Wykład 9

Haskell - Algebraiczne typy danych; mechanizmy ewaluacji gorliwej

Definicje typów jako aliasy (synonimy)

Algebraiczne typy danych

Klauzula deriving

Równość i porządek dla definiowanych typów

Rekordy

Typ „opcjonalny”

Mechanizmy ewaluacji gorliwej

Rekursja ogonowa

Przekazywanie argumentów do funkcji

Użyteczne informacje

Haskell 2010 Language Report

https://www.haskell.org/onlinereport/haskell2010/

Hoogle jest wyszukiwarką, przeszukującą standardowe biblioteki Haskella.

https://www.haskell.org/hoogle/

Umożliwia też wyświetlenie kodu źródłowego funkcji (należy kliknąć w nazwę znalezionej funkcji, a potem w zakładkę # Source z prawej strony).

Standardowy moduł Prelude

Aktualną wersję można znaleźć za pomocą Hoogle.

Niektóre typy bazowe

Bool – ma dwie wartości: True i False

Char – znak (Unicode)

Wiele użytecznych funkcji znajduje się w module Data.Char.

W razie potrzeby należy go zaimportować, umieszczając dyrektywę import Data.Char na początku pliku (a dokładniej modułu).

Int – liczba całkowita ze znakiem o ustalonym rozmiarze

Integer – liczba całkowita ze znakiem dowolnej wielkości

Double – liczba zmiennoprzecinkowa podwójnej precyzji

Float – liczba zmiennoprzecinkowa pojedynczej precyzji

() – typ pustej krotki (unit); jedyna wartość jest również zapisywana jako (), czyli () :: ()

Definicje typów jako aliasy (synonimy)

Typy jako aliasy definiowane są w Haskellu za pomocą słowa kluczowe type. Wszystkie przykłady z OCamla (wykład 4, str. 4) można przepisać w Haskellu. Np.

```
type String = [Char]                -- napisy w Haskellu  są po prostu listami znaków

type Para_i_x param = (Int,param)
type Para_i_f  = Para_i_x Float
Prelude> :set +t
Prelude> let x = (3,3.14)
x :: (Fractional b, Num a) => (a, b)
Prelude> let x = (3,3.14) :: Para_i_f
x :: Para_i_f

x :: Para_i_x  Float              -- plik
x = (3,3.14)                      -- plik
*Main> :t x
x :: Para_i_x  Float
```

Algebraiczne typy danych

Algebraiczne typy danych)definiuje się za pomocą ***konstruktorów typów*** i ***konstruktorów wartości***. Konstruktorem typu dla list w języku OCaml jest list (w notacji postfiksowej), np. int list, float list są już typami, a int i float są argumentami dla konstruktora typu list. Istnieją dwa konstruktory wartości dla list: [] dla listy pustej i infiksowy konstruktor :: dla listy niepustej. We współczesnych funkcyjnych językach programowania ze statyczną typizacją istnieją mechanizmy, pozwalające programiście definiować swoje algebraiczne typy danych.

W języku Haskell dla konstruktorów typów stosowana jest notacja prefiksowa.

Notacja miksfiksowa, np. [Int] to tylko lukier syntaktyczny. Konstruktory wartości mogą być w postaci rozwiniętej, a konstruktory typów muszą być w takiej postaci. Nazwy konstruktorów typów i konstruktorów wartości muszą się zaczynać od wielkiej litery.

