

Programowanie w języku Haskell

Typy danych

Marcin Szlenk (IAiIS PW)

8 marca 2018

Wyrażenia są statycznie i silnie typowane

Tabela 1: Podstawowe typy danych

| Typ | Przykładowa wartość |
|---------|---------------------|
| Int | 56 |
| Integer | 732145682358 |
| Float | 3.1415927 |
| Double | 3.141592653589793 |
| Bool | False |
| Char | 'a' |
| String | "Ala" |

Złożone typy danych: krotki, listy i funkcje

Tabela 2: Krotki (*tuple*)

| Typ | Przykładowa wartość |
|----------------------------------|-------------------------------|
| <code>(Int,Char)</code> | <code>(1,'a')</code> |
| <code>(Int,Char,Float)</code> | <code>(1,'a',3.4)</code> |
| <code>((Bool,String),Int)</code> | <code>((True,"Ala"),2)</code> |
| <code>([Int],Char)</code> | <code>([1,-2],'c')</code> |

Krotka ma określony rozmiar, ale może zawierać elementy różnego typu

Tabela 3: Listy

| Typ | Przykładowa wartość |
|------------------------------|---|
| <code>[Int]</code> | <code>[1,2,3]</code> |
| <code>[Char]</code> | <code>['a','b']</code> |
| <code>[[Int]]</code> | <code>[[1],[1,4],[]]</code> |
| <code>[(String,Bool)]</code> | <code>[("Ala",True),("kot",False)]</code> |

Lista ma nieokreślony rozmiar, ale jej elementy muszą być tego samego typu

Typ `String` jest tożsamy z `[Char]`. Napis `"ab"` jest tożsamy z listą `['a','b']`

Tabela 4: Standardowe operatory i funkcje

| Operator lub funkcja | Opis |
|---|------------------------------------|
| $a==b$, $a/=b$ | równe, nierówne |
| $a<b$, $a>b$ | mniejsze, większe |
| $a<=b$, $a>=b$ | nie większe, nie mniejsze |
| $a\&b$, $a b$, $\text{not } a$ | koniunkcja, alternatywa, negacja |
| $a+b$, $a-b$ | suma, różnica |
| $a*b$, a^b | mnożenie, potęgowanie |
| a/b , $\text{mod } a \ b$, $\text{div } a \ b$ | dzielenie, reszta, część całkowita |

Uwaga!

Za wyjątkiem potęgowania, oba argumenty muszą być tego samego typu (nie ma rzutowania)

Listy i krotki są uporządkowane leksykograficznie:

```
ghci> "Abba" < "Ala"
```

```
True
```

```
ghci> "Abba" < "abba"
```

```
True
```

```
ghci> [0,2] < [1]
```

```
True
```

```
ghci> (0,1) < (0,2)
```

```
True
```

Tabela 5: Priorytety i łączność operatorów

| Priorytet | Operator (łączność) |
|-----------|-------------------------------------|
| 9 | !! (lewostronna) , . (prawostronna) |
| 8 | ^ (prawostronna) |
| 7 | *, / (lewostronna) |
| 6 | +, - (lewostronna) |
| 5 | :, ++ (prawostronna) |
| 4 | ==, /=, <, <=, >, >= (niełączne) |
| 3 | && (prawostronna) |
| 2 | (prawostronna) |
| 1 | >>, >>= (lewostronna) |
| 0 | \$ (prawostronna) |

Tabela 6: Zasady zapisu typów funkcji

| Sygnatura | Opis |
|-----------------------------|-----------------------|
| Typ1 -> Typ | Funkcja 1-argumentowa |
| Typ1 -> Typ2 -> Typ | Funkcja 2-argumentowa |
| Typ1 -> Typ2 -> Typ3 -> Typ | Funkcja 3-argumentowa |

Ostatni typ jest typem wartości zwracanej. Poprzedzają go typy argumentów:

`Int -> Char`

`(Int,Char) -> [Char] -> Bool`

Typy wartości i funkcji są wnioskowane na podstawie ich definicji. Mogą być też jawnie zadeklarowane:

```
x :: Char
```

```
x = 'a'
```

```
isB :: Char -> Bool
```

```
isB c = c == 'B' || c == 'b'
```

Polecenie `:type` w GHCi zwraca typ wyrażenia:

```
ghci> :type isB
```

```
isB :: Char -> Bool
```

Krotki a dopasowanie argumentów funkcji do wzorca:

```
vectorLength :: (Float,Float) -> Float
```

```
vectorLength (0,y) = y
```

```
vectorLength (x,0) = x
```

```
vectorLength (x,y) = sqrt (x2 + y2)
```

```
ghci> vectorLength (0,1)
```

```
1
```

```
ghci> vectorLength (1,1)
```

```
1.4142135
```

Typy polimorficzne:

```
fst :: (a,b) -> a
```

```
fst (x,_) = x
```

```
snd :: (a,b) -> b
```

```
snd (_,y) = y
```

```
ghci> fst (True,"Ala")
```

```
True
```

```
ghci> snd (True,"Ala")
```

```
"Ala"
```

Typ polimorficzny może podlegać ograniczeniom:

```
isNegative x = x < 0
```

```
ghci> :type isNegative
(Num a, Ord a) => a -> Bool
ghci> :t 0
0 :: Num a => a
ghci> :t 1.5
1.5 :: Fractional a => a
```

