Haskell – wprowadzenie

fromIntegral
to Integral

Inspiracja

Materiały do ćwiczeń będą oparte w dużej mierze na materiałach dr. Sławomira Bakalarskiego (do tego samego kursu).

Interpretery Haskella

- Będziemy korzystać z ghc – The Glasgow Haskell Compiler.
- Można zainstalować ze strony internetowej www.haskell.org lub np. przez Package Manager jeśli używają Państwo systemu Linux.
- Będziemy korzystać z interaktywnej wersji ghc, czyli ghci.
- W ostateczności można się uciec do wersji online, np. https://replit.com/languages/haskell.

Podstawowe idee

Języki funkcyjne nie przypominają C, C++, Pythona czy Javy.

- Nie używamy zmiennych (ale możemy definiować stałe).
- Nie używamy pętli (ale mamy do dyspozycji rekurencję).
- Będziemy się zajmować definiowaniem **funkcji**, które będą wykonywać to, co chcemy.
- Bardzo ważne będzie składanie funkcji (wbudowanych oraz samodzielnie zdefiniowanych).
- Funkcje mogą mieć wiele argumentów, którymi mogą być liczby, ale też ciągi znaków, listy czy **inne funkcje**.
- Funkcję n zmiennych z ustalonymi wartościami na $n - k$ argumentach możemy traktować jak funkcję k zmiennych.

Przykład funkcji

Poniższa funkcja dodaje do liczby (typu Int) jedynkę:

```
dodajJeden :: Int -> Int
dodajJeden x = x + 1
```

- Funkcje nazywamy z małej litery.
- Zaczynamy od podania **sygnatury funkcji** (jakiego typu są przyjmowane argumenty oraz zwracana wartość; Haskell jest silnie typowany).
- Po podaniu sygnatury podajemy definicję funkcji.

Korzystanie ze zdefiniowanych funkcji

- Tworzymy plik z rozszerzeniem *.hs, który chcemy skompilować (np. zawierający definicję funkcji dodajJeden ze slajdu wyżej).
- Uruchamiamy ghci.
- Komendą `:l <nazwaPliku>` (wystarczy bez rozszerzenia .hs) ładujemy nasz plik.
- Jeśli plik się kompiluje dostajemy dostęp do zdefiniowanych funkcji, w przykładzie z funkcją dodajJeden możemy teraz np. wywołać `dodajJeden 2` i otrzymać 3.

Podstawowe typy

Przykłady typów występujących w Haskellu:

- Int – "standardowy" int znany z innych języków.
- Integer – liczba całkowita (bez ograniczeń z dokładnością do skończonej pamięci).
- Float, Double – typy zmiennoprzecinkowe.
- Char – pojedynczy znak, używamy pojedynczych '.
- String – ciąg znaków, realizowany jako lista elementów typu Char, używamy ".
- Bool – True lub False.

Typy c.d.

Nazwy typów pisane są z dużej litery, jeśli nie określimy typu stałej wprost, Haskell spróbuje "zgadnąć" o co nam chodziło, np.

```
cztery = 4
```

spowoduje powstanie stałej cztery typu **Integer** (nie **Int**!). Typ możemy sprawdzić w ghci używając `:t <obiekt>`, w szczególności możemy sprawdzać typy nie tylko stałych, ale też funkcji. Jeśli chcemy wymusić typ stałej możemy to zrobić podobnie, jak przy definicji funkcji:

```
cztery :: Int
cztery = 4
```

Po każdej zmianie pliku nie musimy wpisywać `:l <nazwaPliku>`, wystarczy `:r`, który załaduje ponownie plik ładowany wcześniej. Ponieważ w ghci Ctrl-C przerywa wykonywanie aktualnie wykonywanej komendy wewnątrz ghci, aby wyjść używamy `:q` (lub Ctrl-D).

