Monads

Lucas Albertins de Lima

Departamento de Computação - UFRPE

O tipo `Maybe`

```haskell
data Maybe t = Just t | Nothing
```

Exemplo

```haskell
safeDiv :: Integral t => t -> t -> Maybe t
safeDiv x y
  | y /= 0 = Just (x `div` y)
  | otherwise = Nothing
```

Usando safeDiv

```
Prelude> safeDiv 10 5
Just 2
Prelude> safeDiv 10 3
Just 3
Prelude> safeDiv 10 0
Nothing
```

E se quiséssemos fazer assim?

```
Prelude> safeDiv 100 (safeDiv 20 7)
?????
```

E se quiséssemos fazer assim?

```
Prelude> safeDiv 100 (safeDiv 20 7)

<interactive>:98:1:
    No instance for (Integral (Maybe a0))
      arising from a use of `safeDiv'
    Possible fix: add an instance declaration for (Integral (Maybe a0))
    In the expression: safeDiv 100 (safeDiv 20 7)
    In an equation for `it': it = safeDiv 100 (safeDiv 20 7)

<interactive>:98:9:
    No instance for (Num (Maybe a0)) arising from the literal `100'
    Possible fix: add an instance declaration for (Num (Maybe a0))
    In the first argument of `safeDiv', namely `100'
    In the expression: safeDiv 100 (safeDiv 20 7)
    In an equation for `it': it = safeDiv 100 (safeDiv 20 7)

<interactive>:98:14:
    No instance for (Integral a0) arising from a use of `safeDiv'
    The type variable `a0' is ambiguous
    Possible fix: add a type signature that fixes these type variable(s)
    Note: there are several potential instances:
      instance Integral Int -- Defined in `GHC.Real'
      instance Integral Integer -- Defined in `GHC.Real'
      instance Integral GHC.Types.Word -- Defined in `GHC.Real'
      ...plus 8 others
    In the second argument of `safeDiv', namely `(safeDiv 20 7)'
    In the expression: safeDiv 100 (safeDiv 20 7)
    In an equation for `it': it = safeDiv 100 (safeDiv 20 7)

<interactive>:98:22:
    No instance for (Num a0) arising from the literal `20'
    The type variable `a0' is ambiguous
    Possible fix: add a type signature that fixes these type variable(s)
    Note: there are several potential instances:
      instance Num Double -- Defined in `GHC.Float'
      instance Num Float -- Defined in `GHC.Float'
      instance Integral a => Num (GHC.Real.Ratio a)
        -- Defined in `GHC.Real'
      ...plus 11 others
    In the first argument of `safeDiv', namely `20'
    In the second argument of `safeDiv', namely `(safeDiv 20 7)'
    In the expression: safeDiv 100 (safeDiv 20 7)
```

Como resolver?

- Seria necessário mudar o tipo de safeDiv

```
safeDiv :: Integral t =>
           Maybe t -> Maybe t -> Maybe t
```

- ... e a implementação também:

```
safeDiv _ Nothing = Nothing
safeDiv Nothing _ = Nothing
safeDiv (Just x) (Just y)
  | y /= 0 = Just (x `div` y)
  | otherwise = Nothing
```

Quai são as características de safeDiv?

- Funções que podem produzir **Nada**
- Necessário checar resultado anterior:
  - safeDiv :: Maybe t -> Maybe t -> Maybe t
    Se **argumento** é Nothing então devolver Nothing Senão, se o divisor for zero, devolver Nothing Senão, devolver o quociente em um Just
  - Queremos encadear computações
    - Ex.: safeDiv 100 (safeDiv 20 7)
    - Com menos trabalho!
    - Levando contexto em consideração

Maybe é uma Monad

- Computação que leva em conta **contexto**
- Que pode ser **encadeada** com outras
- Através de um operador de **ligação**: >>=

Exemplos:

- `Just 3 >>= (\x-> Just $ show x ++ "!")`
- `Just 3 >>= (\x->Just $ show x ++ "!") >>= (\y->Just $ map (\z->[z]) y)`
- `Nothing >>= (\x->Just $ show x ++ "!") >>= (\y->Just $ map (\z->[z]) y)`
- `Just 33 >>= \x -> safeDiv 10000 x >>= (\y->Just $ show y)`

Contexto mínimo: return

```
Just 33 >>= \x -> safeDiv 10000 x >>= (\y->Just $ show y)
==
return 33 >>= \x -> safeDiv 10000 x >>= (\y->return $ show y)
```

- **return** é apenas uma função
- **Não é** um desvio de fluxo de controle

Componentes de uma monad

- Construtor de tipo M
  - Ex.: Maybe
- Função unidade:
  - `return :: Monad m => a -> m a`
- Função de encadeamento (ou ligação):
  - `(>>=) :: Monad m => m a -> (a -> m b) -> m b`

