Wykład 9

Typy kwalifikowane i klasy typów w Haskellu

Motywacja

Klasy Eq, Show, Ord, Enum, Num

Typy wyższego rzędu

Klasy Functor, Applicative i Monad

Monada stanu

Motywacja

- Jaki powinien być najogólniejszy typ (typ główny, ang. proncipal type) funkcji (+) ?
  - Int -> Int -> Int -- zbyt specyficzny
  - a -> a -> a -- zbyt ogólny
- Dodawanie powinno być zdefiniowane dla wszystkich typów numerycznych Num = {Int, Integer, Float, Double, itd.}.
- Zauważmy, że typ a -> a -> a
  jest skrótem dla ∀ a. a -> a -> a
- *Typy kwalifikowane (ang. qualified types)* pozwalają określić dziedzinę kwantyfikacji:
  ∀ a ∈ Num. a -> a -> a lub ∀ a. Num a => a -> a -> a,
  co w Haskellu jest zapisywane tak: Num a => a -> a -> a

Klasy typów

- “Num” w poprzednim przykładzie jest *klasą typów* (ang. type class). Nie należy jej mylić z konstruktorem typów ani z konstruktorem wartości!

- Klasy typów nie są klasami w sensie programowania obiektowego!

- “Num” jest predykatem opisującym pewną własność typów.

- Klasy typów umożliwiają kontrolowane wprowadzenie mechanizmu *przeciążenia* (ang. overloading) do języka Haskell.

- W standardowej bibliotece Haskella zdefiniowano wiele klas typów; użytkownik może też definiować swoje klasy.

Przykład: porównanie

- Równość nie jest zdefiniowana dla wszystkich typów. Nie można porównywać funkcji.

- Operator (==) w Haskellu ma typ Eq a => a -> a -> Bool. Na przykład:

```
42 == 42                ➔ True
`a` == `a`              ➔ True
`a` == 42               ➔ << błąd typu! >>
                          (różne typy)
(+1) == (\x->x+1)       ➔ << błąd typu! >>
                        (No instance for (Eq (Integer -> Integer)))
```

- Błędy typów są wykrywane w czasie kompilacji (statycznie).

Porównanie cd.

Klasa typów `Eq` jest zdefiniowana następująco:

```
class Eq a  where
        (==), (/=)          :: a -> a -> Bool
        x /= y               =  not (x == y)
        x == y               =  not (x /= y)
```

Ostatnie dwa wiersze opisują implementacje domyślne dla operatorów klasy. Dzięki temu użytkownik musi zdefiniować tylko jedną z metod.

Typ staje się egzemplarzem (instancją) klasy w wyniku deklaracji.

Porównanie cd.

Typy zdefiniowane przez użytkownika również mogą być instancjami klasy `Eq`.

```
data Kolor = Trefl | Karo | Kier | Pik

instance Eq Kolor where
  Trefl == Trefl = True
  Karo  == Karo  = True
  Kier  == Kier  = True
  Pik   == Pik   = True
  _     ==  _    = False
```

Dzięki implementacjom domyślnym dostępna jest też funkcja /=.

```
*Main> Trefl /= Karo

True
```

Porównanie cd.

Oto bardziej skomplikowany przykład.

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

instance Eq a => Eq (Tree a) where
    Leaf a1       == Leaf a2       =  a1 == a2
    Branch l1 r1 == Branch l2 r2 =  l1==l2 && r1==r2
    _             ==         _       =  False
```

Typ `a` wartości przechowywanych w liściach drzewa również musi należeć do klasy Eq.

```
x `mem` []     = False
x `mem` (y:ys) = x==y || x `mem` ys
```

Z powodu porównania `x==y` Haskell sam wydedukuje ograniczenie na typ.

```
*Main> :t mem
mem :: Eq a => a -> [a] -> Bool
```

Automatyczne tworzenie instancji klas

Haskell umożliwia automatyczne utworzenie instancji klasy `Eq` (i niektórych innych klas: `Ord`, `Enum`, `Read`, `Show`, `Bounded`). Należy w definicji typu danych użyć klauzuli `deriving`.