Funkcje w Haskellu

Spójrzmy na minimalnie bardziej złożoną definicję funkcji:

```
dodajTrzy :: Int -> Int -> Int -> Int
dodajTrzy x y z = x + y + z
```

Jest to oczywiście funkcja przyjmująca trzy argumenty typu Int, która zwraca wartość typu Int.

Funkcje w Haskellu

Spójrzmy na minimalnie bardziej złożoną definicję funkcji:

```
dodajTrzy :: Int -> Int -> Int -> Int
dodajTrzy x y z = x + y + z
```

Jest to oczywiście funkcja przyjmująca trzy argumenty typu Int, która zwraca wartość typu Int.
Równie dobrą interpretacją jest jednak również:
Jest to funkcja przyjmująca jako argument Int oraz zwracająca funkcję, która przyjmuje dwie wartości typu Int oraz zwraca wartość typu Int. Innymi słowy, powyższa sygnatura oraz sygnatura

```
dodajTrzy :: Int -> (Int -> Int -> Int)
```

czy nawet

```
dodajTrzy :: Int -> (Int -> (Int -> Int))
```

to w Haskellu dokładnie to samo (o czym można samemu się przekonać używając `:t`).

Funkcje w Haskellu c.d

Z powyższym przykładem można pójść jeszcze dalej:

- Funkcja `(dodajTrzy 7)` jest funkcją dwuargumentową, która przyjmuje dwie wartości typu Int i zwraca jako wynik ich sumę powiększoną o 7 (typu Int).
- Funkcja `(dodajTrzy 64 36)` jest funkcją jednoargumentową, która przyjmuje wartość typu Int i zwraca jako wynik tę wartość powiększoną o 100 (typu Int).

Odpowiada to wprost traktowaniu funkcji wielu zmiennych z ustalonymi niektórymi argumentami jako funkcji mniejszej liczby zmiennych. Jeśli mamy funkcję $f : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ daną przez $f(x, y, z) = x + y + z$, to przez $f(7, \cdot, \cdot)$ rozumiemy funkcję dwuargumentową, z $\mathbb{Z} \times \mathbb{Z}$ w $\mathbb{Z}$, która dana jest "takim samym wzorem" co f, z dokładnością do ustalenia pierwszego argumentu na 7. Podobnie $f(64, 36, \cdot)$ jest funkcją z $\mathbb{Z}$ w $\mathbb{Z}$.
Ponownie, możemy się o tym przekonać w ghci za pomocą `:t`.

Operatory

Operatory, które mamy do dyspozycji:

- Matematyczne: +, -, /, *;
- Porównań: >, >=, <, <=, ==, /=;
- Logiczne: &&, ||;

Stosowane infiksowo, np. 2+3, 3.0/4.0, ale również:

- `mod` – reszta modulo;
- `div` – dzielenie całkowite;
- `not` – zaprzeczenie logiczne

Stosowane prefiksowo, np. `mod 13 10`.
Operatory infiksowe można stosować prefiksowo biorąc je w nawias, np. `(*) 2 3`, a operatory prefiksowe infiksowo używając ‘ (klawisz z tyldą, nie cudzysłów!), np. `13 ‘mod‘ 10`.

Funkcje przyjmujące funkcje

Jak już zostało wspomniane, funkcje poza liczbami mogą też przyjmować inne funkcje np.

```
sumaWartosci :: (Int -> Int) -> (Int -> Int) -> Int ->
 Int -> Int
sumaWartosci f g x y = (f x) + (g y)
```

jest funkcją, która przyjmuje dwie jednoargumentowe funkcje oraz dwie wartości, wykonuje podane funkcje na podanych wartościach (pierwszą na pierwszej, drugą na drugiej) i dodaje wyniki.

