Podstawy Programowania Funkcyjnego

Chlebowski Sz. Gajda A.

Zakład Logiki i Kognitywistyki
Instytut Psychologii
Uniwersytet im. Adama Mickiewicza

08.03.2023

Plan na dziś

1. Typy
2. Klasy

Nazwy obiektów

- 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 -}.

Źródła

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

Typy

Typy

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

Typy

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

Typy

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

Typy

- 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:

Typy

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

  `o :: T`

- Rozważmy następujące przykłady:
  - `False :: Bool`

Typy

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

```
o :: T
```

- Rozważmy następujące przykłady:
  - `False :: Bool`
  - `True :: Bool`

Typy

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

  `o :: T`

- Rozważmy następujące przykłady:
  - `False :: Bool`
  - `True :: Bool`
  - `not :: Bool -> Bool`

Typy

- 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:

$$\texttt{o :: T}$$

- 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:

$$\frac{\texttt{f :: A -> B} \qquad \texttt{e :: A}}{\texttt{f e :: B}}\ \text{app}$$

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

$$\texttt{o :: T}$$

- 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:

$$\frac{\texttt{f :: A -> B} \qquad \texttt{e :: A}}{\texttt{f e :: B}}\ \text{app}$$

- 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`

Podstawowe typy

- `Bool` zawiera dwa obiekty: `True` oraz `False`

Podstawowe typy

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

Podstawowe typy

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

```
String = [Char]
```

Podstawowe typy

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

  `String = [Char]`
- `Int` zawiera liczby całkowie, takie jak -100, czy 999. W GHC, wartości `Int` zawierają się w przedziale od -2^{29} do $2^{29} - 1$

Podstawowe typy

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

  `String = [Char]`
- `Int` zawiera liczby całkowie, takie jak -100, czy 999. W GHC, wartości `Int` zawierają się w przedziale od -2^{29} do $2^{29} - 1$
- `Integer` zawiera liczby całkowie, bez ograniczeń nakładanych na `Int`

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

  `String = [Char]`
- `Int` zawiera liczby całkowie, takie jak -100, czy 999. W GHC, wartości `Int` zawierają się w przedziale od -2^{29} do $2^{29} - 1$
- `Integer` zawiera liczby całkowie, bez ograniczeń nakładanych na `Int`
- `Float` zawiera liczby w rozwinięciu dzesiętnym, takie jak -12.34, czy 3.1415927

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

  `String = [Char]`
- `Int` zawiera liczby całkowie, takie jak -100, czy 999. W GHC, wartości `Int` zawierają się w przedziale od -2^{29} do $2^{29} - 1$
- `Integer` zawiera liczby całkowie, bez ograniczeń nakładanych na `Int`
- `Float` zawiera liczby w rozwinięciu dzesiętnym, 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`.

Listy

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

Listy

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

Listy

- 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]`

Listy

- 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]`

Listy

- 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 ciagiem elementów takich samych lub różnych typów. Piszemy `(T1,T2,...,Tn)`, żeby oznaczyć typ krotki, której elementami są obiekty typów `T1,T2,...,Tn`.

Krotki/n-tki

- Krotka jest skończonym ciagiem elementów takich samych lub różnych typów. Piszemy `(T1,T2,...,Tn)`, żeby oznaczyć typ krotki, której elementami są obiekty typów `T1,T2,...,Tn`.
- Przykłady:

Krotki/n-tki

- Krotka jest skończonym ciagiem elementów takich samych lub różnych typów. Piszemy `(T1,T2,...,Tn)`, żeby oznaczyć typ krotki, której elementami są obiekty typów `T1,T2,...,Tn`.
- Przykłady:
  - `(False, True) :: (Bool, Bool)`

Krotki/n-tki

- Krotka jest skończonym ciagiem elementów takich samych lub różnych typów. Piszemy `(T1,T2,...,Tn)`, żeby oznaczyć typ krotki, której elementami są obiekty typów `T1,T2,...,Tn`.
- Przykłady:
  - `(False, True) :: (Bool, Bool)`
  - `(False, 'a', True) :: (Bool, Char, Bool)`

Krotki/n-tki

- Krotka jest skończonym ciagiem elementów takich samych lub różnych typów. Piszemy `(T1,T2,...,Tn)`, żeby oznaczyć typ krotki, której elementami są obiekty typów `T1,T2,...,Tn`.
- Przykłady:
  - `(False, True) :: (Bool, Bool)`
  - `(False, 'a', True) :: (Bool, Char, Bool)`
  - `("Yes", True, 'a') :: (String, Bool, Char)`

- Krotka jest skończonym ciagiem elementów takich samych lub różnych typów. Piszemy `(T1,T2,...,Tn)`, żeby oznaczyć typ krotki, której elementami są obiekty typów `T1,T2,...,Tn`.
- 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 ciagiem elementów takich samych lub różnych typów. Piszemy `(T1,T2,...,Tn)`, żeby oznaczyć typ krotki, której elementami są obiekty typów `T1,T2,...,Tn`.
- 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)]`

Funkcje

- Funkcja jest przekształceniem obiektów typu `A` w obiekty typu `B`. Typ funkcji z `A` w `B` oznaczamy przez `A -> B`.

Funkcje

- Funkcja jest przekształceniem obiektów typu `A` w obiekty typu `B`. Typ funkcji z `A` w `B` oznaczamy przez `A -> B`.
- Przykłady:

Funkcje

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

Funkcje

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

Funkcje

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

Funkcje

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

Funkcje

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

Currying

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

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
```

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
```

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

Currying

- 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 zwraza `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 zwraza `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 zwraza `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 bedziemy 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 zwraza `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 bedziemy stosować dalej, typ `mult` można zapisać jeszcze prościej:

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

- Wszystkie funkcje bedziemy definiować jako funkcje jednoargumentowe.

Typy polimorficzne

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

Typy polimorficzne

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

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

Typy polimorficzne

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

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

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

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

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

- Jak należy się spodziewać, funkcja ta może być jednorodnie zdefiniowana dla list obiektów dowolnego typu (oznaczanego 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 przepełnione

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

Typy przepełnione

- 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

- 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 wsyzstkich typów dotyczących liczb:

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

Typy przepełnione

- 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 wsyzstkich typów dotyczących liczb:

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

- Napis `Num a` oznacza, że typ `a`, który jes 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`.

Klasy

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

Zadania (1)

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`

Zadania (2)

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`?