```
data Kolor = Trefl | Karo | Kier | Pik
           deriving Eq
data Tree a = Leaf a | Branch (Tree a) (Tree a)
            deriving Eq

*Main> Leaf 1 == Leaf 1
True
*Main> Leaf 1 == Leaf 2
False
```

Klasa Show

Haskell nie potrafi pokazać wartości zdefiniowanych typów.

```
*Main> Trefl


<interactive>:1:0: error:
    * No instance for (Show Kolor) arising from a use of `print'
    * In a stmt of an interactive GHCi command: print it
```

Typ `Kolor` musi być instancją klasy `Show`. To też można zrobić automatycznie.

```
data Kolor = Trefl | Karo | Kier | Pik
           deriving (Eq, Show)
data Tree a = Leaf a | Branch (Tree a) (Tree a)
            deriving (Eq, Show)
*Main> Leaf 5
Leaf 5
```

Klasa Ord

Chcielibyśmy zdefiniować porządek na kolorach i drzewach.

```
*Main> Leaf 4 < Leaf 6

<interactive>:1:0: error:
    * No instance for (Ord (Tree Integer)) arising from a use of `<'
    * In the expression: Leaf 4 < Leaf 6
      In an equation for `it': it = Leaf 4 < Leaf 6
```

Typ `Tree a` musi być instancją klasy `Ord`.

```
instance Ord a => Ord(Tree a) where
  Leaf a1      <  Leaf a2      = a1<a2
  Leaf _       <  Branch _ _   = True
  Branch _ _   < Leaf _        = False
  Branch l1 r1 < Branch l2 r2 = l1<l2 || (l1==l2 && r1<r2)
  t1           <= t2           = t1<t2 || t1==t2
*Main> Leaf 4 < Leaf 6
True
```

Klasa Ord cd.

```
data  Ordering  =  LT | EQ | GT
                deriving  (Eq, Ord, Bounded, Enum, Read, Show)

class  (Eq a) => Ord a  where
    compare              :: a -> a -> Ordering
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a

    compare x y | x == y    = EQ
                | x <= y    = LT
                | otherwise = GT

    x <= y  = compare x y /= GT
    x <  y  = compare x y == LT
    x >= y  = compare x y /= LT
    x >  y  = compare x y == GT

    -- Note that (min x y, max x y) = (x,y) or (y,x)
    max x y | x <= y    =  y
            | otherwise =  x
    min x y | x <= y    =  x
            | otherwise =  y
```

Większość metod ma implementacje domyślne. Wystarczy zdefiniować metodę `compare` lub `<=` .

Klasa Ord cd.

W języku Haskell domyślna równość i porządek (jak w OCamlu) są generowane automatycznie, jeśli zostanie użyta klauzula deriving. **Istotna jest kolejność konstruktorów wartości i ich argumentów.**

```
data Kolor = Trefl | Karo | Kier | Pik
           deriving (Eq, Ord, Show)
data Tree a = Leaf a | Branch (Tree a) (Tree a)
            deriving (Eq, Ord, Show)

*Main> Trefl < Kier
True
*Main> min Pik Karo
Karo
*Main> Leaf 4 < Branch (Leaf 0) (Leaf 1)
True
*Main> Branch (Leaf 0)(Leaf 2) <=  Branch (Leaf 0)(Leaf 1)
False
```

Równość i porządek dla definiowanych typów - OCaml

W języku OCaml operatory porównania (równości i porządku) są przeciążane dla nowo definiowanych algebraicznych typów danych (patrz wykład 4). **Istotna jest kolejność konstruktorów wartości i ich argumentów.** Porównanie wartości funkcyjnych powoduje zgłoszenie wyjątku Invalid_argument. Porównanie struktur cyklicznych może spowodować zapętlenie.

```
type kolor = Trefl | Karo | Kier | Pik;;
type 'a tree = Leaf of 'a | Branch of 'a tree * 'a tree;;

# Trefl < Kier;;
- : bool = true
# min Pik Karo;;
- : kolor = Karo
# Leaf 4 < Branch (Leaf 0, Leaf 1);;
- : bool = true
# Branch (Leaf 0, Leaf 2) <=  Branch (Leaf 0, Leaf 1);;
- : bool = false
```

Klasa Enum

Na wykładzie 3, str. 22 pokazano lukier syntaktyczny Haskella dla ciągów arytmetycznych. Takie ciągi można tworzyć dla każdego typu danych, który jest instancją klasy `Enum`. Np. zapis `[1,3..]` to skrót dla wywołania funkcji `enumFromThen 1 3`.

