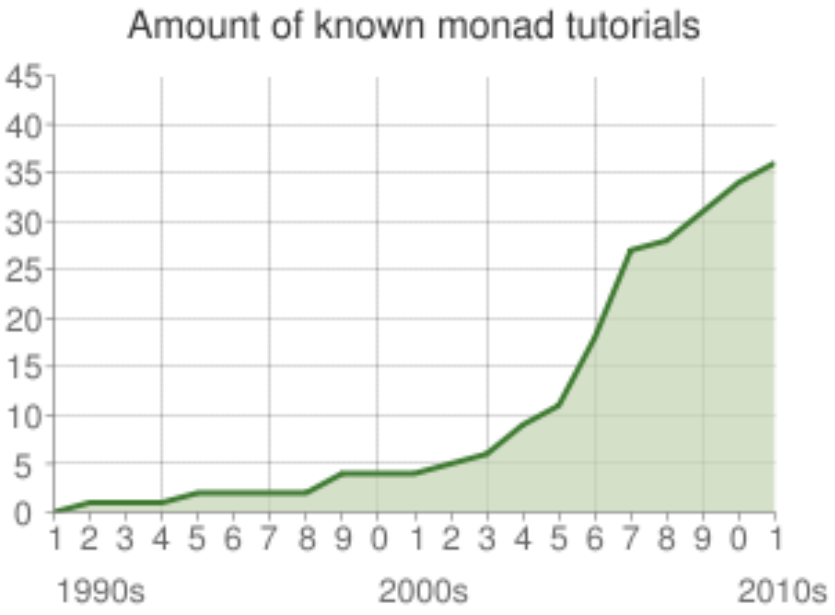
Monady w programowaniu funkcyjnym

Łukasz Dąbek

13 października 2013

Samouczki


Amount of known monad tutorials
45
40
35
30
25
20
15
10
5
0
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
1990s
2000s
2010s

Zwięzła definicja

A monad is just a monoid in the category of endofuctors, what's the problem?

— *Philip Wadler*

Funkcje debugowalne

- Debugowanie w języku nieczystym jest proste (`printf` w dowolnym miejscu).

Funkcje debugowalne

- Debugowanie w języku nieczystym jest proste (`printf` w dowolnym miejscu).
- Jak poradzić sobie w Haskellu?

Funkcje debugowalne

```
f, g :: Float -> Float
```

Funkcje debugowalne

```
f, g :: Float -> Float
f x = x + 4
g x = x * 2
(f . g) 19 = 42
```

Funkcje debugowalne

```
f', g' :: Float -> (Float, String)
```

Funkcje debugowalne

```
f', g' :: Float -> (Float, String)
f' x = (x + 4, "f was called")
g' x = (x * 2, "g " ++ show x)
g' . f' -- błąd typów!
```

Kompozycja

- Chcemy prześledzić wykonanie złożenia funkcji `f'` i `g'`.
- Złożenie `f` i `g` to `f . g`.
- Jak złożyć `f'` i `g'`?

Kompozycja

```
let (y, s) = g' x
    (z, t) = f' y in (z, s ++ t)
```

Kompozycja

- Nie chcemy pisać tego kodu wielokrotnie.
- `bind f' :: (Float,String) -> (Float,String)`

Bind

```
bind :: (Float -> (Float,String))
     -> (Float,String)
     -> (Float,String)
```

Bind

```
bind f (x,s) =
  let (y,t) = f x in (y,s ++ t)

comp = bind f . g
```

Unit

```
unit :: Float -> (Float,String)
unit x = (x,"")
```

Unit - przykład dowodu

```
bind unit (a, s)
= bind (\x -> (x,"")) (a,s)
= let (y,t) = (\x -> (x,"")) in (y,s++t)
= (a, s ++ "")
= (a, s)
```

Funkcje niedebugowalne

- Jak składać funkcje debugowalne ze zwykłymi?
- `f :: Float -> Float`
- `lift f :: Float -> (Float,String)`

Funkcje niedebugowalne

- Jak składać funkcje debugowalne ze zwykłymi?
- `f :: Float -> Float`
- `lift f :: Float -> (Float,String)`
- `lift f = unit . f`

Małe podsumowanie

- `type Dbg a = (a,String)`
- `f, g::Float -> Float`
- `f', g'::Float -> Dbg Float`
- `bind::(a -> Dbg b) -> Dbg a -> Dbg b`
- `unit::a -> Dbg a`

Funkcje wielowartościowe

Pierwiastek kwadratowy (sześcienny) z dodatniej liczby rzeczywistej jest funkcją jednowartościową.

