Programowanie funkcyjne

HASKELL

Częściowe aplikowanie (partial application)

Równoważne:

```
1 add :: Int -> Int -> Int
  add x y = x + y

2 add :: Int -> Int -> Int
  add = \x y -> x + y

3 add :: Int -> (Int -> Int)
  add = \x -> (\y -> x + y)
```

```
ghci> map (add 1) [1, 2, 3]
[2,3,4]
```

Sekcje

Infiksowe operatory w istocie są funkcjami, co sprawia, że mogą być częściowo aplikowane. W Haskellu częściowa aplikacja operatora infiksowego nazywana jest **sekcją** (sekcja to częściowo zaaplikowany operator)

Na przykład:

```
(x+)   ≡   \y -> x+y
(+y)   ≡   \x -> x+y
(+)    ≡   \x y -> x+y
```

Sekcje

```
Prelude> map (+1) [2,3,4]
[3,4,5]
Prelude> map (1+) [2,3,4]
[3,4,5]
Prelude> map (>2) [2,3,4]
[False,True,True]
Prelude> map (2>) [2,3,4]
[False,False,False]
```

Sekcje

```
Prelude> map (1-) [2,3,4]
[-1,-2,-3]
Prelude> map (-1) [2,3,4]

<interactive>:30:1:
    Could not deduce (Num (a0 -> b))
      arising from the ambiguity check for ‘it’
    from the context (Num (a -> b), Num a)
      bound by the inferred type for ‘it’: (Num (a -> b), Num a) => [b]
      at <interactive>:30:1-16

    The type variable ‘a0’ is ambiguous
    When checking that ‘it’
      has the inferred type ‘forall a b. (Num (a -> b), Num a) => [b]’
    Probable cause: the inferred type is ambiguous
Prelude> map (flip (-) 1) [1, 2, 3]
[0,1,2]
```

```
Prelude> map (div 2) [1,2,3]
[2,1,0]
Prelude> map (2 div) [1,2,3]

<interactive>:33:1:
    Could not deduce (Num ((a1 -> a1 -> a1) -> a0 -> b))
      arising from the ambiguity check for ‘it’
    from the context (Num ((a2 -> a2 -> a2) -> a -> b),
                      Num a,
                      Integral a2)
      bound by the inferred type for ‘it’:
                 (Num ((a2 -> a2 -> a2) -> a -> b), Num a, Integral a2) => [b]
      at <interactive>:33:1-19
    The type variables ‘a0’, ‘a1’ are ambiguous
    When checking that ‘it’
      has the inferred type ‘forall a b a1.
                             (Num ((a1 -> a1 -> a1) -> a -> b), Num a, Integral a1) =>
                             [b]’
    Probable cause: the inferred type is ambiguous
Prelude> map (`div` 2) [1,2,3]
[0,1,1]
Prelude> map (2 `div`) [1,2,3]
[2,1,0]
Prelude> map (flip div 2) [1,2,3]
[0,1,1]
```

Filtrowanie list

- *filter*

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []                 = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise = filter p xs

ghci> filter even [3, 2, 1, 4]
[2, 4]
```

- *any*

```
any :: (a -> Bool) -> [a] -> Bool
any _ []                 = False
any p (x:xs) | p x       = True
             | otherwise = any p xs

ghci> any even [3, 2, 1, 4]
True
```

- *takeWhile*

```
takeWhile :: (a -> Bool) -> [a] -> [a]
takeWhile _ []                 = []
takeWhile p (x:xs) | p x       = x : takeWhile p xs
                   | otherwise = []

ghci> takeWhile (/= 'l') "kot Ali"
"kot A"
```

- *dropWhile*

```
dropWhile _ []                 = []
dropWhile p (x:xs) | p x       = dropWhile p xs
                   | otherwise = (x:xs)

ghci> dropWhile (/= 'l') "kot Ali"
"li"
```

Monady w Haskellu

Monady wykorzystywane są w Haskellu.

Struktura monady nadaje się do specyfikacji:

a) operacji wejścia/wyjścia,

b) wyłapywania wyjątków (np. takich jak dzielenie przez zero),

c) interfejsów graficznych.

W ujęciu Haskellowym Monadę tworzy konstruktor typów **m**, wraz z pewnymi szczególnymi operacjami wchodzącymi w skład klasy Monad.

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

Interpretacja konstruktora m jest następująca:

jeśli a jest typem wartości, m a reprezentuje ***typ obliczeń*** zwracających wartość typu a.

„Obliczenie" i „zwracanie wartości" należy rozumieć abstrakcyjnie. Obliczenie typu m a może być na przykład:

- po prostu wartością typu a - m oznacza wtedy trywialne obliczenie;
- wartością typu a lub wartością wyjątkową, reprezentującą błędne obliczenie;
- zbiorem możliwych wartości typu a - m a oznacza wtedy obliczenie niedeterministyczne;
- programem z efektami ubocznymi, reprezentowanym przez funkcję typu s -> (a, s), gdzie s jest typem stanu modyfikowanego przez funkcję.

Operacja **return** konstruuje obliczenie zwracające daną wartość.

(f >>= g) to sekwencyjne złożenie obliczeń f i (g a), gdzie a jest wartością obliczenia f.

(f >> h) to sekwencyjne złożenie obliczeń f i h, przy czym h nie zależy od wartości obliczenia f.

>> można zdefiniować przy pomocy >>= (ćwiczenie)

Monada Id

Monada **Id** – opisuje obliczenia nie robiące nic, poza zwróceniem wartości

