Funktory

Funktor to operacja `T :: * -> *` na typach
wraz z operacją `fmap` na funkcjach

```
fmap :: (a -> b) -> (T a -> T b)
```

zachowującą strukturę składania funkcji, czyli

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

Monady

Monada to konstruktor typów `M`, z operacjami

```
return :: a -> M a
(>>=)  :: M a -> (a ->  M b) -> M b
```

Elementami typu `M a` są obliczenia dające wynik typu `a` (z potencjalnymi efektami ubocznymi)

- `return x` to obliczenie czyste
- `>>=` sekwencjonuje obliczenie z jego kontynuacją, np.

  ```
  readChan stdin >>=   (\userInput -> ... )
  ```

Każda monada jest/powinna być funktorem. To, że Functor nie jest nadklasą Monad jest li tylko zaszłością.

Prawa monadyki

Każda monada musi spełniać następujące prawa:

1. `(return x) >>= k == k x`
2. `m >>= return == m`
3. `(m >>= f) >>= g == m >>= (\x -> (f x >>= g))`

Pierwsze dwa prawa mówią, że `return` nie ma efektów; jest elementem neutralnym dla `(>>=)`

Trzecie prawo mówi, że sekwencjonowanie obliczeń jest łączne, czyli w pewnym sensie, że

```
(o1;o2);o3 === o1;(o2;o3)
```

. . . i możemy je traktować jako sekwencję `o1;o2;o3`

Prawa monadyki, inaczej

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

1. return >=> g     = g
2. f >=> return     = f
3. (f >=> g) >=> h  = f >=> (g >=> h)
```

Inna prezentacja monad

```
class Functor m => Monad' m where
   pure  ::  a -> m a
-- fmap  :: (a -> b) -> m a -> m b
-- fmap g . pure === pure . g

  join :: m (m a) -> m a
-- join . fmap pure === id === join . pure
-- join . fmap join === join . join
```

gdzie ta ostatnia równość jest w typie `m(m(m a)) -> m a`

Trywialny funktor

http://blog.sigfpe.com/2007/04/trivial-monad.html (Dan Piponi, @sigfpe)

```
newtype W a = W a deriving Show

instance Functor W where
  -- fmap :: (a -> b) -> W a -> W b
```

```
  fmap f (W a) = W (f a)

class Pointed f where
  pure :: a -> f a

instance Pointed W where
  pure = W

a, b :: W Int
a = pure 1
b = fmap (+1) a
-- zapakowaną wartość możemy wielokrotnie zwiększać:
s = fmap (+1)
t = s(s(a))
```

Trywialna monada

```
f :: Int -> W Int
f x = W (x+1)
-- Jak zastosować f dwukrotnie?

bind :: (a -> W b) -> (W a -> W b)
bind f (W a) = f a

c = bind f (f 1)

instance Monad W where
  return = W
  (W x) >>= f = f x
```

Ćwiczenia

```
g :: Int -> W Int -> W Int  -- g x (W y) = W (x+y), ale bez rozpakowywania
g x wy = undefined

h :: W Int -> W Int -> W Int --h (W x) (W y) = W (x+y), bez rozpakowywania
h wx wy = undefined

-- Udowodnij, że W spełnia prawa monadyki

join :: W (W a) -> W a -- bez rozpakowywania, tylko return i bind
join wwa = undefined
```

Funktory par

```
-- Dla dowolnego c operacja \ a -> (a,c) jest funktorem:
first :: (a->b) -> (a,c) -> (b,c)
first f (a,c) = (f a, c)

-- podobnie \b -> (c,b)
second :: (b->d) -> (c,b) -> (c,d)
second f (c,b) = (c, f b)

(&&&) :: (a -> b) -> (a -> c) -> a -> (b,c)
f &&& g = \a -> (f a, g a)
-- first f = f &&& id
-- second f = id &&& f
```

Ale czy potrafimy napisać funkcję typu `forall a c.  a -> (a,c)` ?

