

Podstawy Programowania Funkcyjnego

Chlebowski Sz. Gajda A.

Zakład Logiki i Kognitywistyki
Instytut Psychologii
Uniwersytet im. Adama Mickiewicza

03.03.2020

1 Typy

2 Klasy

- Zaczynamy zawsze z małej litery, kolejne znaki są dowolne.
- Nie można używać nazw obiektów, które są predefiniowane.
- Konwencja: `xs`, `ns`, czyli nazwy z 's' na końcu oznaczają wiele obiektów (patrz listy).
- Komentarze:
 - komentarz w jednej linii: `--`,
 - komentarz bloku tekstu: `{- i -}`.

- <https://www.haskell.org/>
- <https://hackage.haskell.org/>
- <https://www.haskell.org/hoogle/>

Түрү

- Bardzo ogólnie mówiąc, przez *typ* rozumiemy pewien zbiór/kolekcję obiektów podobnego rodzaju.

- Bardzo ogólnie mówiąc, przez *typ* rozumiemy pewien zbiór/kolekcję obiektów podobnego rodzaju.
- Przykład:

- Bardzo ogólnie mówiąc, przez *typ* rozumiemy pewien zbiór/kolekcję obiektów podobnego rodzaju.
- Przykład:
 - Typ `Bool` zawiera dwa obiekty — dwie wartości logiczne: `True` i `False`.

- Bardzo ogólnie mówiąc, przez *typ* rozumiemy pewien zbiór/kolekcję obiektów podobnego rodzaju.
- Przykład:
 - Typ `Bool` zawiera dwa obiekty — dwie wartości logiczne: `True` i `False`.
 - Typ `Bool` -> `Bool` zawiera wszystkie funkcje przekształcające obiekt typu `Bool` w obiekt typu `Bool` (np. funkcja negacji — `not`).

- Jeśli chcemy zapisać, że obiekt o jest typu T , używamy notacji:

$o :: T$

- Jeśli chcemy zapisać, że obiekt o jest typu T , używamy notacji:

$$o :: T$$

- Rozważmy następujące przykłady:

- Jeśli chcemy zapisać, że obiekt `o` jest typu `T`, używamy notacji:
 - `o :: T`
- Rozważmy następujące przykłady:
 - `False :: Bool`

- Jeśli chcemy zapisać, że obiekt `o` jest typu `T`, używamy notacji:

`o :: T`

- Rozważmy następujące przykłady:

- `False :: Bool`

- `True :: Bool`

- Jeśli chcemy zapisać, że obiekt `o` jest typu `T`, używamy notacji:

`o :: T`

- Rozważmy następujące przykłady:

- `False :: Bool`
- `True :: Bool`
- `not :: Bool -> Bool`

- Jeśli chcemy zapisać, że obiekt `o` jest typu `T`, używamy notacji:

`o :: T`

- Rozważmy następujące przykłady:

- `False :: Bool`
- `True :: Bool`
- `not :: Bool -> Bool`
- `1 :: Int`

- Jeśli chcemy zapisać, że obiekt `o` jest typu `T`, używamy notacji:

`o :: T`

- Rozważmy następujące przykłady:

- `False :: Bool`
- `True :: Bool`
- `not :: Bool -> Bool`
- `1 :: Int`
- `[1,2] :: [Int]`

- Jeśli chcemy zapisać, że obiekt o jest typu T , używamy notacji:

$$o :: T$$

- Rozważmy następujące przykłady:

- `False :: Bool`
- `True :: Bool`
- `not :: Bool -> Bool`
- `1 :: Int`
- `[1,2] :: [Int]`

- Podstawową regułą pozwalającą na określenie typów wyrażeń bardziej złożonych jest reguła zastosowania funkcji:

$$\frac{f :: A \rightarrow B \quad e :: A}{f e :: B} \text{ app}$$

- Jeśli chcemy zapisać, że obiekt o jest typu T , używamy notacji:

$$o :: T$$

- Rozważmy następujące przykłady:

- `False :: Bool`
- `True :: Bool`
- `not :: Bool -> Bool`
- `1 :: Int`
- `[1,2] :: [Int]`

- Podstawową regułą pozwalającą na określenie typów wyrażeń bardziej złożonych jest reguła zastosowania funkcji:

$$\frac{f :: A \rightarrow B \quad e :: A}{f\ e :: B} \text{ app}$$

- Czy ta reguła coś ci przypomina? (pomyśl o logice)

- Stosując app możemy określić typ następujących wyrażeń:

- Stosując app możemy określić typ następujących wyrażeń:
 - `not False :: Bool`

- Stosując app możemy określić typ następujących wyrażeń:
 - `not False :: Bool`
 - `not True :: Bool`

- Stosując app możemy określić typ następujących wyrażeń:
 - `not False :: Bool`
 - `not True :: Bool`
 - `not (not False) :: Bool`

- Stosując app możemy określić typ następujących wyrażeń:
 - `not False :: Bool`
 - `not True :: Bool`
 - `not (not False) :: Bool`
- Używając reguły app, zbuduj drzewo określające typ wyrażenia:
`not (not (not True))`

- Stosując app możemy określić typ następujących wyrażeń:
 - `not False :: Bool`
 - `not True :: Bool`
 - `not (not False) :: Bool`
- Używając reguły app, zbuduj drzewo określające typ wyrażenia:
`not (not (not True))`
- Czy można, używając app, przypisać typ wyrażeniu:
`not 3`

- Bool zawiera dwa obiekty: True oraz False

- Bool zawiera dwa obiekty: True oraz False
- Char zawiera wszystkie pojedyncze symbole w systemie Unicode

- Bool zawiera dwa obiekty: True oraz False
- Char zawiera wszystkie pojedyncze symbole w systemie Unicode
- String zawiera wszystkie ciągi elementów typu Char, a zatem

`String = [Char]`

- Bool zawiera dwa obiekty: True oraz False
- Char zawiera wszystkie pojedyncze symbole w systemie Unicode
- String zawiera wszystkie ciągi elementów typu Char, a zatem
$$\text{String} = [\text{Char}]$$
- Int zawiera liczby całkowite, takie jak -100 , czy 999 . W GHC, wartości Int zawierają się w przedziale od -2^{29} do $2^{29} - 1$

- Bool zawiera dwa obiekty: True oraz False
- Char zawiera wszystkie pojedyncze symbole w systemie Unicode
- String zawiera wszystkie ciągi elementów typu Char, a zatem
$$\text{String} = [\text{Char}]$$
- Int zawiera liczby całkowite, takie jak -100 , czy 999 . W GHC, wartości Int zawierają się w przedziale od -2^{29} do $2^{29} - 1$
- Integer zawiera liczby całkowite, bez ograniczeń nakładanych na Int

- Bool zawiera dwa obiekty: True oraz False
- Char zawiera wszystkie pojedyncze symbole w systemie Unicode
- String zawiera wszystkie ciągi elementów typu Char, a zatem
$$\text{String} = [\text{Char}]$$
- Int zawiera liczby całkowite, takie jak -100 , czy 999 . W GHC, wartości Int zawierają się w przedziale od -2^{29} do $2^{29} - 1$
- Integer zawiera liczby całkowite, bez ograniczeń nakładanych na Int
- Float zawiera liczby w rozwinięciu dziesiętnym, takie jak -12.34 , czy 3.1415927

- Bool zawiera dwa obiekty: True oraz False
- Char zawiera wszystkie pojedyncze symbole w systemie Unicode
- String zawiera wszystkie ciągi elementów typu Char, a zatem
$$\text{String} = [\text{Char}]$$
- Int zawiera liczby całkowite, takie jak -100 , czy 999 . W GHC, wartości Int zawierają się w przedziale od -2^{29} do $2^{29} - 1$
- Integer zawiera liczby całkowite, bez ograniczeń nakładanych na Int
- Float zawiera liczby w rozwinięciu dziesiętnym, takie jak -12.34 , czy 3.1415927
- Double jest podobny do Float, możliwe jest jednak użycie większej ilości pamięci do przechowywania wartości tego typu

- Lista jest ciągiem elementów *tego samego* typu. Piszemy `[T]`, żeby oznaczyć typ listy, której elementami są obiekty typu `T`.