```
data Id a = Id a

instance Monad Id where
    return = Id
    (Id a) >>= f = f a
```

Monada Maybe

Monada **Maybe** oparta jest na konstruktorze Maybe.

Wartości typu **Maybe a** reprezentują wynik typu **a** lub błędne obliczenie, reprezentowane przez konstruktor **Nothing**.

```
data Maybe a = Just a | Nothing

instance Monad Maybe where
    return = Just
    Nothing  >>= m = Nothing
    (Just a) >>= m = m a
```

Monada definiowana dla Maybe jest podobna do monady „listowej”: wartość Nothing przedstawiana jest jako [], a Just x jako [x]

Maybe

```
data Maybe a = Nothing | Just a

safediv :: Int -> Int -> Maybe Int
safediv _ 0 = Nothing
safediv m n = Just (m `div` n)

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

ghci> safediv 3 2
Just 1
ghci> safediv 3 0
Nothing
ghci> safehead "haskell"
Just 'h'
ghci> safehead []
Nothing
```

Data.Maybe

Monada IO

Monada **IO** pozwala na wyrażenie w Haskellu operacji mającej efekty uboczne, takie jak operacje wejścia/wyjścia.

Niech typ World reprezentuje wszystkie możliwe stany świata.

Obliczenie zwracające wynik typu **a** i zwracające przy tym stan świata (jako efekt uboczny) może być traktowane jako element

World -> (a, World)

W Haskellu zamiast World -> (a, World) używa się abstrakcyjnego typu **IO a.**

Programy interaktywne

Typ reprezentujący operacje IO

- funkcja zmieniająca „stan świata”

  type IO = World -> World
- funkcja zmieniająca „stan świata” i zwracająca wynik

  type IO a = World -> (a, World)

Akcje

Akcja to wyrażenie typu IO a

IO Char (typ akcji zwracającej znak)

IO () (typ akcji zwracającej pustą krotkę)

Typ jednostkowy

data () = ()

Podstawowe akcje

- akcja **getChar** wczytuje znak z klawiatury, wyświetla go na ekranie i zwraca jako rezultat

  getChar :: IO Char
- akcja **putChar c** wyświetla znak c na ekranie i zwraca pustą krotkę

  putChar :: Char -> IO ()
- akcja **return** v zwraca wartość v bez jakichkolwiek interakcji

  return :: a -> IO a

  return v = \world -> (v, world)

Operator sekwencji

(>>=) :: IO a -> (a -> IO b) -> IO b

f >>= g = \world -> case f world of

(v, world') -> g v world'

Uwaga

Jak w przypadku parserów zamiast operatora >>= można korzystać z notacji **do**

Sekwencję elementów monady tłumaczy się na notację (>>=) i (**>>**) następująco:

```
do a <- e        |-->    e >>= \ a -> do e'
   e'

do e             |-->    e >> do e'
   e'

do let x = e     |-->    let x = e
   e'                     in  do e'

do e             |-->    e
```

Operator sekwencji

Przykład

```
a :: IO (Char, Char)
a = do x <- getChar
       getChar
       y <- getChar
       return (x, y)
```

```
*Main> a
123456
('1','3')
*Main> *Main>
('4','6')
```

Uwaga

<- przypomina podstawienie, ale nim nie jest.

Zapis z tą strzałką oznacza uruchomienie akcji, wyciągniecie jej wyniku i skojarzenie go ze zmienną, która jest lokalna względem dalszej części i która - jak wszystkie dane w językach funkcyjnych - już swojej wartości nie zmieni. Może jednak zostać przysłonięta, bo jeśli pojawi się w jednym bloku **do** zapis:

x <- ... ... x <- ...

to drugie x przysłania pierwsze, a nie jest tym samym.

getLine

```
getLine :: IO String
getLine = do  x <- getChar
              if x == '\n'  then return []
                else
                  do xs <- getLine
                     return (x:xs)
```

putStr