Maybe é uma instância de Monad

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

Notação do

```
return 33 >>= \x -> safeDiv 10000 x
>>= (\y->return $ show y) >>= safeTail
```

equivale a

```
do
  x <- return 33
  y <- safeDiv 10000 x
  z <- return $ show y
  safeTail z
```

Notação do

```
return 33 >>= \x -> safeDiv 10000 x
>>= (\y->return $ show y) >>= safeTail
```

equivale a

```
do {
  x <- return 33;
  y <- safeDiv 10000 x;
  z <- return $ show y;
  safeTail z;
}
```

O tipo IO t

- Todo tipo IO a é uma ação de I/O.
- Exemplo - imprimir uma String:

```
writefoo :: IO ()
writefoo = putStrLn "foo"
```

- Somente digitar isso não produz nada, mas ao chamar writefoo, "foo"é impresso.

Exemplo de instâncias da classe Visible

```
instance Visible Char where
  toString ch = [ch]
  size _ = 1

instance Visible Bool where
  toString True  = "True"
  toString False = "False"
  size _ = 1
```

Haskell e efeitos colaterais

- Mas o fato de Haskell ser *puramente funcional* não impediria efeitos colaterais?
  - Por exemplo, entrada e saída.
- Quase!

O tipo IO t

- O tipo IO t é da classe Monad.
- Imagine tais tipos como programas que executam entrada/saída (IO) e retornam um valor do tipo t
- Lê uma linha do teclado
  `getLine :: IO String`
- Escreve uma String na tela.
  `putStr :: String -> IO ()`
  o resultado desta interação tem tipo ()
  - uma tupla vazia.
  - não retorna nenhum resultado interessante, apenas faz I/O.

Sequenciando ações de IO

- A operação if−then

```
putNtimes :: Int -> String -> IO ()
putNtimes n str
  = if n <= 1
      then putStr str
    else do putStr str
      putNtimes (n-1) str
```

Exemplo

```
main :: IO()
main = putStr "Digite␣seu␣nome:" >>
       getLine >>=
       \st -> putStr "Ao␣contrario␣e':" >>
       putStr (reverse st)
```

Exemplo

```
main :: IO()
main = do putStr "Digite␣seu␣nome:"
          st <- getLine
          putStr "Ao␣contrario␣e':"
          putStr (reverse st)
```

Resumo de funções para IO t

```
getLine :: IO String
getChar :: IO Char
putStr :: String -> IO ()
putStrLn :: String -> IO ()
(>>=) :: IO a -> (a -> IO b) -> IO b
(>>) :: IO a -> IO b -> IO b
return :: a -> IO a
```

Manipulação de arquivos

- Haskell manipula arquivos através do tipo FilePath, um tipo sinônimo.
  `type FilePath = String`
- Leitura de arquivos:
  `readFile :: FilePath -> IO String`
- Escrever em arquivos:
  `writeFile :: FilePath -> String -> IO()`
- Anexar a arquivos:
  `appendFile::FilePath -> String -> IO()`

Exemplo

```haskell
main :: IO ()
main =
  putStrLn "Escrevendo" >>
  writeFile "a.txt" "Hello\nworld" >>
  appendFile "a.txt" "\nof\nHaskell" >>
  putStrLn "Lendo o arquivo" >>
  readFile "a.txt" >>=
  \x -> putStrLn x
```

Materiais de Estudo

- Monads
  - http://learnyouahaskell.com/functors-applicative-functors-and-monoids
  - http://learnyouahaskell.com/a-fistful-of-monads
  - http://www.haskell.org/haskellwiki/Monad
- IO
  - http://lambda.haskell.org/hp-tmp/docs/2011.2.0.0/ghcdoc/libraries/haskell2010-1.0.0.0/System-IO.html

Exercício

- Defina um tipo polimórfico Fila e as seguintes funções para operar sobre filas:
  - `criarFila :: Int -> t -> Either String (t, Fila t)`
    - O primeiro parâmetro corresponde à capacidade máxima da fila
    - O segundo ao primeiro elemento da fila
    - O resultado consiste em um valor (que pode ser nenhum) e a fila "modificada"
  - `push :: t -> Fila t -> Either String (t, Fila t)`
  - `pop :: Fila t -> Either String (t, Fila t)`
  - `peek :: Fila t -> Either String (t, Fila t)`
- As operações devem produzir uma mensagem de erro em situações como fila cheia (em um push), vazia (em um pop ou peek) ou com capacidade menor que 1 (no criarFila)

Bibliografia

Slides elaborados a partir de originais por André Santos e Fernando Castor

[1] Simon Thompson.
Haskell: the craft of functional programming.
Addison-Wesley, terceira edition, Julho 2011.