Definicje algebraicznych typów danych

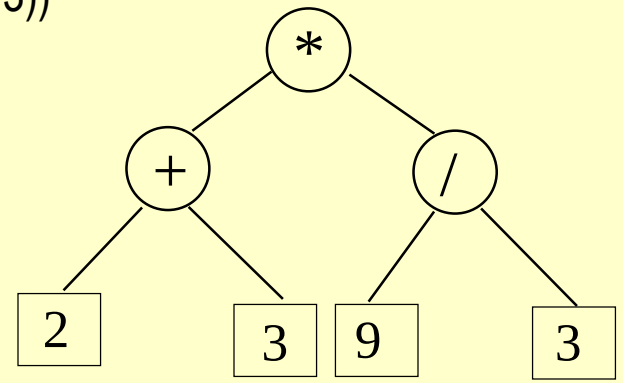
Do definiowania algebraicznych typów danych w Haskellu używa się słowa kluczowego data. Konstruktory typów danych muszą być w postaci rozwiniętej, a konstruktory wartości mogą być w postaci rozwiniętej. Konstruktory wartości mogą być używane jak zwykłe funkcje.

```
data BT a b = Leaf a | Node b (BT a b) (BT a b)
                deriving (Eq,Show,Read)

nrOfLeaves :: BT a b -> Int
nrOfLeaves (Leaf _) = 1
nrOfLeaves (Node n l r) =  nrOfLeaves l + nrOfLeaves r

t1 =
   Node '*' (Node '+' (Leaf 2) (Leaf 3)) (Node '/' (Leaf 9) (Leaf 3))

*Main> nrOfLeaves t1
4
```



Klauzula deriving

Klauzula deriving umożliwia włączenie definiowanego algebraicznego typu danych do niektórych klas typów (kompilator generuje odpowiedni kod). O klasach typów będzie mowa później.

W klauzuli deriving najczęściej używane są klasy: Eq, Ord, Show, Read.

Eq – instancje tej klasy mają zdefiniowane operatory równości == i nierówności /= (oczywiście strukturalnej).

Ord – instancje tej klasy mają zdefiniowany porządek.

Show – instancje tej klasy umożliwiają konwersję wartości do napisów (do typu String). Jest to konieczne, jeśli chcemy je np. wyświetlić.

Read – instancje tej klasy posiadają funkcję

```
read :: Read a => String -> a
```

która parsuje napis i zwraca wartość odpowiedniego typu.

Równość i porządek dla definiowanych typów

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.** Można je zdefiniować inaczej za pomocą własnych klas typów, o których będzie mowa później.

```
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
```

Definicje typów są generatywne

Każda definicja typu tworzy nowy typ nawet wtedy, kiedy wcześniej zdefiniowano identyczny typ (podobnie było w OCamlu).

```
data Bool = False | True
          deriving(Eq,Ord,Show,Read)
*Main> not True
<interactive>:1:4: error:
   Ambiguous occurrence 'True'
   It could refer to either 'Prelude.True',
                            imported from `Prelude' at w9.hs:1:1
                          (and originally defined in 'GHC.Types')
                        or  'Main.True', defined at w9.hs:1:19

*Main> not Main.True
<interactive>:1:4: error:
   Couldn't match expected type `Prelude.Bool'
          with actual type `Main.Bool'
   In the first argument of 'not', namely 'Main.True'
   In the expression: not Main.True
*Main> not Prelude.True
False
```

Definicje newtype

W Haskellu do tworzenia nowych typów można też użyć słowa kluczowego newtype. Definiowany typ musi mieć dokładnie jeden konstruktor wartości z dokładnie jednym argumentem. Konstruktor typu może mieć dowolnie wiele argumentów. Konstruktor wartości takiego typu jest wykorzystywany do jawnej koercji typów. Ta informacja wykorzystywana jest wyłącznie w czasie kompilacji i nie powoduje żadnego narzutu w czasie wykonania.

```
newtype IntX x y = D (Int,x,y) deriving (Show)
ixy = D(5,'x',[5])
D(i,x,y) = ixy    -- dopasowanie do wzorca

*Main> ixy
D (5,'x',[5])
it :: IntX Char [Integer]
*Main> i
5
it :: Int
*Main> x
'x'
it :: Char
*Main> y
[5]
it :: [Integer]
```