```
class  Enum a  where
    succ, pred      :: a -> a
    toEnum          :: Int -> a
    fromEnum        :: a -> Int
    enumFrom        :: a -> [a]             -- [n..]
    enumFromThen    :: a -> a -> [a]        -- [n,n'..]
    enumFromTo      :: a -> a -> [a]        -- [n..m]
    enumFromThenTo :: a -> a -> a -> [a]  -- [n,n'..m]
```

Instancją klasy `Enum` jest typ `Char` i większość typów numerycznych.

```
*Main> ['a'..'d']

"abcd"
```

Klasa Enum cd.

Typy wyliczeniowe, zdefiniowane przez użytkownika, też mogą być instancją klasy `Enum`.

```
data Kolor = Trefl | Karo | Kier | Pik
           deriving (Eq, Ord, Show, Enum)


*Main> [Trefl ..]
[Trefl,Karo,Kier,Pik]
*Main> fromEnum Kier
2
*Main> toEnum 3::Kolor  -- trzeba sprecyzować typ wyniku
Pik
*Main> succ Karo
Kier
```

Klasy numeryczne

Typy numeryczne w Haskellu są instancjami różnych klas, tworzących hierarchię, pokazaną dalej. Na szczycie tej hierarchii znajduje się klasa `Num`.

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)  :: a -> a -> a
    negate         :: a -> a
    abs, signum    :: a -> a

    fromInteger    :: Integer -> a
```

`negate` jest funkcją negacji, jako synonimu można użyć operatora `-`. Nie jest on częścią literału numerycznego, dlatego często trzeba używać nawiasów w celu wymuszenia odpowiedniej kolejności aplikacji funkcji. Np.

`abs -8` ≡ `abs negate 8` ≡ `(abs negate) 8` <= błąd typu

Poprawnie:

`abs(-8)` lub `abs(negate 8)` lub `abs $ -8` itp.

Klasy numeryczne cd.

```
class  (Eq a, Show a) => Num a  where
    (+), (-), (*)   :: a -> a -> a
    negate          :: a -> a
    abs, signum     :: a -> a
    fromInteger     :: Integer -> a
```

`signum` sprawdza znak argumentu i zwraca -1 dla argumentów ujemnych, 0 dla 0 i 1 dla argumentów dodatnich.