Instrukcje warunkowe

Przykładową funkcję

$$f = \begin{cases} -x^2, & \text{dla } x < 0 \\ x^2, & \text{w przeciwnym przypadku} \end{cases}$$

możemy zrealizować na parę sposobów (pomijam sygnaturę):

```
-- dozory (guards)
f x | x < 0 = - x * x
    | otherwise = x * x

-- dozory (guards), w jednej linijce
f x | x < 0 = - x * x | otherwise = x * x

-- "standardowy" if-then-else
f x = if x < 0 then - x * x else x * x
```

Instrukcje warunkowe c.d

Innym przykładem może być dopasowywanie do wzorca:

```
ocena :: Double -> String
ocena 2.0 = "niezaliczone"
ocena 5.0 = "brawo!"
ocena x = "wpisane masz " ++ show x
```

Jeśli nie potrzebujemy wartości argumentu, możemy użyć znaku "_":

```
ocena :: Double -> String
ocena 2.0 = "niezaliczone"
ocena 5.0 = "brawo!"
ocena _ = "inne"
```

Instrukcje warunkowe – uwagi

- Jeśli używamy "if", **zawsze** musimy użyć "else".
- Stosując dozory (guards) lub dopasowywanie do wzorca możemy wypisać dowolnie dużo przypadków.
- Jeśli używamy dozorów, to "otherwise" jest opcjonalne, podobnie jak stosowanie w dopasowywaniu do wzorca na końcu takiego, który "pasuje do wszystkiego". Jeśli jednak podczas obliczania funkcji trafimy na nieobsłużoną sytuację, wykonywanie zakończy się błędem.
- Przypadki czytane są "od góry", po natrafieniu na pierwszy pasujący to właśnie on jest wybierany.

Rekurencja

Używając dopasowania do wzorca możemy wykorzystywać funkcję, którą właśnie definiujemy (to my musimy zadbać, żeby miało to sens), co bezpośrednio umożliwia nam stosowanie definicji rekurencyjnych:

```
silnia :: Integer -> Integer
silnia 0 = 1
silnia n = n * silnia (n-1)
```

Jeśli jednak w ostatniej linijce zamiast $n-1$ użyjemy $n+1$ lub też zamienimy warunki miejscami (znajdywany jest pierwszy pasujący od góry!), to wykonywanie np. `silnia 5` nigdy się nie zakończy.

Listy

Jeśli T jest typem, to $[T]$ oznacza listę elementów typu T.

- `[]` oznacza listę pustą.
- Listy mogą zawierać tylko elementy jednego rodzaju (są *jednorodne*).
- Listy można konkatenować operatorem `++` (w szczególności String jest listą elementów typu Char!). O(n)
- Do listy możemy dodać element na początek operatorem `:`, np. `1:[2,3]` oraz `1:2:3:[]` dadzą nam jako wynik listę `[1,2,3]`. O(1)
- Możemy się odwołać do konkretnego elementu listy po indeksie operatorem `!!`, np. `[1,2,3] !! 1` da nam wynik 2 (listy indeksowane są od zera!). O(n)

Krotki

Jeśli T_1, T_2 są typami, to (T_1, T_2) oznacza parę o elementach odpowiednio T_1 i T_2. Analogicznie możemy tworzyć krotki większej długości. Przykładowo `(Int,Double)` czy `(Int,Double,Integer)`.

- Wywołanie `fst (a,b)` zwraca pierwszy element pary, czyli a.
- Wywołanie `snd (a,b)` zwraca drugi element pary, czyli b.

Listy c.d.

- `[0..1000]` utworzy listę zawierającą wszystkie liczby naturalne nie większe niż 1000.
- `[0,2..1000]` utworzy listę zawierającą wszystkie naturalne liczby parzyste nie większe niż 1000.
- `['A'..'Z']` utworzy listę zawierającą duże litery alfabetu, po kolei.
- `['A','C'..'Z']` utworzy listę zawierającą co drugą literę alfabetu, po kolei.
- `[1,2..]` utworzy listę wszystkich dodatnich liczb naturalnych.
- `[10,20...]` utworzy listę wszystkich dodatnich wielokrotności liczby 10.