Rekordy w języku Haskell

Rekordy w Haskellu są lukrem syntaktycznym. Pozwalają nazywać pola i generują akcesory, czyli funkcje, umożliwiające dostęp do pól. Mogą być użyte jako argumenty konstruktorów wartości w definicjach typów. Konstruktory wartości z argumentami rekordowymi mogą być używane ze zwykłą składnią zarówno przy tworzeniu wartości jak i we wzorcach (trzeba tylko zachować odpowiednią kolejność argumentów). Nie można używać tych samych nazw pól w różnych rekordach.

```
data Complex = Complex {
  re :: Float,
  im :: Float
  } deriving (Eq,Show,Read)

add_complex ::  Complex ->  Complex ->  Complex  -- z akcesorami
add_complex c1 c2 = Complex {re=re c1 + re c2, im=im c1 + im c2}

add_complex' ::  Complex ->  Complex ->  Complex -- z dopasowaniem do wzorca
add_complex' (Complex x1 y1) (Complex x2 y2) = Complex {re=x1+x2, im=y1+y2}

let c = Complex {re = 2.0, im = 3.0}
let c' = Complex 2.0 3.0 –- można użyć tradycyjnej składni
*Main> c == c'
True
*Main> c == Complex {im = 3.0, re = 2.0} -- kolejność pól jest nieistotna
True
```

Typ „opcjonalny”

Typ opcjonalny można wykorzystywać jako bezpieczną alternatywę dla użycia wyjątków lub wartości null. Z tego powodu pojawia się on (pod różnymi nazwami) we wszystkich popularnych językach programowania.

Patrz:

Wikipedia

https://en.wikipedia.org/wiki/Option_type

W Haskellu typ opcjonalny nosi nazwę Maybe i jest zdefiniowany następująco;:

```
data Maybe a = Nothing | Just a
                    deriving(Eq,Ord,Show,Read)
```

Ewaluacja gorliwa w języku Haskell: funkcje seq i deepseq

Haskell domyślnie wykorzystuje leniwą ewaluację, ale widzieliśmy na wykładzie 2 (patrz też niżej) przykład, kiedy potrzebna była ewaluacja gorliwa.

Haskell w standardowym preludium ma zdefiniowaną funkcję seq:

```
seq :: a -> b -> b
```

Wymusza ona ewaluację pierwszego argumentu, a następnie zwraca drugi argument.

Ewaluacja argumentu nie musi być kompletna. Kompletna ewaluacja wykonywana jest dla typów bazowych, np. dla Int czy Bool. Dla wartości strukturalnych jest inaczej. Jeśli argument jest parą, np. (Int, Bool), to ewaluacja jest kontynuowana do otrzymania pary wyrażeń, które nie muszą jeszcze być wartościami. Ogólnie, ewaluacja jest kontynuowana do osiągnięcia **konstruktora wartości**.

```
Prelude> (1, undefined) `seq` "OK"
"OK "
```

W module Control.DeepSeq jest zdefiniowana funkcja deepseq, przeprowadzająca ewaluację pierwszego wyrażenia do postaci normalnej i zwracająca drugi argument.

```
deepseq :: NFData a => a -> b -> b                -- NFData == Normal Form Data
```

```
Prelude> ((1, undefined)::(Int,Int)) `Control.DeepSeq.deepseq` "OK"
*** Exception: Prelude.undefined
```

Ewaluacja gorliwa w języku Haskell: funkcje seq *i* deepseq

Funkcja seq jest wykorzystana w definicji operatora gorliwej aplikacji (ang. strict application) z łącznością prawostronną (jest również w standardowym preludium) :

```
infixr 0 $!
($!)  :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

W analogiczny sposób w module Control.DeepSeq za pomocą deepseq jest zdefiniowany operator:

```
($!!)  :: NFData a => (a -> b) -> a -> b
```

W przypadku funkcji w postaci rozwiniętej operatora gorliwej aplikacji można użyć do wymuszenia gorliwej ewaluacji dowolnych argumentów.

Jeśli f jest funkcją w postaci rozwiniętej od dwóch argumentów, to aplikacja f x y może być zmodyfikowana na trzy sposoby:

```
(f $! x) y       wymusza ewaluację x
(f x) $! y       wymusza ewaluację y
(f $! x) $! y    wymusza ewaluację x i y
```

Ewaluacja gorliwa w języku Haskell: funkcje seq *i* deepseq

Funkcje seq, deepseq, ($!) i ($!!) powinny być wykorzystywane z umiarem ze względu na narzut czasowy. Zwykle standardowa ewaluacja leniwa jest najlepsza. Raczej rzadko potrzebna jest głęboka ewaluacja do postaci normalnej. Ograniczenia funkcji seq można obejść za pomocą standardowych technik programistycznych. Na przykład:

```
strictPair (a,b) = a `seq` b `seq` (a,b)

strictList (x:xs) = x `seq` x:strictList xs
strictList []     = []
```

Użyteczna może być też funkcja force z modułu Control.DeepSeq:

```
force  :: NFData a =>  a -> a
force x = x `deepseq` x
```

Ewaluacja gorliwa w języku Haskell: gorliwe konstruktory

Konstruktory wartości algebraicznych typów danych w Haskellu domyślnie są leniwe.

```
Prelude> data T = K Int deriving (Eq,Ord,Show)
Prelude> let f (K _) = "OK"
Prelude> f (K (error "arg"))
"OK"
```

Istnieje jednak możliwość podania w definicji typu informacji o gorliwej ewaluacji wybranych argumentów konstruktorów. Służy do tego symbol ! , umieszczony przed typem wybranych argumentów konstruktora wartości.

Na przykład:

```
Prelude>  data Tstrict = Kstrict  !Int   deriving (Eq,Ord,Show)
Prelude> let fstrict (Kstrict _) = "OK"
Prelude> fstrict (Kstrict (error "arg"))
*** Exception: arg
```

Rekursja ogonowa w języku Haskell (1)

Zwykła rekursja:

```
suc :: Int -> Int
suc n = if n == 0 then 1 else 1 + suc (n-1)

suc 100000000
*** Exception: stack overflow
```

Rekursja ogonowa:

```
sucTail'   :: Int -> Int
sucTail' n = sucAux 1 n
           where
             sucAux accum 0 = accum
             sucAux accum n = sucAux (accum+1) (n-1)

sucTail' 100000000
*** Exception: stack overflow
```

Rekursja ogonowa w języku Haskell (2)

W Haskellu ewaluacja jest leniwa, więc proces obliczania sucAux przebiega tak:

sucAux 1 3
= sucAux (1+1) 2
= sucAux ((1+1)+1) 1
= sucAux (((1+1)+1)+1) 0
= ((1+1)+1)+1 = (2+1)+1 = 3+1 = 4

Całe wyrażenie dla sumy w akumulatorze jest konstruowane, zanim zostaną wykonane dodawania. Pamięć zaoszczędzona na stosie została wykorzystana na zapamiętanie długiego wyrażenia. **W tym przypadku wartość akumulatora powinna być obliczana gorliwie**.

Haskell w standardowym preludium ma zdefiniowany operator gorliwej aplikacji (ang. strict application) z łącznością prawostronną:

```
infixr 0 $!
($!)  :: (a -> b) -> a -> b
f $! x = x `seq` f x
```

Operator `$` zachowuje leniwe wartościowanie, zmienia tylko łączność na prawostronną.

Rekursja ogonowa w języku Haskell (3)

Funkcja sucTail, wykorzystująca operator $! wygląda tak:

```
sucTail   :: Int -> Int
sucTail n = sucAux 1 n
          where
            sucAux accum 0  = accum
            sucAux accum n  = (sucAux $! accum+1) (n-1)

sucTail 100000000
100000001
```

Obliczenie tego wyniku trwa dużo dłużej niż w OCamlu. Szacowanie złożoności obliczeniowej programów z ewaluacja leniwą jest trudniejsze, niż w przypadku ewaluacji gorliwej.

Żadna ze strategii ewaluacji nie jest uniwersalnym panaceum. W OCamlu bywa przydatna ewaluacja leniwa, a w Haskellu – gorliwa. Trzeba rozumieć obie.

Ewaluacja gorliwa w języku Haskell: przykłady

Powyższy przykład pokazywał dość typowy przypadek, kiedy trzeba wymuszać ewaluację akumulatora dla rekursji ogonowej. W ogólnym przypadku gorliwa aplikacja jest stosowana w celu poprawienia wykorzystania pamięci.

Kolejny przykład ilustruje to dla list. Potrzebujemy funkcji, która tworzy listę kolejnych liczb całkowitych od 1 do m. Poprawna definicja powinna oczywiście wyglądać tak:

```
nats :: Int -> [Int]
nats m = [1 .. m]
```

W celu ilustracji problemu użyjemy jednak rekursji.

Ewaluacja gorliwa w języku Haskell: przykłady

```
nats' :: Int -> [Int]
nats' m = aux 1 m
    where
        aux n m = if m > 0 then n : aux (n+1) (m-1) else []
```

Wywołanie: length (nats' 100000000) na komputerze 32-bitowym z pamięcią 2GB powoduje przepełnienie pamięci: <interactive>: out of memory
Komputer 64-bitowy z pamięcią 4GB oblicza poprawny wynik.

Po wymuszeniu ewaluacji argumentu w obu przypadkach został wyliczony poprawny wynik (chociaż obliczenia trwały dużo dłużej, niż dla funkcji nats).

```
nats'' :: Int -> [Int]
nats'' m = aux 1 m
    where
       aux n m = if m > 0 then n : (aux $!  n+1) (m-1) else []
```

Ten sam efekt daje wywołanie: length (strictList (nats' 100000000))

Przekazywanie argumentów do funkcji

Mechanizmy gorliwe

- przez wartość (ang. call by value)
- przez referencję (ang. call by reference)
- przez przenoszenie w C++ (ang. move semantics)

Mechanizmy leniwe

- przez nazwę (ang. call by name)
- przez potrzebę (ang. call by need)

W językach z ewaluacją gorliwą najczęściej stosowany jest mechanizm przekazywania argumentów do funkcji przez wartość.

Przekazywanie argumentów do funkcji

Wywołanie przez wartość (ang. call by value)
Argument aktualny jest ewaluowany zawsze. Do funkcji jest przekazywana jego wartość lub odwołanie do wartości, ale funkcja nie może zmieniać tej wartości w środowisku wywołującym.
Mechanizm wykorzystywany w językach OCaml, Java, C; domyślnie w językach Scala, C++, C#, Pascal.

Wywołanie przez referencję (ang. call by reference)
Argument aktualny jest ewaluowany zawsze. Do funkcji jest przekazywane odwołanie (referencja) do obliczonej wartości. Zmiana wartości argumentu wewnątrz funkcji powoduje jej zmianę w środowisku wywołującym.
Mechanizm opcjonalny w językach C++ (argument przekazywany przez referencję jest poprzedzany symbolem `&`), Pascal (argument przekazywany przez referencję jest poprzedzany słowem kluczowym `var`), C# (słowo kluczowe `ref`).

Wywołanie przez nazwę (ang. call by name)
Argument aktualny nie jest ewaluowany. Do funkcji jest przekazywane jest jego domknięcie (ang. thunk). Obliczanie wartości domknięcia jest *wymuszane* (ang. forced) tylko wtedy, kiedy wartość argumentu jest potrzebna (za każdym razem).
Mechanizm opcjonalny w języku Scala.

Wywołanie przez potrzebę (ang. call by need)
Jest to odmiana wywołania przez nazwę. Po pierwszym wymuszeniu obliczania wartości domknięcia argumentu aktualnego, wartość ta jest zapamiętywana i wykorzystywana w dalszych obliczeniach. **Mechanizm wykorzystywany w języku Haskell.**