- Lista jest ciągiem elementów *tego samego* typu. Piszemy [T], żeby oznaczyć typ listy, której elementami są obiekty typu T.
- Przykłady:

- Lista jest ciągiem elementów *tego samego* typu. Piszemy [T], żeby oznaczyć typ listy, której elementami są obiekty typu T.
- Przykłady:
 - `[False,True,False] :: [Bool]`

- Lista jest ciągiem elementów *tego samego* typu. Piszemy [T], żeby oznaczyć typ listy, której elementami są obiekty typu T.
- Przykłady:
 - [False,True,False] :: [Bool]
 - ['a','b','c','d'] :: [Char]

- Lista jest ciągiem elementów *tego samego* typu. Piszemy [T], żeby oznaczyć typ listy, której elementami są obiekty typu T.
- Przykłady:
 - [False,True,False] :: [Bool]
 - ['a','b','c','d'] :: [Char]
 - ["One","Two","Three"] :: [String]

- Lista jest ciągiem elementów *tego samego* typu. Piszemy [T], żeby oznaczyć typ listy, której elementami są obiekty typu T.
- Przykłady:
 - [False,True,False] :: [Bool]
 - ['a','b','c','d'] :: [Char]
 - ["One","Two","Three"] :: [String]
 - [['a','b'],['c','d'],'e'] :: [[Char]]

- Krotka jest skończonym ciągiem elementów takich samych lub różnych typów. Piszemy (T_1, T_2, \dots, T_n) , żeby oznaczyć typ krotki, której elementami są obiekty typów T_1, T_2, \dots, T_n .

- Krotka jest skończonym ciągiem elementów takich samych lub różnych typów. Piszemy (T_1, T_2, \dots, T_n) , żeby oznaczyć typ krotki, której elementami są obiekty typów T_1, T_2, \dots, T_n .
- Przykłady:

- Krotka jest skończonym ciągiem elementów takich samych lub różnych typów. Piszemy (T_1, T_2, \dots, T_n) , żeby oznaczyć typ krotki, której elementami są obiekty typów T_1, T_2, \dots, T_n .
- Przykłady:
 - `(False, True) :: (Bool, Bool)`

- Krotka jest skończonym ciągiem elementów takich samych lub różnych typów. Piszemy (T_1, T_2, \dots, T_n) , żeby oznaczyć typ krotki, której elementami są obiekty typów T_1, T_2, \dots, T_n .
- Przykłady:
 - `(False, True) :: (Bool, Bool)`
 - `(False, 'a', True) :: (Bool, Char, Bool)`

- Krotka jest skończonym ciągiem elementów takich samych lub różnych typów. Piszemy (T_1, T_2, \dots, T_n) , żeby oznaczyć typ krotki, której elementami są obiekty typów T_1, T_2, \dots, T_n .
- Przykłady:
 - `(False, True) :: (Bool, Bool)`
 - `(False, 'a', True) :: (Bool, Char, Bool)`
 - `("Yes", True, 'a') :: (String, Bool, Char)`

- Krotka jest skończonym ciągiem elementów takich samych lub różnych typów. Piszemy (T_1, T_2, \dots, T_n) , żeby oznaczyć typ krotki, której elementami są obiekty typów T_1, T_2, \dots, T_n .
- Przykłady:
 - `(False, True) :: (Bool, Bool)`
 - `(False, 'a', True) :: (Bool, Char, Bool)`
 - `("Yes", True, 'a') :: (String, Bool, Char)`
 - `('a', (False, 'b')) :: (Char, (Bool, Char))`

