Programowanie 2009

Monady

9 kwietnia 2008

Monady

```
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
```

Aksjomaty monad:

$$
\begin{aligned}
\texttt{return}\ a \ \texttt{>>=}\ f &= f\ a \\
m\ \texttt{>>=}\ \texttt{return} &= m \\
(m\ \texttt{>>=}\ \lambda a \to n)\ \texttt{>>=}\ f &= m\ \texttt{>>=}\ \lambda a \to (n\ \texttt{>>=}\ f)
\end{aligned}
$$

Domyślne definicje:

```
m >> n = m >>= (λ _ → n)
fail = error
```

Preludium standardowe definiuje też:

```
(=<<) :: Monad m ⇒ (a → m b) → m a → m b
(=<<) = flip (>>=)

sequence :: Monad m ⇒ [m a] → m [a]
sequence = foldr mcons (return []) where
   mcons p q = p >>= λx → q >>= λy → return (x:y)

sequence_ :: Monad m ⇒ [m a] → m ()
sequence_ = foldr (>>) (return ())

mapM :: Monad m ⇒ (a → m b) → [a] → m [b]
mapM f as = sequence (map f as)

mapM_ :: Monad m ⇒ (a → m b) → [a] → m ()
mapM_ f as = sequence_ (map f as)
```

sequence bez tajemnic

$$
\begin{aligned}
\texttt{sequence}\ [m_1, m_2, \ldots, m_n] &= \texttt{do} \\
&\quad\ x_1 \leftarrow m_1 \\
&\quad\ x_2 \leftarrow m_2 \\
&\quad\quad \ldots \\
&\quad\ x_n \leftarrow m_n \\
&\quad\ \texttt{return}\ [x_1, x_2, \ldots, x_n]
\end{aligned}
$$

$$
\begin{aligned}
\texttt{sequence\_}\ [m_1, m_2, \ldots, m_n] &= \texttt{do} \\
&\quad\ m_1 \\
&\quad\ m_2 \\
&\quad\quad \ldots \\
&\quad\ m_n \\
&\quad\ \texttt{return ()}
\end{aligned}
$$

mapM dokładnie objaśniony

$$
\begin{aligned}
\texttt{mapM}\ f\ [x_1, x_2, \ldots, x_n] \quad &= \quad \texttt{do} \\
&\qquad y_1 \leftarrow f\ x_1 \\
&\qquad y_2 \leftarrow f\ x_2 \\
&\qquad\quad \ldots \\
&\qquad y_n \leftarrow f\ x_n \\
&\qquad \texttt{return}\ [y_1, y_2, \ldots, y_n]
\end{aligned}
$$

$$
\begin{aligned}
\texttt{mapM\_}\ f\ [x_1, x_2, \ldots, x_n] \quad &= \quad \texttt{do} \\
&\qquad f\ x_1 \\
&\qquad f\ x_2 \\
&\qquad\quad \ldots \\
&\qquad f\ x_n \\
&\qquad \texttt{return ()}
\end{aligned}
$$

Przykłady użycia sequence, mapM i mapM_:

```
f k xs = sequence $ take k (repeat xs)
```

```
putStrLn :: String → IO ()

mapM_ putStrLn ["Ala","ma","kota"]
```

```
openFile :: FilePath → IOMode → IO Handle

fhs :: IO [Handle]
fhs = mapM (flip openFile ReadMode)
           ["ala.txt","ma.txt","kota.txt"]
```

Inne ważne funkcje przetwarzające monady

```
join :: (Monad m) ⇒ m (m a) → m a
join = (>>= id)

liftM :: (Monad m) ⇒ (a → b) → (m a → m b)
liftM f = (>>= λ a → return$ f a)

liftM2 :: (Monad m) ⇒ (a → b → c) → (m a → m b → m c)
liftM2 f a b = do
   a' ← a
   b' ← b
   return$ f a' b'

ap :: (Monad m) ⇒ m (a → b) → m a → m b
ap = liftM2 ($)
```

„Instrukcje” wyboru:

```
when :: (Monad m) ⇒ Bool → m () → m ()
when p s = if p then s else return ()

unless :: (Monad m) ⇒ Bool → m () → m ()
unless p s = when (not p) s
```