```
putStr :: String -> IO ()
putStr [] = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

putStrLn (**put** a **Str**ing followed by a new **Lin**e

```
putStrLn :: String -> IO ()
putStrLn xs = do
              putStr xs
              putChar '\n'
```

print

print :: Show a => a -> IO ()

najpierw wykonuje Show na argumencie po czym przekazuje wynik do putStrLn i zwraca akcję wejścia/wyjścia, która przekazuje do terminala.

print x = putStrLn (show x)

```
Prelude> print "PPD"
"PPD"
Prelude> print ['p','p','d']
"ppd"
Prelude> print []
[]
```

```
Prelude> putStrLn (show 1)
1
Prelude> print 1
1
Prelude> print (show [1,2,3])
"[1,2,3]"
Prelude> print [1,2,3]
[1,2,3]
```

Przykład

```
main = do
putStrLn "Podaj imie:"
imie <- getLine
putStrLn („Witaj " ++ imie ++ ".")
```

Część kodu imie <- getLine czytamy następująco:

Wykonaj akcję wejścia/wyjścia getLine, a następnie związ jej wartość wynikową z imie.

getLine posiada typ IO String zatem imie będzie miało typ String

Akcje

```
*Main> putStr "Ala  ma  kota . "
Ala  ma  kota . *Main> putStr "Ala  ma  kota . "
Ala  ma  kota . *Main> putStrLn "Ala  ma  kota . "
Ala  ma  kota .
*Main> getLine
Ala ma kota
"Ala ma kota"
*Main> getLine >>= putStrLn
Ala ma kota
Ala ma kota
```

Akcje

```
*Main> putStr "Jak  sie  nazywasz?  " >> getLine
Jak  sie  nazywasz?  Anna
"Anna"
*Main> putStr "Jak  sie  nazywasz?  " >> getLine >>= putStrLn
Jak  sie  nazywasz?  Anna
Anna
```

Złe składanie akcji

```
*Main> putStrLn ( "Witaj ,  "
++ (putStr "Jak  sie  nazywasz?  " >> getLine)
++ " ! " )

<interactive>:29:28:
    Couldn't match type 'IO' with '[]'
    Expected type: [()]
      Actual type: IO ()
    In the first argument of '(>>)', namely
      'putStr "Jak  sie  nazywasz?  "'
    In the first argument of '(++)', namely
      '(putStr "Jak  sie  nazywasz?  " >> getLine)'

<interactive>:29:62:
    Couldn't match type 'IO' with '[]'
    Expected type: [Char]
      Actual type: IO String
    In the second argument of '(>>)', namely 'getLine'
    In the first argument of '(++)', namely
      '(putStr "Jak  sie  nazywasz?  " >> getLine)'
```

Dobre składanie akcji z użyciem funkcji

```
main1 = putStr "Jak sie nazywasz: "
        >> getLine
        >>= powitaj
            where powitaj imie
                  = putStrLn ("Witaj, " ++ imie ++ "!")

main2 = putStr "Jak sie nazywasz: "
        >> getLine
        >>= \imie -> putStrLn ("Witaj, " ++ imie ++ "!")
```

W **main1** rozwiązujemy problem przez zdefiniowanie własnej funkcji powitaj zależnej od parametru i zwracającej akcję używającą tego parametru. Takie podejście ma tę wadę, że w bardziej skomplikowanym programie trzeba zdefiniować wiele takich funkcji pomocniczych.
Jednakże, dzięki rachunkowi lambda można wstawiać w potrzebne miejsce od razu funkcje anonimowe, co zostało wykorzystane w definicji **main2**.

Dobre składanie akcji z użyciem funkcji

```
main4 = putStr "Jak sie nazywasz: "
        >> getLine >>= \imie
        -> putStrLn ("Witaj, " ++ imie ++ "!")

main5 = do putStr "Jak sie nazywasz: "
           imie <- getLine
           putStrLn ("Witaj, " ++ imie ++ "!")
```

Definicja **main4** jest dokładnie taka, jak main2, różni się jedynie podziałem na wiersze. Zwróćmy uwagę na to, że zmienna pomocnicza imie dostaje wynik z akcji getLine.
Definicja **main5** (równoważna poprzednim) używa notacji **do**.