- Krotka jest skończonym ciągiem elementów takich samych lub różnych typów. Piszemy (T_1, T_2, \dots, T_n) , żeby oznaczyć typ krotki, której elementami są obiekty typów T_1, T_2, \dots, T_n .
- Przykłady:
 - `(False, True) :: (Bool, Bool)`
 - `(False, 'a', True) :: (Bool, Char, Bool)`
 - `("Yes", True, 'a') :: (String, Bool, Char)`
 - `('a', (False, 'b')) :: (Char, (Bool, Char))`
 - `[('a', False), ('b', True)] :: [(Char, Bool)]`

- Funkcja jest przekształceniem obiektów typu A w obiekty typu B.
Typ funkcji z A w B oznaczamy przez $A \rightarrow B$.

- Funkcja jest przekształceniem obiektów typu A w obiekty typu B.
Typ funkcji z A w B oznaczamy przez $A \rightarrow B$.
- Przykłady:

- Funkcja jest przekształceniem obiektów typu A w obiekty typu B.
Typ funkcji z A w B oznaczamy przez $A \rightarrow B$.
- Przykłady:
 - `not :: Bool -> Bool`

- Funkcja jest przekształceniem obiektów typu A w obiekty typu B.
Typ funkcji z A w B oznaczamy przez $A \rightarrow B$.
- Przykłady:
 - `not :: Bool -> Bool`
 - `even :: Int -> Bool`

- Funkcja jest przekształceniem obiektów typu A w obiekty typu B.
Typ funkcji z A w B oznaczamy przez $A \rightarrow B$.
- Przykłady:
 - `not :: Bool -> Bool`
 - `even :: Int -> Bool`
 - `f1 :: (Int, Int) -> Int`

- Funkcja jest przekształceniem obiektów typu A w obiekty typu B.
Typ funkcji z A w B oznaczamy przez $A \rightarrow B$.
- Przykłady:
 - `not :: Bool -> Bool`
 - `even :: Int -> Bool`
 - `f1 :: (Int, Int) -> Int`
 - `f2 :: Int -> Int -> Int`

- Funkcja jest przekształceniem obiektów typu A w obiekty typu B.
Typ funkcji z A w B oznaczamy przez $A \rightarrow B$.
- Przykłady:
 - `not :: Bool -> Bool`
 - `even :: Int -> Bool`
 - `f1 :: (Int, Int) -> Int`
 - `f2 :: Int -> Int -> Int`
- Czym różni się f1 od f2?

- Każdą n -argumentową funkcję da się zapisać jako jednoargumentową funkcję wyższego rzędu (*currying*).

- Każdą n -argumentową funkcję da się zapisać jako jednoargumentową funkcję wyższego rzędu (*currying*).
- Rozważmy funkcję, mnożącą przez siebie trzy liczby:

```
mult :: (Int, Int, Int) -> Int  
mult x y z = x*y*z
```

- Każdą n -argumentową funkcję da się zapisać jako jednoargumentową funkcję wyższego rzędu (*currying*).
- Rozważmy funkcję, mnożącą przez siebie trzy liczby:

```
mult :: (Int, Int, Int) -> Int
mult x y z = x*y*z
```

- Deklarując jej typ, możemy założyć, że jako argument bierze ona krotkę trzech liczb.

- Z drugiej strony można sobie wyobrazić, że funkcja `mult` bierze jako argument jedną liczbę `x`, następnie zwraca funkcję `mult x`, która bierze jako argument kolejną liczbę `y` i zwraca funkcję `mult x y`, która w końcu bierze trzecią liczbę `z` i zwraca $x * y * z$.