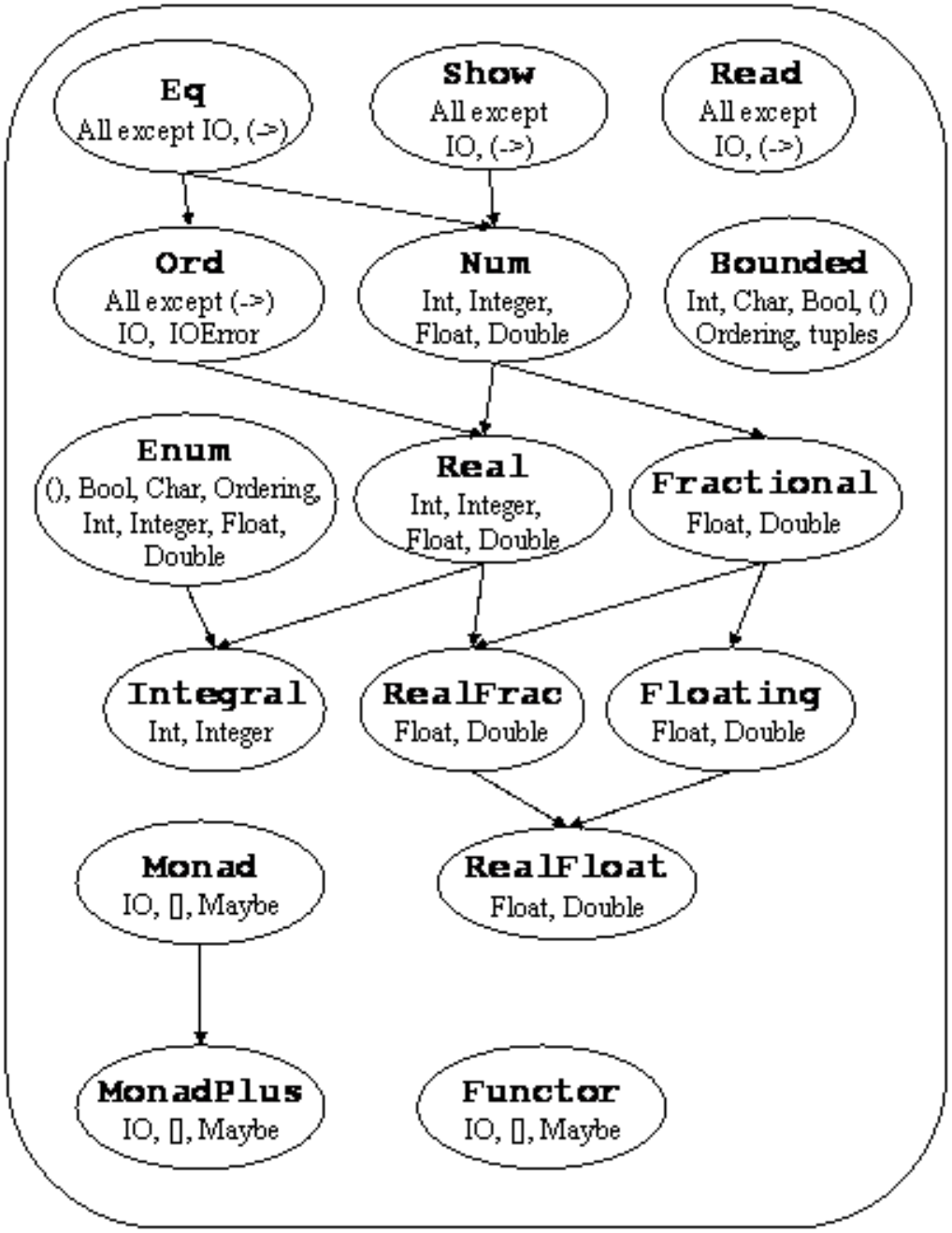
`fromInteger` jest funkcją koercji.

```
*Main> :t fromInteger
fromInteger :: Num a => Integer -> a
```

Każdy literał całkowitoliczbowy, np. „`55`” jest w rzeczywistości skrótem dla „`fromInteger 55`”.

```
*Main> :t 55
55 :: Num a => a
```

Hierarchia klas numerycznych Haskella



Typy wyższego rzędu

W Haskellu na wszystkich poziomach (aplikacja funkcji, konstruktory wartości, konstruktory typów) konsekwentnie stosowana jest notacja prefiksowa. Na wszystkich poziomach można też używać postaci rozwiniętej. Występująca również notacja infiksowa czy miksfiksowa to tylko lukier syntaktyczny.

Funkcje:

```
*Main> :t (+)
(+) :: Num a => a -> a -> a
*Main> :t (+) 5
(+) 5 :: Num a => a -> a
```

Konstruktory wartości:

```
*Main> :t Branch
Branch :: Tree a -> Tree a -> Tree a
*Main> :t Branch (Leaf 5)
Branch (Leaf 5) :: Num a => Tree a -> Tree a
```

Typy wyższego rzędu cd.

Konstruktory wartości, notacja miksfiksowa:

```
*Main> (2,Trefl)
(2,Trefl)
*Main> :t (,)
(,) :: a -> b -> (a, b)
*Main> :t (,) 2
(,) 2 :: Num a => b -> (a, b)
*Main> :t (,,)
(,,) :: a -> b -> c -> (a, b, c)
```

itd. dla wszystkich krotek.

Konstruktory wartości, notacja infiksowa:

```
*Main> 0:[1,2]
[0,1,2]
*Main> :t (:)
(:) :: a -> [a] -> [a]
*Main> :t (:[1,2])  -- por. wykład 3, str.8
(:[1,2]) :: Num a => a -> [a]
```

Typy wyższego rzędu cd.