Uogólnienia funkcji dla list

```
mapAndUnzipM :: (Monad m) =>
    (a -> m (b,c)) -> [a] -> m ([b], [c])
mapAndUnzipM f xs =
    sequence (map f xs) >>= return . unzip

zipWithM :: (Monad m) =>
    (a -> b -> m c) -> [a] -> [b] -> m [c]
zipWithM f xs ys =  sequence $ zipWith f xs ys

zipWithM_ :: (Monad m) =>
    (a -> b -> m c) -> [a] -> [b] -> m ()
zipWithM_ f xs ys =  sequence_ $ zipWith f xs ys
```

```
foldM :: (Monad m) ⇒ (a → b → m a) → a → [b] → m a
foldM f a [] =  return a
foldM f a (x:xs) = f a x >>= λ y → foldM f y xs

filterM :: Monad m ⇒ (a → m Bool) → [a] → m [a]
filterM p [] = return []
filterM p (x:xs) = do
   b  ← p x
   ys ← filterM p xs
   return (if b then (x:ys) else ys)
```

Monada + Monoid = MonadPlus

```
class Monad m ⇒ MonadPlus m where
  mzero :: m a
  mplus :: m a → m a → m a
```

Dodatkowe aksjomaty monad z plusem:

$$
\begin{aligned}
\mathtt{mzero}\ `\mathtt{mplus}`\ m &= m \\
m\ `\mathtt{mplus}`\ \mathtt{mzero} &= m \\
(m\ `\mathtt{mplus}`\ n)\ `\mathtt{mplus}`\ p &= m\ `\mathtt{mplus}`\ (n\ `\mathtt{mplus}`\ p) \\
\mathtt{mzero}\ \mathtt{>>=}\ f &= \mathtt{mzero} \\
m\ \mathtt{>>=}\ \lambda a \to \mathtt{mzero} &= \mathtt{mzero}
\end{aligned}
$$

`MonadPlus` przypomina więc algebraiczny *pierścień* (gdzie `mzero` gra rolę zera, `mplus` — dodawania, `return` — jedynki, zaś `(>>=)` — mnożenia).

Dla monad z plusem mamy też:

```
msum :: MonadPlus m => [m a] -> m a
msum xs = foldr mplus mzero xs

guard :: MonadPlus m => Bool -> m ()
guard p = if p then return () else mzero
```

Listy jako monady (z plusem)

```
instance Monad [] where
  (>>=) = flip concatMap
  return = (:[])
  fail _ = []

instance MonadPlus [] where
  mzero = []
  mplus = (++)
```

Listy mogą służyć do reprezentowania wyników obliczeń, które zwracają więcej niż jedną wartość:

```
type Generator = []
```

Funkcje standardowe w notacji monadycznej

```
enumFrom :: Num a ⇒ a → Generator a
enumFrom n = return n `mplus` enumFrom (n+1)

filter :: (a → Bool) → Generator a → Generator a
filter p g = do
   a ← g
   guard$ p a
   return a

map :: (a → b) → Generator a → Generator b
map f g = do
   a ← g
   return$ f a
```

Liczby pierwsze

```
primes :: [Integer]
primes = 2 : filter (λ n → all (λ p → n `mod` p ≠ 0)
                    (takeWhile (λ p → p*p ≤ n) primes))
                    (enumFrom 3)

primes :: [Integer]
primes = 2 : [ n | n ← [3..], all (λ p → n `mod` p ≠ 0)
                      (takeWhile (λ p → p*p ≤ n) primes) ]

primes :: Generator Integer
primes = return 2
   `mplus`
   do
      n ← enumFrom 3
      guard (all (λ p → n `mod` p ≠ 0)
             (takeWhile (λ p → p*p ≤ n) primes))
      return n
```

Podlisty

```
sublist :: [a] → [[a]]
sublist [] = [[]]
sublist (x:xs) = map (x:) yss ++ yss where
   yss = sublist xs
```