- Z drugiej strony można sobie wyobrazić, że funkcja `mult` bierze jako argument jedną liczbę `x`, następnie zwraca funkcję `mult x`, która bierze jako argument kolejną liczbę `y` i zwraca funkcję `mult x y`, która w końcu bierze trzecią liczbę `z` i zwraca `x * y * z`.
- W tym ujęciu `mult` jest jednoargumentową funkcją wyższego rzędu. Jej typ można zatem zapisać w następujący sposób:

```
mult :: Int -> (Int -> (Int -> Int))  
mult x y z = x*y*z
```


- Z drugiej strony można sobie wyobrazić, że funkcja `mult` bierze jako argument jedną liczbę `x`, następnie zwraca funkcję `mult x`, która bierze jako argument kolejną liczbę `y` i zwraca funkcję `mult x y`, która w końcu bierze trzecią liczbę `z` i zwraca `x * y * z`.
- W tym ujęciu `mult` jest jednoargumentową funkcją wyższego rzędu. Jej typ można zatem zapisać w następujący sposób:

```
mult :: Int -> (Int -> (Int -> Int))  
mult x y z = x*y*z
```

- W uproszczonej notacji, którą będziemy stosować dalej, typ `mult` można zapisać jeszcze prościej:

```
mult :: Int -> Int -> Int -> Int
```

- Z drugiej strony można sobie wyobrazić, że funkcja `mult` bierze jako argument jedną liczbę `x`, następnie zwraca funkcję `mult x`, która bierze jako argument kolejną liczbę `y` i zwraca funkcję `mult x y`, która w końcu bierze trzecią liczbę `z` i zwraca `x * y * z`.
- W tym ujęciu `mult` jest jednoargumentową funkcją wyższego rzędu. Jej typ można zatem zapisać w następujący sposób:

```
mult :: Int -> (Int -> (Int -> Int))  
mult x y z = x*y*z
```

- W uproszczonej notacji, którą będziemy stosować dalej, typ `mult` można zapisać jeszcze prościej:

```
mult :: Int -> Int -> Int -> Int
```

- Wszystkie funkcje będziemy definiować jako funkcje jednoargumentowe.

- Typ, który zawiera zmienną jakiegoś typu, jest nazywany typem polimorficznym.

- Typ, który zawiera zmienną jakiegoś typu, jest nazywany typem polimorficznym.
- Rozważmy funkcję `length`, która liczy długość listy. Jej typ jest następujący:

```
length :: [a] -> Int
```

- Typ, który zawiera zmienną jakiegoś typu, jest nazywany typem polimorficznym.
- Rozważmy funkcję `length`, która liczy długość listy. Jej typ jest następujący:

```
length :: [a] -> Int
```

- Jak należy się spodziewać, funkcja ta może być jednorodnie zdefiniowana dla list obiektów dowolnego typu (oznaczonego zmienną `a`), jest zatem *funkcją polimorficzną*.

- Typ, który zawiera zmienną jakiegoś typu, jest nazywany typem polimorficznym.
- Rozważmy funkcję `length`, która liczy długość listy. Jej typ jest następujący:

```
length :: [a] -> Int
```

- Jak należy się spodziewać, funkcja ta może być jednorodnie zdefiniowana dla list obiektów dowolnego typu (oznaczonego zmienną `a`), jest zatem *funkcją polimorficzną*.
- Rozważmy następujące przykłady:

```
fst  :: (a,b) -> a
head :: [a]  -> a
take :: Int  -> [a] -> [a]
zip  :: [a]  -> [b] -> [(a,b)]
```

- Typy przepętnione przypominają typy polimorficzne — zawierają zmienną określającą typ. Poza tym zawierają ograniczenie dotyczące tej zmiennej.

- Typy przepełnione przypominają typy polimorficzne — zawierają zmienną określającą typ. Poza tym zawierają ograniczenie dotyczące tej zmiennej.
- Jak powiedzieliśmy wcześniej, dysponujemy różnymi typami określającymi sposoby reprezentowania liczb (Int, Float itd.).

Chcemy jednak, żeby niektóre funkcje były określone jednorodnie dla wszystkich tych typów.

- Typy przepełnione przypominają typy polimorficzne — zawierają zmienną określającą typ. Poza tym zawierają ograniczenie dotyczące tej zmiennej.
- Jak powiedzieliśmy wcześniej, dysponujemy różnymi typami określającymi sposoby reprezentowania liczb (Int, Float itd.).

