, ao contrário de outras linguagens de programação, incluir expressões condicionais como neste exercício é desnecessário: tem diferença apenas sintática em relação a chamada de função **`if - then - else`**. Defina tal função **`if - then - else`**.

Referências

[1] Martin Grabmüller. Monad Transformers Step by Step. `http://catamorph.de/publications/2004-10-01-monad-transformers.html`, October 2006.

[2] Simon Peyton Jones et al. GHC — The Glasgow Haskell Compiler. http://www.haskell.org/ghc/, 1998.

[3] Jeff Newbern. Haskell/Monad transformers, (usado em) Junho de 2017. `https://en.wikibooks.org/wiki/Haskell/Monad_transformers`.

[4] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly Media, Inc., 2008.