Ważnym szczególnie w kontekście dwóch ostatnich list jest fakt, że Haskell jest leniwy, tzn. liczy dopiero wtedy, kiedy musi, więc na przykład `[0,10..] !! 5` poprawnie zwróci 50, nie zostanie policzona cała nieskończona lista.

Listy c.d.

O(1)? O(n)

- Funkcje `head` i `last` zwracają odpowiednio pierwszy i ostatni element listy.

O(n) O(n)

- Funkcje `init` i `tail` zwracają odpowiednio wszystkie elementy poza ostatniem oraz wszystkie poza pierwszym.
- Wywołanie `take n list` zwraca *n* pierwszych elementów listy.
- Wywołanie `drop n list` opuszcza *n* pierwszych elementów listy.

`head`, `last`, `init` oraz `tail` wywołane na pustej liście zwrócą błąd.

Przykład połączenia list i dopasowania do wzorca

Przykład funkcji, która zareaguje inaczej dla listy pustej, jednoelementowej, dwuelementowej oraz pozostałych:

```
lista :: [Int] -> String
lista [] = "Lista jest pusta"
lista (x:[]) = "Lista zawiera tylko " ++ show x
lista (x:y:[]) = "Lista zawiera tylko " ++ show x ++
                 " oraz " ++ show y
lista (x:xs) = "Lista zaczyna sie od " ++ show x ++
               ", reszta to " ++ show xs
```

where oraz let

Klauzula `where` umożliwia stosowanie funkcji/stałych, których potrzebujemy tylko lokalnie. Przykład funkcji obliczającej Symbol Newtona $\binom{n}{k}$ i wykorzystującej `where`:

```
{- Symbol Newtona n po k -}
binomial :: Int -> Int -> Int
binomial n k = fact n `div`(fact k * fact (n - k))
               where
                 fact 0 = 1
                 fact n = n * fact (n - 1)
```

where oraz let c.d.

Analogiczny przykład, wykorzystujący `let`:

```haskell
{- Symbol Newtona n po k -}
binomial :: Int -> Int -> Int
binomial n k = let fact 0 = 1
                   fact n = n * fact (n - 1)
               in  fact n `div`(fact k * fact (n - k))
```

O różnicach między `where` oraz `let` można przeczytać dokładniej na stronie:
http://learnyouahaskell.com/syntax-in-functions#let-it-be

Polecane strony (dr S. Bakalarski)

- http://learnyouahaskell.com/
- http://book.realworldhaskell.org/
- https://hoogle.haskell.org/

Haskell – wprowadzenie c.d.

Inspiracja

Materiały do ćwiczeń będą oparte w dużej mierze na materiałach dr. Sławomira Bakalarskiego (do tego samego kursu).

Klasy typów – wprowadzenie

W Haskellu istnieje możliwość definiowania funkcji (i stałych!) bardziej ogólnie, niż dla jednego wybranego typu. Można się natknąć na to samemu używając `:t`, na przykład na dodawaniu. Możemy przeczytać, że:

```
(+) :: Num a => a -> a -> a
```

`Num` oznacza tutaj klasę typów, w tym konkretnych wypadku chodzi o klasę typów, na których można wykonywać działania arytmetyczne. Przykładami typów z klasy `Num` są `Int`, `Integer` czy `Double`. W Haskellu klasy typów mogą wymuszać zdefiniowanie odpowiedniego działania w odpowiednim kontekście. Czasem, mimo, że coś nie jest wymagane jest "oczekiwane", na przykład rozdzielność mnożenia względem dodawania w klasie `Num` (szczegóły dla konkretnych klas można sprawdzić w dokumentacji, np. Hoogle).