Konstruktory typów, notacja miksfiksowa:

```
-- toPair :: a -> b -> (a,b)
toPair :: a -> b -> (,) a b
toPair a b = (a,b)   -- toPair a b = (,) a b


-- toTriple :: a -> b -> c -> (a,b,c)
toTriple :: a -> b ->  c-> (,,) a b c
toTriple a b c = (a,b,c)     -- toTriple a b c = (,,) a b c


-- toList :: a -> [a]
toList :: a -> [] a
toList a = [a]   -- toList a = (:) a []
```

Typy wyższego rzędu cd.

Konstruktory typów, notacja infiksowa:

```
-- apply :: ( a -> b) -> a -> b
-- apply :: (->) a b -> a -> b
-- apply :: (->) ((->) a b) (a -> b)
apply :: (->) ((->) a b) ((->) a b)
apply f a = f a
```

W komentarzach powyżej pokazano – krok po kroku – zamianę notacji infiksowej na prefiksową.

Typy wyższego rzędu cd.

Można zadać sobie pytanie: jaki jest „typ” typu `Int`,
czy konstruktorów typów `[]`, `(->)`, `(,)`, `(,,)` itd.?

„Typ” typu jest nazywany *gatunkiem* (ang. kind) i jest oznaczany symbolem *.
Reprezentuje on rodzaj dla typu, który może być przypisany konkretnej wartości. Np. gatunkiem typu `Int`, `Kolor` czy `Tree Int` jest *. A jaki jest gatunek konstruktora typu `Tree`, czy wyżej wymienionych konstruktorów typów? Potrzebna jest definicja indukcyjna.

1. Symbol * jest gatunkiem.
2. Jeśli κ_1 i κ_2 są gatunkami, to $\kappa_1 \rightarrow \kappa_2$ jest gatunkiem (konstruktora) typu, który bierze jako argument typ gatunku κ_1 i zwraca typ gatunku κ_2.

Konstruktory typów `Tree`, `[]` są gatunku * → *, `(,)` i `(->)` są gatunku * → * → *,
`(,,)` jest gatunku * → * → * → *, itd. Typ `(,)Int` jest gatunku * → *.
Typ `(,)Int Kolor` jest gatunku *. Aplikacja typów ma łączność lewostronną, tak jak aplikacja funkcji.

Zwykle gatunki nie występują bezpośrednio w programach Haskella, są jednak wykorzystywane przez system typów Haskella. Ułatwiają też zrozumienie niektórych klas typów i komunikatów o błędach gatunku.

Typy wyższego rzędu cd.

Jak widzieliśmy już w materiałach do wykładu 1, a także do wykładu bieżącego, w środowisku interakcyjnym GHCi Haskella istnieje komenda `:type` (w skrócie `:t`), pokazująca typ dowolnego wyrażenia. Analogicznie za pomocą komendy `:kind` (w skrócie `:k`), można sprawdzić gatunek dowolnego typu. W Haskellu dla wielu typów (np. dla list) dla konstruktora typu i konstruktora wartości jest używana ta sama notacja (w przypadku list `[]`). Są one rozróżniane na podstawie kontekstu użycia.

```
Prelude> :kind Int
Int :: *
Prelude> :k []
[] :: * -> *
Prelude> :t []
[] :: [a]
Prelude> :k (,)
(,) :: * -> * -> *
Prelude> :k (,) Char
(,) Char :: * -> *
Prelude> :t (,)
(,) :: a -> b -> (a, b)
Prelude> :t (,) 'x'
(,) 'x' :: b -> (Char, b)
Prelude> :k (,,)
(,,) :: * -> * -> * -> *
Prelude> :k Tree
Tree :: * -> *
```

Klasa Functor

Klasa Functor jest zdefiniowana w standardowym preludium Haskella.

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

Instancjami klasy Functor mogą być typy, na wartościach których można wykonać mapowanie. Listy, IO, and Maybe są instancjami tej klasy.
Instancje klasy Functor powinny spełniać następujące aksjomaty:

fmap id = id

fmap (f . g) = fmap f . fmap g

Wszystkie instancje klasy Functor zdefiniowane w preludium spełniają te aksjomaty.