Przykłady

```
echo :: IO ()
echo = do line <- getLine
          putStr line

palindrom :: IO ()
palindrom = do putStr "Napisz cos: "
               line <- getLine
               let line' = filter (not . (==) ' ') (map toLower line)
               if line' == reverse line'
                   then putStr ("'" ++ line ++ "' jest palindromem!\n")
                   else putStr ("'" ++ line ++ "' nie jest palindromem.\n")
```

```
*Main> palindrom
Napisz cos: einawargorp
'einawargorp' nie jest palindromem.
*Main> palindrom
Napisz cos: abcdcba
'abcdcba' jest palindromem!
```

Przykład

```
strlen :: IO ()
strlen = do putStr "Enter a string: "
            xs <- getLine
            putStr "String ma "
            putStr (show (length xs))
            putStrLn " znakow"
```

```
*Main> strlen
Enter a string: Programowanie funkcyjne
String ma 23 znakow
*Main> strlen
Enter a string: Haskell
String ma 7 znakow
```

Przykład

```
power = do putStr "Podaj liczbe: "
           n <- getLine
           let x = read n
               y = x^2
           putStrLn (n ++ " do kwadratu: " ++ show y)
```

```
*Main> power
Podaj liczbe: 15
15 do kwadratu: 225
```

Funkcje z rodziny typów IO (podsumowanie)

- putChar :: Char -> IO ()
  pobiera znak jako parametr i zwraca akcję wejścia/wyjścia, która pisze ten znak do terminala
- putStr :: String -> IO ()
  pobiera string jako parametr i zwraca akcję wejścia/wyjścia, która pisze do terminala (nie przechodzi do nowej linii)
- putStrLn :: String -> IO ()
- print :: Show a => a -> IO ()
  najpierw wykonuje Show na argumencie po czym przekazuje wynik do putStrLn i zwraca akcję wejścia/wyjścia, która pisze do terminala.
- getChar :: IO Char czyta znak ze standardowego wejścia
- getLine :: IO String

Zauważmy, że funkcje „wyjściowe" zwracają wynik typu IO (), gdzie () oznacza typ pusty, zaś funkcje „wejściowe" zwracają wynik typu IO a, gdzie a jest typem wczytywanej wartości.

Przydatne funkcje

Funkcja **when** znajduje się w **Control.Monad**.

Jest ona interesująca z tego względu, że w bloku ***do*** wygląda jak wyrażenie sterujące przepływem. Przyjmuje ona wartość logiczną i w przypadku fałszu zwraca return () zaś dla prawdy akcję wejścia/wyjścia.

Przykład:

```
main = do
   c <- getChar
   when (c /= ' ') $ do
     putChar c
     main
```

Przydatne funkcje

Funkcja **sequence** pobiera listę akcji wejścia/wyjścia i zwraca te akcje wykonywane jedna po drugiej.

Przykład:

```
main = do
   a <- getLine
   b <- getLine
   c <- getLine
   print [a,b,c]
```

Można zapisać np. jako

```
main = do
   rs <- sequence [getLine, getLine, getLine]
   print rs
```

Przydatne funkcje

Funkcja **forever** pobiera akcję wejścia/wyjścia i zwraca tę akcję powtarzając ją.

Przykład:

```
md = forever $ do
   putStr " Wprowadź ciąg znaków"
   k <- getLine
   putStrLn $ map toUpper k
```

Monady

Dla list:

(>>=) :: [a] -> (a ->[b]) -> [b]

List comprehensions może być wyrażona za pomocą operacji monadycznych dla list

1
```
[(x,y) | x <- [1,2,3] , y <- [1,2,3], x /= y]
```

2
```
do x <- [1,2,3]
   y <- [1,2,3]
   True <- return (x /= y)
   return (x,y)
```

```
Prelude> [(x,y) | x<-[1,2,3], y<-[1,2,3], x/=y]
[(1,2),(1,3),(2,1),(2,3),(3,1),(3,2)]
```

3
```
[1,2,3] >>= (\ x -> [1,2,3] >>= (\y -> return (x/=y) >>=
   (\r -> case r of True -> return (x,y)
                    _    -> fail "")))
```

Monady

X <- [1,2,3] oznacza monadyczne obliczenie 3 razy, raz dla każdego elementu listy

```
mvLift2              :: (a -> b -> c) -> [a] -> [b] -> [c]
mvLift2 f x y        =  do x' <- x
                           y' <- y
                           return (f x' y')
```

```
mvLift2 (+) [1,3] [10,20,30]      ⇒   [11,21,31,13,23,33]
mvLift2 (\a b->[a,b]) "ab" "cd"   ⇒   ["ac","ad","bc","bd"]
mvLift2 (*) [1,2,4] []            ⇒   []
```

Literatura

- B.O'Sullivan,J.Goerzen, D.Stewart, Real World Haskell, O'REILLY, 2008.
- K.Doets, J.van Eijck, The Haskell Road to Logic, Math and programming, 2004.
- G.Brzykcy, A.Meissner, Programowanie w Prologu i programowanie funkcyjne, Wyd.PP, 1999.
- M.Lipovaca, Learn You a Haskell for Great Good!
- J.Bylina, B.Bylina, Przegląd języków i paradygmatów programowania, UMCS, 2011