Klasy typów – wprowadzenie

Klasy mogą być definiowane w oderwaniu od innych klas lub mogą zachodzić między nimi relacje, np. `Fractional` jest podklasą `Num` (wymaga zdefiniowania operatora "/"), a `Floating` jest podklasą `Fractional` (wymaga dodatkowo zdefiniowania `exp`). Kompilator pilnuje, czy możemy wykonać operację, którą chcemy wykonać w danym kontekście. Gdybyśmy na przykład chcieli skompilować funkcję

```
napiszDodawanie :: Num a => a -> a -> String
napiszDodawanie x y = "Liczba " ++ show x ++
 " powiekszona o " ++ show y ++ " daje nam " ++ show (x+y)
```

dostaniemy błąd, ponieważ od liczby nie wymagamy możliwości przedstawienia jako `String`.

Klasy typów – wprowadzenie

Kompilator podpowie nam też, że możliwym rozwiązaniem problemu jest dorzucenie klasy `Show` w definicji. Istotnie,

```
napiszDodawanie :: (Num a, Show a) => a -> a -> String
napiszDodawanie x y = "Liczba " ++ show x ++
 " powiekszona o " ++ show y ++ " daje nam " ++ show (x+y)
```

jest już poprawną definicją. Na podobne problemy możemy natknąć się porównując, czy obiekty są takie same (klasa `Eq`) lub czy jeden jest większy od drugiego (Klasa `Ord`, jest w szczególności podklasą `Eq`).

Klasy typów – wprowadzenie

Jeśli będziemy definiować własne funkcje, ale nie napiszemy wprost sygnatury, to Haskell będzie "zgadywał" typy, nie zostawi "najbardziej ogólnego przypadku". Np. definicja

```
dodawanie = (+)
```

spowoduje powstanie funkcji

```
dodawanie :: Integer -> Integer -> Integer
```

Jest to zachowanie analogiczne do zgadywania typu stałej. Pisząc funkcje czy stałe samemu dobrym pomysłem może być więc pisanie sygnatury za każdym razem, ponieważ nie zyskujemy automatycznie pełnej ogólności nie pisząc nic. Możemy jednak sami zdefiniować

```
dodawanie :: Num a => a -> a -> a
dodawanie :: Integer -> Integer -> Integer
```

Klasy typów – wprowadzenie

Warto zwrócić jeszcze uwagę, że w danej sygnaturze, jeśli nastąpiło dopasowanie np. do a jakiegoś typu, to musi się ono zgadzać na każdym wystąpieniu a. Na przykład, jeśli napiszemy funkcję

```
wypiszRzeczy :: Show a => a -> a -> String
wypiszRzeczy x y = "Najpierw " ++
 show x ++ ", potem " ++ show y
```

to nie zadziała ona wywołana na argumentach `Int` oraz `Integer` mimo, że oba są w klasie `Show`. Możemy jednak dostać taką funkcjonalność:

```
wypiszRzeczy :: (Show a, Show b) => a -> b -> String
wypiszRzeczy x y = "Najpierw " ++
 show x ++ ", potem " ++ show y
```

Klasy typów – wprowadzenie

Możemy też definiować funkcje używając **zupełnie dowolnego** typu, nie musimy się zawężać do żadnej klasy. Przykładem jest wbudowana funkcja składania

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

o definicji odpowiadającej standardowemu składaniu funkcji, tzn.

$$(f \circ g)(x) = f(g(x))$$

przy standardowych założeniach co do dziedziny i przeciwdziedziny (które widać odzwierciedlone w sygnaturze).