Zauważ, że typ `f` w definicji klasy Functor jest gatunku * → * (jednoargumentowy konstruktor typu).

```
*Main> fmap (+1) [1,2]   -- List jest instancją klasy Functor
[2,3]
*Main> fmap (+1)(Just 5) -- Maybe jest instancją klasy Functor
Just 6
```

Klasa Functor cd.

Zdefiniujmy własny typ Maybe' jako instancję klasy Functor.

```
data Maybe' a = Just' a | Nothing'
   deriving (Show)

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

fmap (^2) (Just' 5)   -- Just' 25

fmap show (Just' 5)   -- Just' "5"
```

Klasa Functor pozwala na uogólnienie aplikacji funkcji jednoargumentowych na dowolne typy (instancje klasy Functor).

Klasa Functor cd.

Tree też może być instancją klasy Functor.

```
instance Functor Tree  where
  fmap f (Leaf x)        = Leaf (f x)
  fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)


*Main> fmap (+1) (Branch (Leaf 1) (Leaf 2))
Branch (Leaf 2) (Leaf 3)
```

Zwróć uwagę na informacje o błędach gatunku.

```
instance Functor (Tree Int) where
  fmap f (Leaf x)        = Leaf (f x)
  fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)

     * Expecting one fewer arguments to ‘Tree Int’
       Expected kind ‘* -> *’, but ‘Tree Int’ has kind ‘*’
     * In the first argument of ‘Functor’, namely ‘(Tree Int)’
       In the instance declaration for ‘Functor (Tree Int)’
```

Klasa Applicative

Klasa Functor pozwala nam „podnieść” (ang. to lift) funkcję jednoargumentową i zaaplikować ją bezpośrednio do wartości typu Maybe (lub innej instancji klasy Functor).

```
fmap (+2) (Just 3)  -- Just 5
```

fmap nie umożliwia jednak zaaplikowania funkcji do kilku wartości, np.

```
fmap (+) (Just 3) (Just 2)  -- nie da się
```

Do tego celu służy klasa Applicative dziedzicząca z klasy Functor.

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

Maybe jest instancją klasy Applicative, więc powyższy problem możemy rozwiązać tak:

```
pure (+) <*> Just 2 <*> Just 3  -- Just 5
```

Klasa Applicative cd.

Zadeklarujmy nasz typ Maybe' jako instancję klasy Applicative.

```
instance Applicative Maybe' where
  pure f = Just' f
  Nothing'  <*> _        =  Nothing'
  _         <*> Nothing' = Nothing'
  (Just' f) <*> (Just' x) = Just' (f x)
```

Teraz też możemy wykonać:

```
pure (+) <*> Just' 2 <*> Just' 3  -- Just' 5
```

W analogiczny sposób można „podnosić" inne funkcje:

```
pure (,) <*> Just' 2 <*> Just' 3  -- Just' (2,3)
```

To samo możemy zrobić ze złożeniem funkcji:

```
pure (.) <*> Just' (+2) <*> Just' (+3) <*> Just' 1    -- Just' 6
```

lub inaczej:

```
Just' (.) <*> Just' (+2) <*> Just' (+3) <*> Just' 1    -- Just' 6
```

Klasa Monad

Klasa Monad od wersji 7.10 GHC dziedziczy z klasy Applicative.