```
p1 :: Monoid c => a -> (a,c)
p1 a = (a,mempty)
```

Monada stanu

```
type S = Int  -- przykładowo
type SM a = S -> (a,S)

-- Nie można napisać instance Functor SM ...
smap :: (a->b) -> (SM a -> SM b)
smap f t = first f . t -- \s -> first f (t s)

spure :: a -> SM a
spure a s = (a, s)
-- spure = (,)

sbind :: SM a -> (a -> SM b) -> SM b
sbind f k = \s -> let (a,s') = f s in k a s'

sjoin :: SM (SM a) -> SM a
-- sjoin :: (S -> (S -> (a,S),S)) -> S -> (a,S)
sjoin mma = \s -> let (ma,s') = mma s in ma s'

-- uncurry ($) :: (b -> c, b) -> c
sjoin' :: SM (SM a) -> SM a
-- sjoin' mma = \s -> let (ma, s') = mma s in ma s'
```

```
-- sjoin' mma = \s -> uncurry ($) (mma s)
sjoin' mma = uncurry ($) . mma
```

Monada State

Jesli chcemy zrobić porządną instancję `Monad` musimy opakować to wszystko w newtype:

```
newtype State s a = State { runState :: s -> (a, s) }

instance Functor (State s) where
    fmap f m = State $ \s -> let
        (a, s') = runState m s
        in (f a, s')

instance Monad (State s) where
    return a = State $ \s -> (a, s)
    m >>= k  = State $ \s -> let
        (a, s') = runState m s
        in runState (k a) s'
```

Stan a lenistwo

Możemy zapisać instancje Functor i Monad trochę inaczej:

```
instance Functor (State s) where
    fmap f m = State $ \s -> case runState m s of
                                  (a, s') -> (f a, s')
instance Monad (State s) where
    return a = State $ \s -> (a, s)
    m >>= k  = State $ \s -> case runState m s of
                                  (a, s') -> runState (k a) s'
```

Jaka jest różnica?

Control.Monad.State.Lazy

```
import Debug.Trace

f = \s ->
```

```
        let (x, s')  = doSomething s
            (y, s'') = doSomethingElse s'
        in (3, s'')

doSomething s = trace "doSomething" $ (0, s)
doSomethingElse s = trace "doSomethingElse" $ (3, s)

main = print (f 2)
```

```
$ runhaskell LazyTest.hs
doSomethingElse
doSomething
(3,2)
```

Control.Monad.State.Strict

```
import Debug.Trace

f = \s ->
        let (x, s')  = doSomething s
            (y, s'') = doSomethingElse s'
        in (3, s'')

doSomething s = trace "doSomething" $ (0, s)
doSomethingElse s = trace "doSomethingElse" $ (3, s)

main = print (f 2)
```

```
ben@sowa:~/Zajecia/Zpf/Slides/7$ runhaskell StrictTest.hs
doSomething
doSomethingElse
(3,2)
```

Zwykle kolejność obliczeń jest nam obojętna, ale np. w wypadku IO...

Czytelnik

Okrojona wersja stanu (stan sie nie zmienia):

```
type E = Int  -- na przykład
type RM a = E -> a
```

```
rmap :: (a->b) -> RM a -> RM b
rmap = (.)

rpure :: a -> RM a
rpure = const

rbind :: RM a -> (a -> RM b) -> RM b
-- (E -> a) -> (a -> E -> b) -> E -> b
rbind m k e = k (m e) e

rjoin :: RM (RM e) -> RM e
-- (E -> E -> a) -> (E -> a)
rjoin mm e = mm e e
```

Monada kontynuacji

```
type Cont r a = (a -> r) -> r

contra :: (a->b) -> (b->r) -> (a->r)
contra f g = g . f

cmap :: (a -> b) -> Cont r a -> Cont r b
--   :: (a -> b) -> ((a -> r) -> r) ->  (b -> r) -> r
cmap f m = \c -> m $ c . f -- \c -> m (contra f c)

cpure :: a -> Cont r a
cpure = flip ($) -- \a c -> c a

cbind :: Cont r a -> (a -> Cont r b) -> Cont r b
-- ((a->r)->r)) -> (a -> (b->r)->r)
cbind m k = \c -> m (\a -> k a c)
```

Jak zwykle w bibliotece jest to zapakowane w newtype, ale mamy funkcje

```
cont :: ((a->r)->r) -> Cont r a
runCont :: Cont r a -> (a->r)->r
```

Kontynuacje