Bez śmiecenia:

```
sublist :: [a] → [[a]]
sublist [] = [[]]
sublist (x:xs) = foldr (λ ys → ((x:ys):)) yss yss where
   yss = sublist xs
```

Podlisty ładniej (choć śmiecąc)

```
sublist :: [a] → [[a]]
sublist [] = [[]]
sublist (x:xs) = [ ys | zs ← sublist xs, ys ← [zs, x:zs]]

sublist :: [a] → Generator [a]
sublist [] = return []
sublist (x:xs) = do
   zs ← sublist xs
   return zs `mplus` return (x:zs)
```

Permutacje przez wstawianie

```
permi :: Permut a
permi [] = return []
permi (x:xs) = do
  ys ← permi xs
  insert ys where
    insert [] = return [x]
    insert ys@(y:ys') =
      return (x:ys)
        `mplus` do
          zs ← insert ys'
          return (y:zs)
```

Permutacje przez wstawianie w notacji listowej

```
permi :: [a] → [[a]]
permi [] = [[]]
permi (x:xs) = [ zs | ys ← permi xs, zs ← insert ys ]
  where
    insert [] = [[x]]
    insert ys@(y:ys') =
      (x:ys) : [ y:zs | zs ← insert ys' ]
```

Permutacje przez wybieranie

```
perms :: Permut a
perms [] = return []
perms xs = do
   (x,xs') ← select xs
   ys ← perms xs'
   return (x:ys) where
      select [y] = return (y,[])
      select (y:ys) =
         return (y,ys)
            `mplus` do
               (z,zs) ← select ys
               return (z,y:zs)
```

Permutacje przez wybieranie w notacji listowej

```
perms :: [a] → [[a]]
perms [] = [[]]
perms xs = [ x:ys | (x,xs') ← select xs, ys ← perms xs' ]
  where
    select [x] = [(x,[])]
    select (x:xs) =
      (x,xs) : [ (y,x:ys) | (y,ys) ← select xs ]
```

Co to jest sortowanie?

```
monotone :: Ord a ⇒ [a] → Bool
monotone [] = True
monotone [_] = True
monotone (x:xs@(y:_)) = x ≤ y ∧ monotone xs

sort :: Ord a ⇒ Permut a → Permut a
sort perm xs = do
   ys ← perm xs
   guard (monotone ys)
   return ys

iSort', sSort' :: Ord a ⇒ Permut a
iSort' = sort permi
sSort' = sort perms
```

Wykładnicze!!!

Algorytmy kwadratowe

```
iSort :: Ord a ⇒ [a] → [a]
iSort [] = []
iSort (x:xs) = insert . iSort $ xs where
   insert [] = [x]
   insert ys@(y:ys')
      | x ≥ y = y : insert ys'
      | otherwise = x:ys

sSort :: Ord a ⇒ [a] → [a]
sSort [] = []
sSort xs = y : sSort ys where
   (y:ys) = select xs
   select ys@[_] = ys
   select (y:ys)
      | y<z = y:zs
      | otherwise = z:y:zs' where
       zs@(z:zs') = select ys
```

W teorii list możemy jednak udowodnić, że

$$\texttt{iSort}\ xs\ \ \texttt{`elem`}\ \ \texttt{iSort'}\ xs$$

$$\texttt{sSort}\ xs\ \ \texttt{`elem`}\ \ \texttt{sSort'}\ xs$$

dla każdej listy $xs :: \forall a.\mathrm{Ord}\ a \Rightarrow [a]$.

Monada IO

```
type IO a = --- ukryte przed programistą
instance Monad IO where --- ukryte przed programistą
putChar :: Char → IO ()
putStr :: String → IO ()
putStrLn :: String → IO ()
print :: Show a ⇒ a → IO ()
getChar :: IO Char
getLine :: IO String
getContents :: IO String
interact :: (String → String) → IO ()
type FilePath = String
readFile :: FilePath → IO String
writeFile :: FilePath → String → IO ()
appendFile :: FilePath → String → IO ()
```

Więcej w module IO.