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

    m >> k  =  m >>= \_ -> k
    fail s  = error s
    return  = pure        -- od wersji 7.10 GHC
```

Typ `m` w definicji klasy Monad jest gatunku * → *. Domyślne implementacje metod `(>>)`, `fail` i `return` praktycznie nigdy nie wymagają zmiany. Instancje klasy Monad muszą tylko zaimplementować `(>>=)`. Operator bind `(>>=)` łączy monadę (`m a`) z funkcją `(a -> m b)`, nazywaną **funkcją monadyczną** (ang. monadic function).

Notacja “do”, omawiana na wykładzie poświęconym typowi IO, który jest instancją klasy Monad, odnosi się do wszystkich instancji tej klasy.

Klasa Monad cd.

Oprócz IO, w standardowym preludium również Maybe oraz listy zostały zadeklarowane jako instancje klasy Monad. Metoda fail dla list zwraca pustą listę [], dla Maybe zwraca Nothing, a dla IO zgłasza wyjątek user error.

Instancje klasy Monad powinny spełniać następujące aksjomaty (por. wykład 8, str.27):

return x >>= f = f x

m >>= return = m

m1 >>= (\x -> m2 >>= \y->m3) = (m1 >>= \x->m2) >>= \y->m3
jeśli x nie jest zmienną wolną w m3

Instancje obu klas: Monad i Functor powinny dodatkowo spełniać aksjomat :

fmap f xs = xs >>= return . f

Wszystkie instancje klasy Monad zdefiniowane w preludium spełniają te aksjomaty.

Klasa Monad cd.

Zadeklarujmy nasz typ Maybe' jako instancję klasy Monad.

```
instance Monad Maybe' where
-- return  =  pure         -- domyślna implementacja
   Nothing'  >>=   _     =  Nothing'
   (Just' x)   >>=   f     =    (f x)
```

Teraz też możemy wykonać np..:

```
Just' 10 >>= \x -> Just' (show (x::Int))    -- Just' "10"

Nothing' >>= \x -> Just' (show (x::Int))   -- Nothing'
```

Monada stanu

Monady są zwykle używane w obliczeniach z modyfikowalnym stanem. Typowym przykładem takiej monady jest IO, która zmienia stan „świata”, np. zawartość systemu plików, obraz na monitorze, wydruk na drukarce itp. Napiszemy swoją monadę stanu, w której stan nie jest aż tak globalny.

Każdy krok obliczeń jest wykonywany w pewnym stanie `s` i produkuje pewien wynik (wartość) `a` oraz zmodyfikowany, nowy stan. Krok obliczeń jest więc typu `s->(a,s)`.

Odpowiedni typ można zdefiniować jako:

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

`runState` jest akcesorem typu `State s a -> s -> (a, s)` (patrz wykład 4, str. 26).

Wartość typu `State s a` jest „akcją”, która może zmienić stan `s` zanim zwróci wynik typu `a` (por. wykład 8, monada IO). Dla monady stanu są też zdefiniowane dwie poniższe funkcje ewaluacji:

```
evalState :: State s a -> s -> a
evalState sm = fst . (runState sm)

execState :: State s a -> s -> s
execState sm = snd . (runState sm)
```

Monada stanu cd.

Zanim zadeklarujemy typ State jako instancję klasy Monad, musimy go zadeklarować jako instancję klasy Applicative, co pociąga za sobą konieczność zadeklarowania go jako instancję klasy Functor. Podobnie zrobiliśmy dla naszego typu Maybe'.

```
import Control.Monad       (liftM, ap)

instance Functor (State s) where
  fmap = liftM                                  -- liftM jest monadyczna wersją fmap


instance Applicative (State s) where
  pure a =  State  (\s -> (a,s))                -- kod przeniesiony z  'return' z instancji monady
  (<*>) = ap
```

Monada stanu cd.

```
instance Monad (State s) where
                -- nie zmieniaj stanu, zwróć wartość a
    --  do wersji 7.10 było tak:
    -- return a  =  State  (\s -> (a,s))
    --  od wersji 7.10 domyślna implementacja return wygląda tak:
    return = pure   -- rekomendowane jest pominięcie tej definicji
                -- step :: s -> (a, s)
                -- computeAction :: a -> State s b
    State step >>=  computeAction  =  State $ \initState ->
                -- wykonaj krok obliczeń na początkowym stanie
                -- wynikiem jest wartość pośrednia i stan pośredni
      let  (interValue,interState)  =  step initState
                -- wylicz następną akcję (zależy ona od wartości pośredniej)
           State nextStep   =  computeAction interValue
                -- wykonaj następny krok obliczeń na stanie pośrednim)
           (finalValue,finalState)  =  nextStep interState
                -- wynikiem jest wartość końcowa i stan końcowy
      in (finalValue,finalState)
--  czyli ostatecznie
--  State step >>=  computeAction
--    = State $ \initState -> dwa kroki obliczeń -> finalValue,finalState)
```

Monada stanu cd.

Wykorzystanie akcesora znacznie skraca kod.