```
import Control.Monad.Cont
```

```
ex1 :: Cont r Int
ex1 = do
  a <- return 1
  b <- return 10
  return (a+b)

-- test :: (forall r. (Show r) => Cont r Int) -> String
test ex = runCont ex show

> test ex1
"11"

-- cont :: ((a->r)->r) -> Cont r a
ex2 :: Cont r Int
ex2 = do
  a <- return 1
  b <- cont (\c -> c 10)
  return (a+b)

> test ex2
"11"
```

Brak wyniku - wyjątki

```
ex3 = do
   a <- return 1
   b <- cont (\c -> "escape")
   return $ a+b

> test ex3
"escape"
```

. . . czyli mamy wyjątki

```
escape :: r -> Cont r a
escape r = cont (const r)

ex3e = do
   a <- return 1
   b <- escape "escape"
   return $ a+b
```

Wiele wyników

```
ex4 = do
   a <- return 1
   b <- cont (\c -> c 10 ++ c 20)
   return $ a+b

> test ex4
"1121"
```

Hmm, to prawie jak monada list:

```
test5 = do
  a <- return 1
  b <- [10, 20]
  return $ a+b

> test5
[11,21]
```

Wiele wyników (2)

```
ex6 = do
  a <- return 1
  b <- Cont (\c -> c 10 ++ c 20)
  return $ a+b

test6 = runCont ex6 (\x -> [x])

> test6
[11,21]
```

Albo inaczej:

```
ex7 = do
   a <- return 1
   b <- cont (\c -> concat [c 10, c 20])
   return $ a+b

test7 = runCont ex7 (\x -> [x])

ex8 = do
```

```
  a <- return 1
  b <- cont (\c -> [10,20] >>= c)
  return $ a+b

test8 = runCont ex8 return
```

Bonus: trochę teorii kategorii

```
class Category (~>) where
  id  :: a ~> a
  (.) :: (b ~> c) -> (a ~> b) -> (a ~> c)

instance Category (->) where
  id x = x
  (f . g) x = f (g x)

class (Category (~>), Category (~~>))
      => Functor' f (~>) (~~>)  | f (~>) -> (~~>), f (~~>) -> (~>)  where
  fmap' :: (a ~> b) -> (f a ~~> f b)

class Category (~>) => Monad' m (~>) where
  return :: a ~> m a
  bind   :: (a ~> m b) -> (m a ~> m b)

-- 1. bind return = id
-- 2. bind f . return = f
-- 3. bind f . bind g = bind (bind g . f)
```

Komonady

```
type a :~> b = a -> b

class Functor m => Monad m where
  return :: a :~> m a
  bind   :: (a :~> m b) -> (m a :~> m b)

-- Komonada w kategorii C to monada w C^op:
class Functor w => Comonad w where
  extract :: w a :~> a
  extend :: (w b :~> a) -> (w b :~> w a)

(=>>) :: Comonad w => w b -> (w b -> a) -> w a
(=>>) = flip extend
```

Przykład

```
data Pointer i e = P i (Array i e) deriving Show

instance Ix i => Functor (Pointer i) where
   fmap f (P i a) = P i (fmap f a)

instance Ix i => Comonad (Pointer i) where
   extract (P i a) = a!i
   extend f (P i a) = P i $ listArray bds (fmap (f . flip P a) (range bds))
       where bds = bounds a

x = listArray (0,9) [0..9]
wrap i = if i<0 then i+10 else if i>9 then i-10 else i
blur (P i a) = let
       k = wrap (i-1)
       j = wrap (i+1)
   in 0.25*a!k + 0.5*a!i + 0.25*a!j

test1 = P 0 x =>> blur
x ==> f = f x
test2 = P 0 x ==> fmap (+1) =>> blur ==> fmap (*2) ==> fmap (^2)
```

Ciągi operacji na poszczególnych elementach tablicy moga byc wykonywane przez osobne wątki. Komonadyczne =>> wskazuje miejsca gdzie konieczna jest synchronizacja.

Inna prezentacja monad

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

-- fmap g . pure = pure . g
-- fmap g x = pure g <*> x

class Applicative m => Monad'' m where
  join :: m (m a) -> m a
```

O Applicative jeszcze będziemy mówić.