Gdzie

Przy tworzeniu list bardzo pomocny może być operator "gdzie". Za pomocą definicji

```
kwadraty :: [Int]
kwadraty = [x * x | x <- [1..10]]
```

utworzymy listę kwadratów pierwszych 10 dodatnich liczb naturalnych. Odpowiada to oczywiście zapisowi matematycznemu

$$\{x^2 \mid x \in \{1, \ldots, 10\}\}.$$

Podobnie możemy uzyskać listy spełniające więcej warunków, na przykład kwadraty tylko liczb parzystych:

```
kwadraty2 :: [Int]
kwadraty2 = [x * x | x <- [1..10], even x]
```

Inne funkcje na listach

Przykłady innych funkcji związanych z listami:

- `repeat` – tworzy nieskończona listę powtarzając podany argument
- `cycle` – tworzy nieskończona listę powtarzając podaną listę
- `elem <obiekt> <lista>` – sprawdza, czy obiekt jest na liście
- `filter <warunek> <lista>` – zwraca podlistę elementów spełniających warunek

Operacje na listach

Inspiracja

Materiały do ćwiczeń będą oparte w dużej mierze na materiałach dr. Sławomira Bakalarskiego (do tego samego kursu).

map

Funkcja `map` o sygnaturze

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

przyjmuje funkcję f oraz listę $(x_1, \ldots, x_n)$ i zwraca listę $(f(x_1), \ldots, f(x_n))$. Przykładowo:

```
map (10+) [1..10]
map (10*) [1..10]
map (\x->x*x) [1..10]
```

zwrócą kolejno

```
[11,12,13,14,15,16,17,18,19,20]
[10,20,30,40,50,60,70,80,90,100]
[1,4,9,16,25,36,49,64,81,100]
```

filter

Funkcja `filter` o sygnaturze

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

przyjmuje funkcję f zwracającą wartość logiczną oraz listę $(x_1, \ldots, x_n)$ i zwraca listę tylko tych elementów z oryginalnej listy, na których f zwróciła prawdę. Przykładowo:

```
filter (odd) [1..10]
filter (\x->x `mod` 3 == 1) [1..10]
filter (>=5) [1..10]
```

zwrócą kolejno

```
[1,3,5,7,9]
[1,4,7,10]
[5,6,7,8,9,10]
```

filter na listach nieskończonych

Używając filter możemy działać na listach nieskończonych, na przykład `filter (even) [1..]` jest poprawną listą nieskończoną, na której możemy operować; przykładowo `take 10 (filter (even) [1..])` zwróci poprawnie `[2,4,6,8,10,12,14,16,18,20]`. Problematyczne jednak mogą być wywołania typu `filter (<5) [1..]`. W przypadku list nieskończonych Haskell oblicza kolejny element dopiero wtedy, kiedy musi, cztery pierwsze elementy wyniku to więc 1, 2, 3 i 4. Jeśli jednak będziemy chcieli skorzystać z piątego elementu listy, Haskell nigdy go nie obliczy (choć będzie próbować), ponieważ będzie przeszukiwał kolejne liczby całkowite (i nigdy nie znajdzie kolejnej, która spełnia warunek).

takeWhile/dropWhile

Przydatną funkcją w sytuacji powyżej może być `takeWhile`, która pobiera elementy z listy, póki spełniają podany warunek.
Przykładowo:

```
takeWhile (<10) [1..]
```

poprawnie zwróci skończoną listę

```
[1,2,3,4,5,6,7,8,9]
```

Analogiczna funkcja `dropWhile` działa podobnie, jednak ignoruje elementy zamiast je pobierać, na przykład

```
dropWhile (<5) [1..10]
```

zwraca

```
[5,6,7,8,9,10]
```

concatMap

Funkcja `concatMap` działa podobnie do `map`, jednak przyjmuje listę list, po zastosowaniu na każdej z nich podanej funkcji scala wyniki w jedną listę. Przykładowo:

```
concatMap (take 2) ["Ala","ma","kota"]
```

zwraca

```
"Almako"
```

iterate

Funkcja `iterate` o sygnaturze

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

przyjmuje funkcję f oraz element x, na którym można wywołać funkcję f i zwraca nieskończoną listę $(x, f(x), f(f(x)), \ldots)$. Przykładowo `iterate (+1) 0` zwróci nieskończoną listę wszystkich liczb naturalnych (od 0).