Ewaluator wyrażeń z dzieleniem

Por.:
Philip Wadler, Monads for functional programming,
Marktoberdorf Summer School 1992,
Båstad Spring School 1995

Ewaluator klasycznie

```
data Expr = Const Integer | Expr :/: Expr

eval :: Expr → Integer
eval (Const n) = n
eval (e1 :/: e2) = eval e1 ‘div‘ eval e2

kalkulator :: Expr → String
kalkulator = show . eval
```

Z ochroną przed dzieleniem przez zero — wymaga przepisania całej funkcji eval

```
data Wynik a = Blad String | OK a

eval :: Expr → Wynik Integer
eval (Const n) = OK n
eval (e1 :/: e2) =
   case eval e1 of
      Blad s → Blad s
      OK n1 →
         case eval e2 of
            Blad s → Blad s
            OK n2 →
               if n2 == 0
                  then Blad "dzielenie przez zero"
                  else OK$ n1 ‘div‘ n2
```

Wypisywanie wyniku

```
kalkulator :: Expr → String
kalkulator e =
   case eval e of
      Blad s → s
      OK n → show n
```

Ze zliczaniem operacji dzielenia — również wymaga przepisania funkcji eval w całości

```
type Tick a = Int → (a, Int)

eval :: Expr → Tick Integer
eval (Const n) c = (n,c)
eval (e1 :/: e2) c =
   let
      (n1,c1) = eval e1 c
      (n2,c2) = eval e2 c1
   in
      (n1 `div` n2, c2+1)

kalkulator :: Expr → String
kalkulator e =
   "Wynik: " ++ show n ++ ", liczba dzielen: " ++ show c
      where (n,c) = eval e 0
```

Z ochroną przed dzieleniem przez zero i zliczaniem operacji dzielenia — oba rozszerzenia się przeplatają

```
eval :: Expr → Tick (Wynik Integer)
eval (Const n) c = (OK n, c)
eval (e1 :/: e2) c =
   case eval e1 c of
      (Blad s, c1) → (Blad s, c1)
      (OK n1, c1) →
         case eval e2 c1 of
            (Blad s, c2) → (Blad s, c2)
            (OK n2, c2) →
               if n2 == 0
                  then (Blad "dzielenie przez zero", c2)
                  else (OK$ n1 ‘div‘ n2, c2 + 1)
```

Wypisywanie wyniku

```
kalkulator :: Expr → String
kalkulator e =
   case eval e 0 of
      (Blad s, c) →
         s ++ " po wykonaniu " ++ show c ++ " dzielen"
      (OK n, c) →
         "Wynik: " ++ show n ++ ", liczba dzielen: "
            ++ show c
```

Monadycznie

```
eval :: Monad m ⇒ Expr → m Integer
eval (Const n) = return n
eval (e1 :/: e2) = do
   n1 ← eval e1
   n2 ← eval e2
   return$ n1 ‘div‘ n2
```

Każdy wie, że to cukier syntaktyczny dla definicji:

```
eval :: Monad m ⇒ Expr → m Integer
eval (Const n) = return n
eval (e1 :/: e2) =
   eval e1 >>= λ n1 →
      eval e2 >>= λ n2 →
         return$ n1 ‘div‘ n2
```

Aby odtworzyć oryginalne rozwiązanie wystarczy zdefiniować monadę identycznościową

```
data Id a = Id { unId :: a }
instance Monad Id where
   return = Id
   (Id a) >>= f = f a

kalkulator :: Expr → String
kalkulator = show . unId . eval
```

Monadycznie z ochroną przed dzieleniem przez zero

```
instance Monad Wynik where
   return = OK
   fail = Blad
   OK w >>= f = f w
   Blad s >>= _ = Blad s
```

Teraz przerobienie eval wymaga lokalnej zmiany w jednym miejscu:

```
eval :: Monad m ⇒ Expr → m Integer
eval (Const n) = return n
eval (e1 :/: e2) = do
   n1 ← eval e1
   n2 ← eval e2
   if n2 ≠ 0 -- lokalna zmiana tylko tutaj
      then return$ n1 `div` n2
      else fail "dzielenie przez zero"
```

Por. kalkulator — całkiem bez zmian!

```
kalkulator :: Expr → String
kalkulator e =
   case eval e of
      Blad s → s
      OK n → show n
```

Ze zliczaniem operacji dzielenia

Typ `Tick` owijamy w celofanik żeby można było określić na nim strukturę monady

```
newtype TTick a = TTick { unTTick :: Tick a }
instance Monad TTick where
   return a = TTick (λ c → (a,c))
   (TTick f) >>= g = TTick (λ c →
      let
         (a,c') = f c
         TTick h = g a
      in
         h c')

tick :: TTick ()
tick = TTick (λ c → ((),c+1))
```

Znów przerobienie eval wymaga dopisania jedynie operacji tick.