```
instance Monad (State s) where
   -- nie zmieniaj stanu, zwróć wartość a
   -- return a  =  State  (\s -> (a,s))
   -- initAction :: State s a
   -- computeAction :: a -> State s b
   initAction >>=  computeAction  =  State $ \initState ->
     let  (interValue,interState) =  runState initAction initState
     in runState (computeAction interValue) interState
```

Konstruktor typu `State` jest gatunku * → * → *, więc `State s` jest gatunku * → *.

Warto też zdefiniować:

```
get :: State s s          -- zwraca stan jako wartość
get = State (\s->(s,s))
put :: s -> State s ()    -- zmienia stan na zadany, zwracana wartość nas nie interesuje
put s = State (\_ -> ((),s))
```

Monada stanu - przykład

Jako prosty, dydaktyczny przykład wykorzystania monady stanu rozważmy następujące zadanie.

Mamy zegar, który może wykonywać pewne akcje. Stanem zegara jest oczywiście aktualny czas. Akcja Tick przesuwa czas, SetTime pozwala ustawić nowy czas, a Display wyświetla aktualny czas bez jego zmiany. Typ dla akcji może być taki:

```
data Act = Tick|Display|SetTime Int
         deriving (Eq, Show)
```

Należy napisać funkcję

```
runClock :: [Act] -> [Int]
```

która dla danej listy akcji zegara zwraca listę odczytanych czasów.

Dość naturalne byłoby tu wykorzystanie monady stanu `State Int [Int]`.

Monada stanu – przykład cd.

Pierwsza wersja nie wykorzystuje monad.

```
runClock :: [Act] -> [Int]
runClock acts = fst(runC acts 0)


runC :: [Act] -> Int -> ([Int],Int)
runC (Tick:acts) t = let (displays, time)= runC acts (t+1)
                     in (displays, time)
runC (Display:acts) t = let (displays, time)=runC acts t
                        in (t:displays, time)
runC ((SetTime nt):acts) t = let (displays, time)= runC acts nt
                             in (displays, time)
runC [] t = ([], t)
*Main> runClock [Display,Tick,Tick,Display,Tick,SetTime 5,Tick,Display]
[0,2,6]
```

Monada stanu – przykład cd.

A teraz wersja z monadą stanu.

```
type Clock a = State Int a
mrunClock :: [Act] -> [Int]
mrunClock acts = evalState (mrunC acts) 0
              -- fst (runState (mrunC acts) 0)

mrunC :: [Act] -> Clock [Int]
mrunC (Tick:acts) = do t <- get
                       put (t+1)
                       displays <- mrunC acts
                       return displays
mrunC (Display:acts) = do  t <- get
                           displays <- mrunC acts
                           return (t:displays)
mrunC ((SetTime nt):acts) = do put nt
                               displays <- mrunC acts
                               return displays
mrunC [] = return []
*Main>  mrunClock [Display,Tick,Tick,Display,Tick,SetTime 5,Tick,Display]
[0,2,6]
```

Wykorzystanie monady State z biblioteki standardowej wymaga jej zaimportowania (import Control.Monad.State).

Konstruktor typu State s rodzaju * → * jest instancją klasy Monad, jak to pokazano wyżej, a także klasy MonadState, definiującej prosty interfejs dla monad stanu (przedstawione wyżej funkcje get i put).

Więcej informacji o typach i klasach typów w Haskellu można znaleźć w: Haskell 2010. Language Report, Chapter 6. Predefined Types and Classes.

http://www.haskell.org/onlinereport/haskell2010/haskellch6.html#x13-1160006

Migracja GHC do wersji 7.10.x

Od wersji 7.10 GHC klasa Applicative jest nadklasą klasy Monad. Może to powodować konieczność uaktualnienia starych programów, np. znalezionych w książkach bądź w internecie. Wskazówki można znaleźć na stronie:

https://ghc.haskell.org/trac/ghc/wiki/Migration/7.10