Programowanie funkcyjne
(wykład 6.)

Roman Dębski

Instytut Informatyki, AGH

14 grudnia 2021

AGH

Plan wykładu

1. Monada jako wzorzec projektowy/obliczeniowy (vs. flatMappable)
2. Monady: kilka przykładowych instancji

Plan wykładu

1. Monada jako wzorzec projektowy/obliczeniowy (vs. flatMappable)
2. Monady: kilka przykładowych instancji

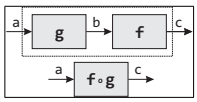
Składanie funkcji "zwykłych" vs. "rozszerzonych" [extended]

```haskell
incInt :: Int -> Int
incInt i = i + 1

ghci> incInt (incInt 1)   -- 3
ghci> incInt . incInt $ 1 -- 3
```

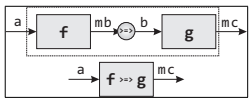
```haskell
incIntBox :: Int -> Box Int
incIntBox x = MkBox (x + 1)

safeTail :: [a] -> Maybe [a]
safeTail []     = Nothing
safeTail (x:xs) = Just xs
```

```haskell
ghci> incIntBox 1 -- MkBox 2
ghci> safeTail [1..5] -- Just [2,3,4,5]
```

```haskell
ghci> incIntBox . incIntBox $ 1 -- !
ghci> safeTail . safeTail $ [1..5] -- !
ghci> safeTail [1..5] >>= safeTail -- OK
```



vs.



Przykład (motywacyjny :): "Maybe chaining"

sekwencja (warunkowych) obliczeń (*)

```haskell
...
case m_x of
  Nothing -> Nothing
  Just x  ->
    case f x of
      Nothing  -> Nothing
      Just res1 ->
        case g res1 of
          Nothing   -> Nothing
          Just res2 ->
            case h res2 of
              Nothing   -> Nothing
              Just res3 -> ...
```

lub w nieco innym stylu

```haskell
if (x != null) {
  if (y != null) {
    if (z != null) {
      // ...
    } else {
      // x, y, z = null
    }
  } else {
    // x, y = null
  }
} else {
  // x = null
}
```

obliczenie "monadyczne" - odpowiada (*) :)

```haskell
m_x >>= f >>= g >>= h
```


Pojęcie monady: różne [wybrane] perspektywy

*"If C is a category, a **monad** on C consists of an endofunctor $T : C \to C$ together with two natural transformations: $\eta : 1_C \to T$ (where 1_C denotes the identity functor on C) and $\mu : T^2 \to T$ (where T^2 is the functor $T \circ T$ from C to C)".*

— https://en.wikipedia.org/wiki/Monad_(category_theory)

*"[...] a **monad** in C is just a monoid in a category of endofunctors of C, with product $\times$ replaced by composition of endofunctors of C and $unit$ set by the identity endofunctor".**

— Saunders Mac Lane, Categories for the Working Mathematician

Uwaga: czy to są definicje?

*"**Monads** in Haskell can be thought of as composable computation descriptions".*

— https://wiki.haskell.org/Monad

*"[...] the monad **type constructor** defines a type of computation, the **return** function creates primitive values of that computation type and the **bind operator** combines computations of that type together to make more complex computations of that type"*

— https://wiki.haskell.org/All_About_Monads

* "if that confuses you, it might be helpful to see a Monad as a lax functor from a terminal bicategory" :)

Pojęcie monady: jeszcze jedno spojrzenie [kontekst programistyczny]

"A *monad* is created by defining a type constructor m and two operations, *bind* and *return* (where *return* is often also called *unit*):

- the unary *return* operation takes a value from a plain type and puts it into a container/context using the constructor, creating a monadic value: $m\ a$
- the binary *bind* operation takes as its arguments: a monadic value $m\ a$ and a function $(a \to m\ b)$ that can transform the value

  - the bind operator *'unwraps'* the plain value a embedded in its input monadic value $m\ a$, and feeds it to the function
  - the function then creates a new monadic value $m\ b$ that can be fed to the next bind operators composed in the pipeline

With these elements, the programmer composes a sequence of function calls (the *'pipeline'*) with several bind operators chained together in an expression. Each function call transforms its input plain type value, and the bind operator handles the returned monadic value, which is fed into the next step in the sequence. Between each pair of composed function calls, the bind operator can inject into the monadic value some additional information that is not accessible within the function, and pass it along. It can also exert finer control of the flow of execution, for example by calling the function only under some conditions, or executing the function calls in a particular order"

— https://en.wikipedia.org/wiki/Monad_(functional_programming)

Różne [potencjalne] możliwości definicji monady [jako klasy typu]

```haskell
class Monad m where
  return :: a -> m a
  (>=>) :: (a -> m b) -> (b -> m c) -> (a -> m c)
```

(Haskell)

```haskell
class Monad m where
  return :: a -> m a                -- inject
  (>>=) :: m a -> (a -> m b) -> m b -- bind/chain
```

```haskell
class Functor m => Monad m where
  return :: a -> m a
  join :: m (m a) -> m a
```

```haskell
f >=> g = \a -> let mb = f a in mb >>= g = \a -> f a >>= g
ma >>= f = join (fmap f ma)
join mma = mma >>= id
fmap f ma = ma >>= \a -> return (f a) = ma >>= (return . f)
```