zip

Funkcja `zip` o sygnaturze

```
zip :: [a] -> [b] -> [(a, b)]
```

przyjmuje dwie listy i zwraca listę par postaci `(<elementLewej>,<elementPrawej>)`. Przykładowo

```
zip [1,2,3] "abc"
```

zwraca

```
[(1,'a'),(2,'b'),(3,'c')]
```

Jeśli listy nie są równej długości, elementy dłuższej "bez pary" z elementem listy krótszej są pomijane.

zipWith

Funkcja `zipWith` o sygnaturze

```
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

przyjmuje funkcję oraz dwie listy i zwraca listę elementów postaci $f(x, y)$, gdzie f jest podaną funkcją, a x oraz y to kolejne elementy list. Przykładowo

```
zipWith (+) [1,2,3] [4,5,6]
```

zwraca

```
[5,7,9]
```

Jeśli listy nie są równej długości, elementy dłuższej "bez pary" z elementem listy krótszej są pomijane.

Operacje na listach c.d.

Inspiracja

Materiały do ćwiczeń będą oparte w dużej mierze na materiałach dr. Sławomira Bakalarskiego (do tego samego kursu).

foldl/foldr

Funkcja `map` pozwala nam na wywołanie operacji na każdym elemencie listy. Co jednak, jeśli z listy argumentów chcemy otrzymać jeden wynik? Znamy już funkcje działające w ten sposób, na przykład `sum` czy `product`; o funkcjach `foldl` i `foldr` można myśleć jako o ich uogólnieniu.

Funkcja `foldl` wywołana na dwuargumentowym działaniu $\circ$, początkowej wartości x oraz liście elementów $(x_1, \ldots, x_n)$ zwróci nam wartość

$$(\ldots((x \circ x_1) \circ x_2) \circ \ldots) \circ x_n.$$

Funkcja `foldr` działa analogicznie, jednak od prawej; wywołana na takich samych argumentach zwróci nam wartość

$$x_1 \circ (x_2 \circ \ldots \circ (x_{n-1} \circ (x_n \circ x)) \ldots).$$

foldl/foldr

W praktyce początkowa wartość to często element neutralny działania $\circ$. Na przykład, sumę listy możemy zaimplementować jako:

```
sumaListyLewa :: Num a => [a] -> a
sumaListyLewa xs = foldl (+) 0 xs
```

Ponieważ dodawanie liczb rzeczywistych jest łączne i przemienne, definicja sumy jako

```
sumaListyPrawa :: Num a => [a] -> a
sumaListyPrawa xs = foldr (+) 0 xs
```

zwróci nam (w skończonych przypadkach) te same wyniki.

foldl/foldr

Funkcje `foldl` i `foldr` są bardzo uniwersalne. Podobnie, jak w przypadku `map`, używając ewentualnie zdefiniowanych przez nas funkcji jesteśmy w stanie definiować złożone zachowania używając funkcji dwuargumentowej i listy. Jeśli chcielibyśmy na przykład sumować punkty na płaszczyźnie zamiast liczb, możemy zdefiniować funkcję

```
o :: Num a => (a,a) -> (a,a) -> (a,a)
o (a,b) (c,d) = (a+c,b+d)
```

Funkcją sumującą listę par po współrzędnych będzie wtedy `foldl o (0,0)` (oczywiście foldr da nam tutaj ten sam wynik). Działa to oczywiście dużo ogólniej, jeśli np. mielibyśmy zdefiniowany własny typ reprezentujący macierze oraz dwuargumentową operację mnożenia macierzy, jesteśmy w stanie w jednej linijce zdefiniować mnożenie całej listy macierzy.

Typy danych

Inspiracja

Materiały do ćwiczeń będą oparte w dużej mierze na materiałach dr. Sławomira Bakalarskiego (do tego samego kursu).

Nowe typy

Używając `type` możemy zdefiniować nową nazwę typu dla typu już istniejącego (w ten sposób zdefiniowany jest też `String`). Przykładowo:

```
type Imie = String
```

Po takiej definicji możemy używać nowego typu, na przykład

```
jan :: Imie
jan = "Jan"
```

Ponieważ wprowadziliśmy tak na prawdę nową nazwę dla istniejącego już typu, w funkcjach używających `String` możemy użyć zdefiniowanego w taki sposób typu `Imie`. Podobnie funkcja, która używa `[Char]` działa dla `String`.

Nowe typy

Używając `newtype` możemy zdefiniować nowy typ za pomocą dokładnie jednego konstruktora przyjmującego dokładnie jeden argument. Na przykład:

```
newtype Imie2 = Imie2 String
```

Zdefiniowanego w ten sposób typu możemy używać np. tak:

```
janek :: Imie2
janek = Imie2 "Janek"
```

W przeciwieństwie do `type` nie możemy używać `Imie2` wymiennie ze `String`. W szczególności nie możemy wypisać nawet powyższej stałej na ekran, ponieważ typ `Imie2` nie jest w klasie `Show`.

Nowe typy

Używając `data` możemy zdefiniować nowy typ używając dowolnej liczby konstruktorów i dowolnej liczby pól. Na przykład:

```
data Rozmiar = S | M | L
maly :: Rozmiar
maly = S
```

lub

```
data Student = Student {imie :: String,
                        nazwisko :: String,
                        nrAlbumu :: Int
                        }
janKowalski = Student {imie="Jan",
 nazwisko="Kowalski", nrAlbumu=1234567}
```

Instancje klasy

Pomocne (chociażby dla możliwości łatwego wyświetlania) jest należenie typu do klasy `Show`. Aby to osiągnąć możemy użyć operatora `instance`:

```
instance Show Rozmiar
    where
    show S = "S"
    show M = "M"
    show L = "L"
```

Aby sprawdzić, jakie funkcje musimy zdefiniować, żeby poprawnie użyć operatora `instance` możemy skorzystać np. z Hoogle (w klasie Show jest prosto, wymagana jest tylko jedna funkcja, w klasie `Num` jest już ich dużo więcej: dodawanie, mnożenie itd.).

deriving

W prostych przypadkach Haskell może "zgadnąć" sensowne implementacje `show` czy ==. Używamy w tym celu operatora `deriving`:

```
data Rozmiar = S | M | L deriving Show
```

W powyższym przypadku `show` zostanie zdefiniowane dokładnie tak, jak na poprzednim slajdzie zrobiliśmy to ręcznie. Możemy też podać więcej, niż jedną klasę, np.:

```
deriving (Show,Eq)
```

W prostych przypadkach, np. dla typu `Student` zdefiniowanego wcześniej Haskell będzie (odpowiednio) wypisywał pola (nazwy i wartości) oraz sprawdzał, czy odpowiadające pola w dwóch podanych instancjach mają te same wartości. Jeśli to, co zostanie przypisane nam nie odpowiada, możemy oczywiście zdefiniować pożądane zachowanie sami.

Przykład: liczby naturalne

Możemy zdefiniować liczby naturalne w następujący sposób:

```
data Naturalne = Zero | Nastepnik Naturalne
 deriving (Show,Eq)
```

Przykładowymi liczbami naturalnymi są wtedy:

```
zero = Zero
jeden = Nastepnik Zero
dwa = Nastepnik (Nastepnik Zero)
```

Zdefiniowanych typów możemy używać w funkcjach, na przykład:

```
natToInt :: Naturalne -> Integer
natToInt Zero = 0
natToInt (Nastepnik x) = (natToInt x) + 1
```

Przykład: drzewo binarne

Możemy zdefiniować drzewo binarne w następujący sposób:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Drzewo zawierające w wierzchołkach liczby całkowite, złożone tylko z korzenia, w którym jest zero możemy wtedy zdefiniować następująco:

```
drzewo :: Tree Integer
drzewo = Node 0 Empty Empty
```