(Zawęża się też typ jedynie do monad, które można ticknąć).

```
eval :: Expr → TTick Integer
eval (Const n) = return n
eval (e1 :/: e2) = do
   n1 ← eval e1
   n2 ← eval e2
   tick  -- tu jedyna zmiana
   return$ n1 ‘div‘ n2

kalkulator :: Expr → String
kalkulator e =
   "Wynik: " ++ show n ++ ", liczba dzielen: " ++ show c
      where (n,c) = unTTick (eval e) 0
```

Z ochroną przed dzieleniem przez zero i zliczaniem operacji dzielenia

Wkładamy monadę w monadę...
Program główny dałoby się jeszcze napisać:

```
kalkulator :: Expr → String
kalkulator e =
   case unTTick (eval e) 0 of
      (Blad s, c) →
         s ++ " po wykonaniu " ++ show c ++ " dzielen"
      (OK n, c) →
         "Wynik: " ++ show n ++ ", liczba dzielen: "
            ++ show c
```

Jak napisać eval?

```
eval :: Expr → TTick (Wynik Integer)
eval (Const n) = return (return n)
eval (e1 :/: e2) = do
   w1 ← eval e1
   ???
```

Jak odwinąć jeden celofanik?

Trzeba zdefiniować jedną monadę obsługującą i wyjątki i licznik

```haskell
class Monad m => ExcMonad m where
   throw :: String -> m a

class Monad m => TickMonad m where
   tick :: m ()

data Expr = Const Integer | Expr :/: Expr
```

Monada m musi obsługiwać i wyjątki i licznik

```haskell
eval :: (ExcMonad m, TickMonad m) ⇒ Expr → m Integer
eval (Const n) = return n
eval (e1 :/: e2) = do
   n1 ← eval e1
   n2 ← eval e2
   if n2 ≠ 0
      then do
         tick
         return$ n1 ‘div‘ n2
      else throw "dzielenie przez zero"
```

Transformator monad

```
class Transformer t where
   promote :: Monad m ⇒ m a → t m a
```

Rodzaj typu `t` jest trzeciego rzędu!

```
t :: (* → *) → (* → *)
```

Monada wyjątków

```
data Wynik a = Blad String | OK a
newtype Exc m a = Exc (m (Wynik a))

instance Monad m ⇒ Monad (Exc m) where
   return a = Exc (return (OK a))
   (Exc m) >>= f = Exc (m >>= g) where
      g (Blad s) = return (Blad s)
      g (OK a) = case f a of Exc m → m

instance Monad m ⇒ ExcMonad (Exc m) where
   throw s = Exc (return (Blad s))

instance Transformer Exc where
   promote m = Exc (m >>= return . OK)
```

Monada liczników

```
newtype Tick m a = Tick (Int → m (a, Int))

instance Monad m ⇒ Monad (Tick m) where
   return a = Tick (λ st → return (a, st))
   (Tick f) >>= g = Tick (λst →
      do
         (a, st') ← f st
         let Tick h = g a
         h st')

instance Monad m ⇒ TickMonad (Tick m) where
   tick = Tick (λ st → return ((), st+1))

instance Transformer Tick where
   promote m = Tick (λ st → do
      a ← m
      return (a, st))
```

Monada identycznościowa

```
data Id a = Id { unId :: a }
instance Monad Id where
   return = Id
   (Id a) >>= f = f a
```

Składamy monady

```
instance ExcMonad m ⇒ ExcMonad (Tick m) where
   throw = promote . throw

evalTE :: Expr → Tick (Exc Id) Integer
evalTE = eval
```