Klasa (typu) *Monad*: aktualna definicja

```
class Applicative m => Monad m where
    -- Sequentially compose two actions, passing any value produced
    -- by the first as an argument to the second.
    (>>=) :: m a -> (a -> m b) -> m b

    -- Sequentially compose two actions, discarding any value produced
    -- by the first, like sequencing operators (such as the semicolon)
    -- in imperative languages.
    (>>) :: m a -> m b -> m b
    m >> k = m >>= \_ -> k

    -- Inject a value into the monadic type.
    return :: a -> m a
    return = pure

    -- Fail with a message.
    fail :: String -> m a -- *
    fail s = errorWithoutStackTrace s
```

* this operation is invoked on pattern-match failure in a do expression; *fail* will be moved (soon) to MonadFail

Monad laws

Wykorzystanie "złożenia monadycznego" (*Kleisli composition operator*)

```
return >=> g == g                    -- Left identity
f >=> return == f                    -- Right identity
(f >=> g) >=> h == f >=> (g >=> h)   -- Associativity
```

Wykorzystanie operatora *bind*

```
return a >>= f == f a                        -- Left identity
m >>= return == m                            -- Right identity
(m >>= f) >>= g == m >>= (\x -> f x >>= g)   -- Associativity
```

Złożenie "zwykłych" funkcji

```
id . f == f                -- Left identity
f . id == f                -- Right identity
(f . g) . h == f . (g . h) -- Associativity
```


Functor vs. applicative functor vs. monad

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

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

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

```
return :: a -> f a -- m <-> f
pure ::   a -> f a
fmap  ::    (a -> b) -> f a -> f b
(<*>) :: f (a -> b) -> f a -> f b
(=<<) :: (a -> f b) -> f a -> f b -- m <-> f
(>>=) :: m a -> (a -> m b) -> m b
```


Plan wykładu

1. Monada jako wzorzec projektowy/obliczeniowy (vs. flatMappable)
2. Monady: kilka przykładowych instancji

Maybe monad

```
data Maybe a = Nothing | Just a
```

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

```
instance Applicative Maybe where
  pure = Just
  Just f  <*> m  = fmap f m
  Nothing <*> _  = Nothing
  Just _   *> m2 = m2
  Nothing  *> _  = Nothing
```

```
join :: Maybe (Maybe a) -> Maybe a
join (Just (Just a)) = Just a
join (Just Nothing)  = Nothing
join Nothing         = Nothing
```

```
instance Monad Maybe where
  (Just x) >>= k = k x
  Nothing  >>= _ = Nothing
  (>>) = (*>)
  fail _ = Nothing
```


List monad

```
data [] a = [] | a : [a]
```

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

```
instance Applicative [] where
  pure x = [x]
  fs <*> xs = [f x | f <- fs, x <- xs]
  xs *> ys  = [y | _ <- xs, y <- ys]
```

```
join :: [[a]] -> [a]
join [] = []
join (xs:xss) = xs ++ join xss
```

```
instance Monad [] where
  xs >>= f = [y | x <- xs, y <- f x]
  (>>) = (*>)
  fail _ = []
```

list comprehension vs. monad

```
[(x,y) | x <- xs, y <- ys] ==
  do {x <- xs; y <- ys; return (x,y)}
```


Reader ((->) r) monad

```
instance Functor ((->) r) where
  fmap = (.)
```

```
instance Applicative ((->) a) where
  pure = const
  (<*>) f g x = f x (g x)
```

```
join :: (r -> (r -> a)) -> (r -> a)
join f = \x -> f x x
```

```
instance Monad ((->) r) where
  f >>= k = \x -> k (f x) x
```


Error (Either a) monad

```
data Either e a = Left e | Right a
```

```
instance Functor (Either e) where
  fmap _ (Left err) = Left err
  fmap g (Right x)  = Right (g x)
```

```
instance Applicative (Either e) where
  pure = Right
  Left  e <*> _ = Left e
  Right f <*> r = fmap f r
```

```
join :: Either e (Either e a) ->
        Either e a
join (Right (Right x))  = Right x
join (Right (Left err)) = Left err
join (Left err)         = Left err
```

```
instance Monad (Either e) where
  -- return = Right
  Right x  >>= mg = mg x
  Left err >>= _  = Left err
```

Bibliografia

- Eugenio Moggi, Notions of computation and monads, https://core.ac.uk/download/pdf/21173011.pdf
- Philip Wadler, Monads for Functional Programming, https://cse.sc.edu/~mgv/csce330f16/wadler_monadsForFP_95.pdf
- http://learnyouahaskell.com
- http://book.realworldhaskell.org/read
- https://wiki.haskell.org/Typeclassopedia
- https://wiki.haskell.org/All_About_Monads
- https://wiki.haskell.org/What_a_Monad_is_not
- https://wiki.haskell.org/Monads_as_computation
- https://www.quora.com/Functional-Programming/What-are-monads-in-functional-programming-and-why-are-they-useful