Funkcje wielowartościowe

```
sqrt, cbrt :: Float -> Float
```

Funkcje wielowartościowe

```
sqrt, cbrt :: Float -> Float
sixthRoot = sqrt . cbrt
```

Funkcje wielowartościowe

W przypadku liczb zespolonych każda niezerowa liczba ma n pierwiastków n-tego stopnia.

Funkcje wielowartościowe

```
sqrt', cbrt' :: Complex -> [Complex]
```

Kompozycja

- Chcemy złożyć funkcje `sqrt'` i `cbrt'`.
- Problem: `sqrt'` nie przyjmuje listy!

Kompozycja

```
sixthRoot x = concatMap sqrt' (cbrt' x)
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap = concat . map
```

Bind

- `bind`$_d$`::(a -> Dbg b) -> Dbg a -> Dbg b`
- `concatMap::(a -> [b]) -> [a] -> [b]`
- `bind`$_m$ `= concatMap`

Monada listowa

- `type Mv a = [a]`
- $\texttt{bind}_m$ `= concatMap`
- $\texttt{unit}_m$ `x = [x]`

Monady ogólnie

Widzieliśmy przykłady, teraz będzie bardziej abstrakcyjnie. Jednak najpierw - funktory.

Funktory

- Typ danych będący „polimorficznym kontenerem”.
- Przykładowo: `Maybe`, `List`, `Either`.

Funktory

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

Funktory - prawa

```
fmap id = id
fmap (f . g) = fmap f . fmap g
```

Lista jako funktor

```
instance Functor ([]) where
    fmap = map
```

Maybe jako funktor

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

Para jako funktor

```
instance Functor (a,) where
    fmap f (a,x) = (a,f x)
```

Either jako funktor

```
instance Functor (Either a) where
    fmap f (Left a) = Left a
    fmap f (Right b) = Right (f b)
```

Dwa nietypowe funktory

```
data Identity a = I a
data Const x a = C x

instance Functor Identity where
    fmap f (I a) = I (f a)

instance Functor (Const a) where
    fmap _ (Const a) = Const a
```

Z czego składa się monada?

- Typu `m` będącego funktorem.
- Funkcji `unit :: a -> m a`.
- Funkcji
  `bind :: (a -> m b) -> m a -> m b`.

Z czego składa się monada?

- Typu `m` będącego funktorem.
- Funkcji `unit :: a -> m a`.
- Funkcji
  `bind :: (a -> m b) -> m a -> m b`.
- (W Haskellu nazywają się one trochę inaczej).

Klasa Monad w Haskellu

```
class Monad m where
    -- unit zwany return'em
    return :: a -> m a

    -- bind zwany (>>=)
    (>>=) :: m a -> (a -> m b) -> m b
```

Writer Monad

```
data Writer a = (a, String)
instance Monad Writer where
    return x = (x,"")
    (x,s) >>= f = let (y,t) = f x
                  in (y,s++t)
```

List Monad

```
instance Monad [] where
    return x = [x]
    (>>=) = flip concatMap
```

List Monad

```
solutions = [1..5] >>= \x ->
            [1..5] >>= \y ->
            if x * y `mod` 3 == 0
              then return (x,y)
              else []
```

List Monad

```
solutions = do
    x <- [1..5]
    y <- [1..5]
    if x * y `mod` 3 == 0
      then return (x,y)
      else []
```

Maybe Monad

```
data Maybe a = Just a | Nothing
instance Monad Maybe where
    return = Just
    Just a  >>= f = f a
    Nothing >>= _ = Nothing
```

