Programowanie Funkcyjne

Rafał Włodarczyk

INA 4, 2025

Contents

1 Lecture I - Haskell

Haskell = leniwy język funkcyjny

functions are first class objects

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

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
```

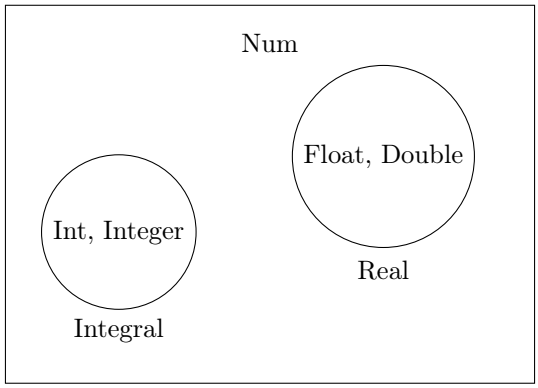
Podstawowy typ Num

```
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

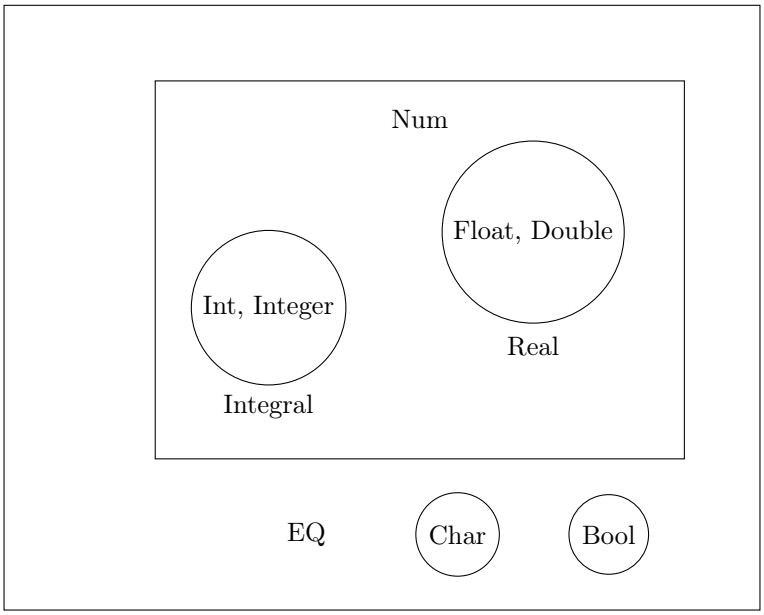
Int, Integer (unlimited size), Float, Double, Char, Bool

Int, Integer $\in$ Integral $\subseteq$ 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).

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

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
```