Programowanie Funkcyjne

Rafał Włodarczyk

INA 4, 2025

Contents

1 Lecture I - Haskell **1**
1.1 Instalacja Haskella - GHCup . . . 2
1.2 Problem wczytywania zmiennych . . . 2
1.3 Podstawowe typy . . . 3
1.4 λ-wyrażenie . . . 4

2 Lecture II **5**
2.1 Tworzenie własnych typów . . . 6
2.2 Pary . . . 6
2.3 Listy . . . 6

1 Lecture I - Haskell

Haskell = leniwy język funkcyjny

functions are first class objects

$$x \to \boxed{\text{f}} \to f(x)$$

Rozważmy fragment kodu:

```
int c = 2;
int f(int x) {
    return (c*x);
}
```

Ta funkcja nie jest czysta - wykorzystuje swoje środowisko.

Rozważmy fragment kodu:

```
int f(int x) {
    printf("Hello");
    return (2 * x);
}
```

Zadziała na środowisku zewnętrznym - to nie jest czysta funkcja

```
int f(int x) {
    return (2*x);
}
```

Nie wpływa na otoczenie, nie wykorzystuje, ani nie zmienia występujacych obiektów.
Języki funkcyjne operują na czystych funkcjach.

1.1 Instalacja Haskella - GHCup

ghci - interaktywna konsola GHC (Glasgow Haskell Compilers)

```
ghci> 1 + 2
3
ghci> :? # help
ghci> :q # exit
ghci> :load file.hs
```

1.2 Problem wczytywania zmiennych

```
!! readInt() {...} :: Int
```

Definition. Funkcja jednej zmiennej. .

Zdefiniujmy funkcję

```
w1.hs

f x = 1 + x*(1+x)

ghci>:load w1
ghci>f 1
3
ghci>:type f
f :: Num a => a -> a
ghci>:info Num
```

Podstawowy typ Num

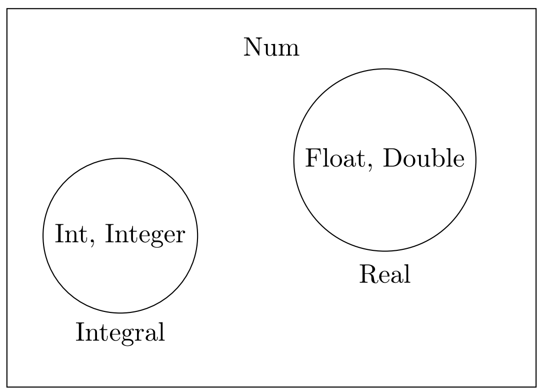
```
ghci> :info Num
type Num :: * -> Constraint
class Num a where
    (+) :: a -> a -> a
    (-) :: a -> a -> a
    (*) :: a -> a -> a
    negate :: a -> a
    abs :: a -> a
    signum :: a -> a
    fromInteger :: Integer -> a
```

```
    {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
        -- Defined in ‘GHC.Num’
instance Num Double -- Defined in ‘GHC.Float’
instance Num Float -- Defined in ‘GHC.Float’
instance Num Int -- Defined in ‘GHC.Num’
instance Num Integer -- Defined in ‘GHC.Num’
instance Num Word -- Defined in ‘GHC.Num’
```

1.3 Podstawowe typy

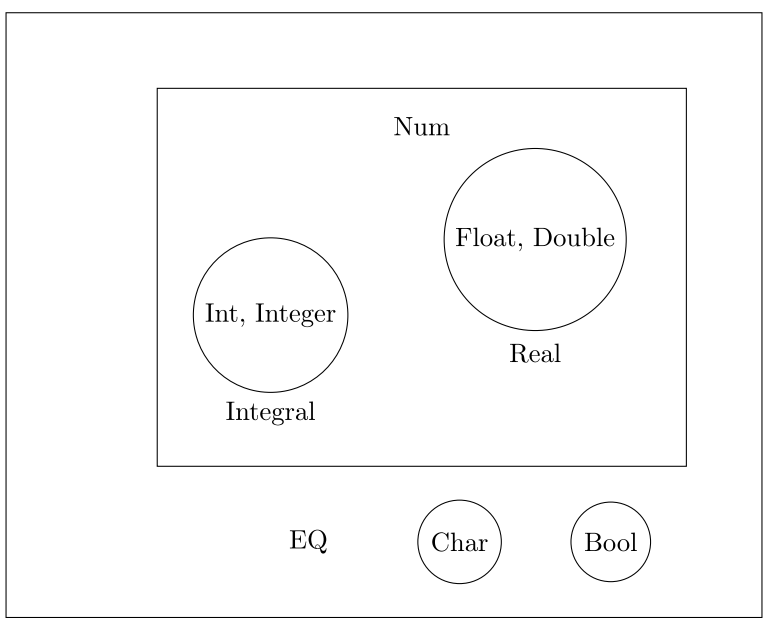
Int, Integer (unlimited size), Float, Double, Char, Bool

$$\text{Int, Integer} \in \text{Integral} \subseteq \text{Num}$$



Pod spodem działa Teoria Typowania Hindleya - Milnera

```
f(3.23 :: Double) # możemy explicite wymusić typ
```



Definition. w1.hs. Zdefiniujmy funkcje:

```
ghci> g x y = 1 + x * y
ghci> :type g
g :: (Fractional t1, Num a) => t1 -> t2 -> a
ghci> g x y = 1 + x * y
ghci> h = g (2::Int)
ghci> :t h
h :: Int -> Int
```

Z podobną sytuacją mieliśmy do czynienia przy potęgowaniu liczb kardynalnych:

$$\left|C^{B\times A}\right| = \left|\left(C^B\right)^A\right| \tag{1}$$

1.4 λ-wyrażenie

Definition. λ-wyrażenie (funkcja anonimowa).

$$(\lambda x \to \mathrm{expr})\,(t) = \mathrm{expr}\,[x \rightsquigarrow t]$$

Chcielibyśmy znaleźć:

$$\Psi : C^{B\times A} \to \left(C^B\right)^A \tag{2}$$

$$\Psi(t) = (\lambda a : A \to (\lambda b : B \to f(a,b))) \tag{3}$$

$$\Psi(f)(a) = (\lambda b : B \to f(a,b)) \tag{4}$$

Funkcję Ψ nazywamy funkcją curry. Wszyskie funkcje w Haskellu są poddane curryingowi.