Tabela 7: Wybrane klasy typów

| Klasa typów | Opis |
|-------------------------|--|
| <code>Eq</code> | wartości typu są porównywalne |
| <code>Ord</code> | wartości typu są uporządkowane |
| <code>Num</code> | typ reprezentuje wartości numeryczne |
| <code>Integral</code> | jak <code>Num</code> , ale całkowitoliczbowe |
| <code>Fractional</code> | jak <code>Num</code> , ale nie tylko całkowitoliczbowe |
| <code>Show</code> | wartości typu można wyświetlić |

Tabela 8: Wybrane operatory i funkcje operujące na listach

| Wyrażenie | Wartość wyrażenia |
|----------------------|-------------------------------|
| <i>głowa:ogon</i> | lista złożona z głowy i ogona |
| <i>head lista</i> | głowa listy |
| <i>tail lista</i> | ogon listy |
| <i>length lista</i> | długość listy |
| <i>sum lista</i> | suma elementów listy |
| <i>maximum lista</i> | największy element listy |
| <i>elem e lista</i> | prawda, jeśli lista zawiera e |
| <i>reverse lista</i> | lista odwrócona |
| <i>lista++lista</i> | konkatenacja list |

Operator : konstruuje listę z głowy (*head*) i ogona (*tail*):

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

```
ghci> 3 : [4,5]
```

```
[3,4,5]
```

```
ghci> True : []
```

```
[True]
```

```
ghci> "Ala" : ["ma","kota"]
```

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

```
ghci> 1 : 2 : 3 : []
```

```
[1,2,3]
```

Definiując funkcję, która przyjmuje argument typu lista, można użyć dopasowania do wzorca:

Tabela 9: Przykładowe wzorce list

| Wzorzec | Opis |
|-----------------------|--------------------------|
| <code>[]</code> | lista pusta |
| <code>[x]</code> | 1-elementowa |
| <code>[x,y]</code> | 2-elementowa |
| <code>(x:xs)</code> | co najmniej 1-elementowa |
| <code>(x:y:ys)</code> | co najmniej 2-elementowa |

Funkcja zwracająca głowę listy:

```
head :: [a] -> a  
head (x:xs) = x
```

```
ghci> head "Ała ma kota"
```

```
'A'
```

```
ghci> head []
```

```
*** Exception: Non-exhaustive patterns
```

Funkcja zwracająca długość listy:

```
length :: [a] -> Int
length []      = 0
length (x:xs) = 1 + length xs
```

```
length [3,4]
= 1 + length [4]
= 1 + (1 + length [])
= 1 + (1 + 0)
= 2
```

Funkcja zwracająca sumę elementów listy:

```
sum :: Num a => [a] -> a
```

```
sum [] = 0
```

```
sum (x:xs) = x + sum xs
```

```
sum [3,4]
```

```
= 3 + sum [4]
```

```
= 3 + (4 + sum [])
```

```
= 3 + (4 + 0)
```

```
= 7
```

Funkcja zwracająca największy element listy:

```
maximum :: Ord a => [a] -> a
maximum [x]                = x
maximum (x:y:ys) | x > y   = maximum (x:ys)
                    | otherwise = maximum (y:ys)
```

```
maximum [2,1,3,0]
= maximum [2,3,0]
= maximum [3,0]
= maximum [3]
= 3
```

Funkcja sprawdzająca, czy element należy do listy:

```
elem :: Eq a => a -> [a] -> Bool
elem x []                = False
elem x (y:ys) | x == y   = True
                  | otherwise = elem x ys
```

```
elem 'c' "abcd"
= elem 'c' "bcd"
= elem 'c' "cd"
= True
```

Funkcja zwracająca listę odwróconą:

```
reverse :: [a] -> [a]
reverse l = rev l []
           where rev []      acc = acc
                  rev (x:xs) acc = rev xs (x:acc)
```

```
reverse "abc"
= rev "abc" []
= rev "bc"  ('a':[])
= rev "c"   ('b': 'a':[])
= rev []    ('c': 'b': 'a':[])
= "cba"
```

Operator konkatenciji list:

```
(++) :: [a] -> [a] -> [a]
```

```
[] ++ ys      = ys
```

```
(x:xs) ++ ys = x : (xs ++ ys)
```

```
"ab" ++ "cde"
```

```
= 'a' : ("b" ++ "cde")
```

```
= 'a' : ('b' : ([] ++ "cde"))
```

```
= 'a' : ('b' : "cde")
```

```
= "abcde"
```

Ćwiczenia podsumowujące:

1. Przeanalizuj złożoność operatora konkatencji. Czy lepiej, żeby ten operator był lewostronnie, czy prawostronnie łączny:

$$\begin{array}{ccc} & ("abc"++"de")++"fg" & \\ "abc"++"de"++"fg" = & & ? \\ & "abc"++("de"++"fg") & \end{array}$$

2. Napisz funkcję `concat :: [[a]] -> [a]`, konkatenującą listy na podanej liście:

```
ghci> concat ["ab",[],"c"]  
"abc"
```

3. Napisz operator `(!!) :: [a] -> Int -> a`, zwracający element pod podanym indeksem listy:

```
ghci> "Ala" !! 1  
'l'
```