Chcemy jednak, żeby niektóre funkcje były określone jednorodnie dla wszystkich tych typów.

- Przykładowo, wiemy, że dodawanie ma być określone jednorodnie dla wszystkich typów dotyczących liczb:

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

- Typy przepelnione przypominają typy polimorficzne — zawierają zmienną określającą typ. Poza tym zawierają ograniczenie dotyczące tej zmiennej.
- Jak powiedzieliśmy wcześniej, dysponujemy różnymi typami określającymi sposoby reprezentowania liczb (Int, Float itd.).

Chcemy jednak, żeby niektóre funkcje były określone jednorodnie dla wszystkich tych typów.

- Przykładowo, wiemy, że dodawanie ma być określone jednorodnie dla wszystkich typów dotyczących liczb:

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

- Napis `Num a` oznacza, że typ `a`, który jest zmienną w definicji dodawania, musi należeć do *klasy* `Num`. Co to jest klasa?

Klasy

- Typ jest kolekcją obiektów podobnego rodzaju. Klasa jest kolekcją typów, które wspierają takie same metody.

- Typ jest kolekcją obiektów podobnego rodzaju. Klasa jest kolekcją typów, które wspierają takie same metody.
- Zatem dwa typy a i b należą do tej samej klasy, jeśli można na obiektach typu a i b wykonywać takie same (przepełnione) operacje/funkcje.

- Rozważmy przykład klasy Eq (equality class) — więcej przykładów znajdziecie w książce *Programming in Haskell* Hutton'a.

- Rozważmy przykład klasy Eq (equality class) — więcej przykładów znajdziecie w książce *Programming in Haskell* Hutton'a.
- Klasa Eq zawiera typy, których wartości mogą być porównywane, czyli można stwierdzić, czy dwie wartości są takie same, czy różne.

- Rozważmy przykład klasy Eq (equality class) — więcej przykładów znajdziecie w książce *Programming in Haskell* Hutton'a.
- Klasa Eq zawiera typy, których wartości mogą być porównywane, czyli można stwierdzić, czy dwie wartości są takie same, czy różne.
- A zatem wszystkie typy należące do tej klasy wspierają dwie metody:

```
(==) :: a -> a -> Bool  
(/=) :: a -> a -> Bool
```


- Rozważmy przykład klasy Eq (equality class) — więcej przykładów znajdziecie w książce *Programming in Haskell* Hutton'a.
- Klasa Eq zawiera typy, których wartości mogą być porównywane, czyli można stwierdzić, czy dwie wartości są takie same, czy różne.
- A zatem wszystkie typy należące do tej klasy wspierają dwie metody:

```
(==) :: a -> a -> Bool  
(/=) :: a -> a -> Bool
```

- Podaj przykład typów w klasie Eq.

- Ord
- Show
- Read
- Num
- Integral
- Fractional
- ...

- 1 Określ typy następujących wyrażeń:
 - (a) `['a', 'b', 'c']`
 - (b) `('a', 'b', 'c')`
 - (c) `[(False, '0'), (True, '1')]`
 - (d) `([False, True], ['0', '1'])`
 - (e) `[tail, init, reverse]`
- 2 Wymyśl przykłady obiektów, które mają następujący typ:
 - (a) `bools :: [Bool]`
 - (b) `nums :: [[Int]]`
 - (c) `add :: Int -> Int -> Int -> Int`
 - (d) `copy :: a -> (a, a)`
 - (e) `apply :: (a -> b) -> a -> b`

- 3 Określ typy następujących funkcji:
 - (a) `second xs = head (tail xs)`
 - (b) `swap (x,y) = (y,x)`
 - (c) `pair x y = (x,y)`
 - (d) `double x = x*2`
 - (e) `palindrome xs = reverse xs == xs`
 - (f) `twice f x = f (f x)`
- 4 Sprawdź swoje odpowiedzi używając GHCi.
- 5 Dlaczego typ funkcji nie należy do klasy Eq?