Maybe Monad

```
father :: Person -> Maybe Person
grandgrandfather x =
  case father x of
    Nothing -> Nothing
    Just f1 -> case father f1 of
      Nothing -> Nothing
      Just f2 -> ...
```

Maybe Monad

```
grandgrandfather x =
  father x  >>= \f1 ->
  father f1 >>= \f2 ->
  father f2 >>= return
```

Maybe Monad

```
grandgrandfather x = do
  f1 <- father x
  f2 <- father f1
  father f2
```

Maybe Monad

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

f >=> g = \a -> f a >>= g

grandgrandfather =
    father >=> father >=> father
```

Maybe Monad

```
nFather :: Int -> Person -> Maybe Person
nFather n = foldr1 (>=>) (repeat n father)
```

Monoid w kategorii...

- `>=>` składa funkcje monadyczne.
- Można zdefiniować go w terminach `>>=`.
- `f >=> g = \x -> f x >>= g`

Prawa monadyczne

- `return >=> f = f`
- `f >=> return = f`
- `a >=> (b >=> c) = (a >=> b) >=> c`

Prawa monadyczne

- `return >=> f = f`
- `f >=> return = f`
- `a >=> (b >=> c) = (a >=> b) >=> c`
- Monoid! Operacja łączna z elementem neutralnym.
- (`g :: Functor f => a -> f b` nazywamy endofunktorem)

Prawa monadyczne inaczej

- `m >>= return = m`
- `return x >>= f = f x`
- `(a >>= b) >>= c =`
  `a >>= (\x -> b x >>= c)`

Więcej przykładów

Obejrzymy jeszcze kilka przykładów monad i typowych zastosowań.
Następnie opowiemy o zastosowaniach mniej typowych.

Either Monad

```
data Either a b = Left a | Right b
instance Monad (Either a) where
    return = Right
    Left a  >>= _ = Left a
    Right b >>= f = f b
```

Either Monad

```
catch :: Either a b
      -> (b -> Either a b)
      -> Either a b

catch (Left a) handler = handler a
catch (Right b) _ = Right b
```

Either Monad

```
integrate :: Either String Float
solution :: Either String String

solution =
  (integrate >>=
    (\f -> return $ "Result: " ++ show f))
  `catch`
  (\s -> return $ "Error: " ++ s)
```

State Monad

```
data State s a = State (s -> (a,s))

instance Monad (State s) where
    return x = \s -> (x,s)
    (State f) >>= g = \s ->
      let (a,s')   = f s
          State g' = g a
      in g' s'
```

State Monad

```
runState (State f) s = f s
get :: State s s
get = State $ \s -> (s,s)

set :: s -> State s ()
set s = State $ \_ -> ((),s)

inc :: State Int ()
inc = do
    s <- get
    set (s + 1)
```

State Monad

```
getRandom :: State Seed Int
getRandom = do
    seed <- get
    let (n, seed') = fromSeed seed
    set seed'
    return n
```

State Monad

```
getOne :: [a] -> State Seed a
getOne xs = do
    rand <- getRandom
    return xs !! (rand `mod` length xs)
```

State Monad

```
example :: State Seed Int
example = do
    a <- getOne [0..10]
    b <- getOne [0..10]
    c <- getOne [0..10]
    return (a + b + c)

runState example initialSeed
```

State Monad

```
nCoins :: Int -> State Seed Int
nCoins n = fmap sum $ sequence
                (replicate n (getOne [0,1]))

sequence :: Monad m => [m a] -> m [a]
sequence [] = []
sequence (m:ms) = do
    x <- m
    xs <- sequence ms
    return (x:xs)
```

Podstawowe monady - podsumowanie

- Monady to „design pattern” programowania funkcyjnego.
- Pozwalają operować na jednym z elementów pudełka (funktora) bez znajomości jego struktury (>>=).
- Pozwalają na modelowanie efektów w czystym języku funkcyjnym (np. stanu).

Tablica

Reszta klasycznie - na tablicy.
W międzyczasie: pytania?

NULL

To juz koniec. Dziękuję.