Curry Haskell - Amerykański Logik z XX wieku.

Podstawowym narzędziem języków funkcyjnych jest rekursja.

Information. Silnia. Zapiszmy silnię w Haskellu. Najsilniejsze działanie w Haskellu to aplikacja funkcji na argumencie:

```
fact1 n = if n == 0 then 1
          else n * fact1 (n - 1);
```

else musi być w Haskellu - wynik zawsze musi być czymś.

Information. Pattern Matchings. Zapiszmy lepszą silnie:

```
fact2::Integer -> Integer
fact2 0 = 1
fact2 n = n * fact2(n-1)
```

Information. Case Expression. Zapiszmy za pomocą case expression:

```
fact3 n = case n of
          0 -> 1
          otheriwse -> n * fact3(n-1)
```

Information. Pseudozmienne. Zapiszmy pseudozmienne:

```
fact4 n = let y = n - 1 in
        if n == 0 then 1
        else n * fact4 y

fact4 n = (lambda y -> if n==0 then 1 else n * fact4 y)(n-1)
```

To jest język C

```
h x = x + x sin(x) sin^2 (x)
float h(float x) {
    float y = sin(x);
    return (x + x*y + y*y);
}
```

Równoważnik Haskellowy:

```
h x = (\y -> x + x*y + y*y)(sin x)
h x = let y = sin x in
      x + x*y + y*y
```

2 Lecture II

Zbadajmy identyczność:

```
id:: a -> a
id:: forall a => a -> a
```

Lambda kwantyfikuje typy, a nie zmienne - głębokie znaczenie polimorfizmu

$$\exp = (\lambda a : \mathrm{Typ} \to (a \to a)) \tag{5}$$
$$\exp(\mathrm{Int}) :: \mathrm{Int} \to \mathrm{Int} \tag{6}$$
$$\exp(\mathrm{Double}) :: \mathrm{Double} \to \mathrm{Double} \tag{7}$$

```
ghci> inc x = x + 1
ghci> :t inc
inc :: Num a => a -> a
```

$$\exp = (\forall a : \mathrm{Num} \to (a \to a)) \tag{8}$$

Zasada: *W linijce przed wyrażeniem opisuje jego typ.*

2.1 Tworzenie własnych typów

1. pary
2. listy

2.2 Pary

```
ghci> :t (13, 'a')
(13, 'a') :: Num a => (a, Char)
ghci> :t (13::Integer, 'a')
(13::Integer, 'a') :: (Integer, Char)
```

```
coll n = | n == 1 then 1
         | even n then call (div n 2)  -- n `div` 2
         | otherwise call (3*n + 1)

collatz :: (Int, Int) -> (Int, Int)
collatz (n, s) | n == 1 = (n, s)
               | even n = collatz (div n 2, s + 1)
               | otherwise = collatz (3 * n + 1, s + 1)
```

2.3 Listy

```
[a] =    elementów a
```

$$\{[a_1, \ldots, a_k] : a_1, \ldots, a_k, a_i \in a, k \in \mathbb{N}\} \tag{9}$$

```
[1,2,3]

[1,_]->[2, ]->[3, ]->[ ]
x_0: [x_1, x_2, ..., x_k] = [x_0, x_1, ... x_k]
[] <- lista pusta
[1,2,3] ~= 1:2:3:[]
konkatenacja dwóch list ++
[1,2,3] ++ [4,5,6] = [1,2,3,4,5,6]

concat :: [[a]] -> [a]
concat [[1,2,3],[5],[0,1]] = [1,2,3,4,0,1]

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

aplikacja funkcji do zmiennej ma najwyższy priorytet
f x = ...

[] ++ ys = ys
```

```
(x:xs)++ys = x:(xs++ys)
[1,2]++[] = 1:([2]++[]) = 1:(2:([]||[])) = [1,2]

head [] = error bad: pusta lista
head (x:_) = x

tail [] = []
tail (x:xs) = xs

-- mapowanie
[x_1,x_2,...x_k], f: a -> b
[fx_1,fx_2... fx_k]

map::(a->b)->[a]->[b]
map _ [] = []
-- ciąg którego głowa to jest x, a ogon to xs
map f (x:xs) = f x : map f xs

map(\x -> x^2)[1..10] -> [1^2, 2^2, ..., 100]

-- filtrowanie

filter::(a->Bool)->[a]->[a]
filter even [1..10] -> [2,4,6,8,10]

filter _ [] = []
filter p (x:xs) = if p x then x:filter p xs
                  else filter p xs

-- prototypy

list comprehension

[f x_1 x_2 x_3 | x_1 <- xs, x_2 <- ys, x_3 <- zs]
[f x_1 x_2 x_3 | x_1 <- xs, x_2 <- ys, x_1<x_2, x_3 <- zs]

# wszystkie trójki pitagorejskie do 100
[(x,y,z) | z <- [1..100] y<-[1..z], x<-[1..y], x^2+y^2==z^2, gcd(x,y)==1]
```