Języki i Paradygmaty Programowania

Haskell i programowanie funkcyjne

Marcin Benke

MIM UW

6 marca 2017

Typy algebraiczne

Możemy definiować własne typy, np:

```
data TakNie = Tak | Nie
```

…i używać ich we wzorcach

```
nie :: TakNie -> TakNie
nie Tak = Nie
nie Nie = Tak
```

Tak i **Nie** nazywamy *konstruktorami* (wartości). Zachowują się one jak funkcje (w tym wypadku zeroargumentowe).

Dla wbudowanego typu listowego **[]** jest konstruktorem 0-argumentowym, a **(:)** — 2-argumentowym.

Typy algebraiczne

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)

mapTree :: (a->b) -> Tree a -> Tree b
mapTree f (Leaf a) = Leaf (f a)
mapTree f (Branch l r) = Branch (m l) (m r) where
  m = mapTree f
```

Leaf jest 1-argumentowym konstruktorem,
Branch — 2-argumentowym.

Per analogiam mówimy, że **Tree** jest jednoargumentowym *konstruktorem typu*:

- jeśli **x** jest wartością, to **Leaf x** jest wartością;
- jesli **a** jest typem, to **Tree a** jest typem.

Typy Maybe i Either

Dwa przydatne typy (predefiniowane w Prelude):

```
data Maybe a = Nothing | Just a
data Either a b = Left a | Right b
-- Prelude> head []
-- *** Exception: Prelude.head: empty list

safeHead :: [a] -> Maybe a
safeHead [] = Nothing
safeHead (x:_) = Just x
-- *Main> safeHead []
-- Nothing

safeHead2 :: [a] -> Either String a
safeHead2 [] = Left "Empty list"
safeHead2 (x:xs) = Right x
```

Synonimy

Czasem przydatne jest wprowadzenie własnej nazwy (synonimu) dla jakiegoś typu.

```
type Name = String
type Possibly = Either Name

safeHead3 :: [a] -> Possibly a
safeHead3 [] = Left "Empty list"
safeHead3 (x:xs) = Right x
```

Synonim **nie jest** konstruktorem typu; jest identyczny z nazywanym typem.

Etykiety pól

Spójrzmy na definicje

```
data Point = Pt Float Float
pointx                  :: Point -> Float
pointx (Pt x _)         =  x
pointy ...
```

Definicja **pointx** jest "oczywista"; możemy krócej:

```
data Point = Pt {pointx, pointy :: Float}
```

W jednej linii definiujemy typ **Point**, konstruktor **Pt** oraz funkcje **pointx** i **pointy**.

Opakowywanie typów: **newtype**

Jeśli chcemy opakować istniejacy typ w nowy konstruktor typu, mozemy uzyć konstrukcji **newtype**:

```
newtype Identity a = Identity { runIdentity :: a }
  deriving (Eq, Show)

*Newtype> Identity "Ala"
Identity {runIdentity = "Ala"}
*Newtype> runIdentity it
"Ala"
```

newtype działa niemal identycznie jak **data** z jednym konstruktorem (ale efektywniej; pakowanie/odpakowywanie odbywa się w czasie kompilacji a nie wykonania).

Klasy — motywacja

Rozważmy funkcję sprawdzającą, czy element należy do listy:

```
el x [] = False
el x (y:ys) = (x==y) || el x ys
```

Będzie ona miała typ $\mathbf{a} \to [\mathbf{a}] \to \mathbf{Bool}$ dla każdego typu **a**, dla którego jest zdefiniowana równość.

W ML jest specjalna notacja dla takich typów:
`''a -> ''a list -> bool` ale dotyczy tylko równości.

W Haskellu **el** ma typ $(\mathbf{Eq\ a}) \Rightarrow \mathbf{a} \to [\mathbf{a}] \to \mathbf{Bool}$, przy czym fragment $(\mathbf{Eq\ a}) \Rightarrow$ nazywamy ograniczeniem (kontekstem) typu; jest on częścią szerszego mechanizmu: klas

Klasy

- Pojęcie klasy nie jest tym samym co w językach obiektowych.
- Określa raczej protokół, który musi wypełnić typ, by traktować go jako instancję danej klasy.
- Mechanizm klas można kojarzyć z polimorfizmem w Smalltalku, czy *duck typing* w językach skryptowych (ale to odległe skojarzenie).
- Bliższym skojarzeniem jest mechanizm przeciążania operatorów, ale klasy pozwalają na o wiele więcej…
- Funkcja typu $(\mathbf{C}\ \mathbf{a}) \Rightarrow \mathbf{t}$ jest przeciążona zwn **a**:
  - działa dla typów **a**, które są instancjami **C**;
  - Może działać w różny sposób dla różnych typów **a** (w odróżnieniu od funkcji polimorficznych);
  - sposób działania wynika z definicji instancji **C**

Definicja klasy

Zdefiniujmy własną klasę typów Porównywalnych:

```
class Por a where
  (===) :: a -> a -> TakNie
  (=/=) :: a -> a -> TakNie
```

teraz możemy powiedzieć, że elementy typu **TakNie** są Porównywalne (pamiętając o wcięciach!):

```
instance Por TakNie where
  Tak === Tak = Tak
  Nie === Nie = Tak
  _   ===  _  = Nie

  a =/= b = nie $ a === b
```

Zauważmy, że to ostatnie równanie będzie się pojawiało w każdej prawie definicji instancji, możemy więc je przenieść do definicji klasy jako domyślną definicję **(=/=)**

Klasy a synonimy

Instancje klas zasadniczo nie powinny sie odnosić do synonimów:

```
type Baby = Maybe

instance (Por a) => Por (Baby a) where
  Nothing === Nothing = Tak

Baby.hs:7:0:
Illegal instance declaration for `Por (Baby a)'
(All instance types must be of the form
  (T t1 ... tn) where T is not a synonym.
  Use -XTypeSynonymInstances if you want to
  disable this.)
    In the instance declaration for `Por (Baby a)'
```

Oczywiście nie przeszkadza to w zdefiniowaniu

```
instance (Por a) => Por (Maybe a) where...
```

Klasy Eq i Ord

```
class  Eq a  where
 (==), (/=)  ::  a -> a -> Bool
 x /= y = not (x == y)
 x == y = not (x /= y)
```

Ord: tylko sygnatury, bez domyślnych definicji

```
class  (Eq a) => Ord a  where
 compare              :: a -> a -> Ordering
 (<), (<=), (>=), (>) :: a -> a -> Bool
 max, min             :: a -> a -> a
```

Ta druga notacja oznacza “typ jest instancją Ord jeśli jest instancją Eq, a ponadto są zdefiniowane funkcje…”

Klasa Enum

Typy wyliczeniowe (niekoniecznie skończone)

```
class  Enum a  where
 succ, pred     :: a -> a
 toEnum         :: Int -> a
 fromEnum       :: a -> Int
 enumFrom       :: a -> [a]             -- [n..]
 enumFromThen   :: a -> a -> [a]        -- [n,n'..]
 enumFromTo     :: a -> a -> [a]        -- [n..m]
 enumFromThenTo :: a -> a -> a -> [a] -- [n,n'..m]
```

Klasa Show

Klasa **Show** zawiera funkcje dające tekstową reprezentację wartości. W skrócie

```
class Show a where
  show :: a -> String
  ...
```

ghci potrafi wydrukować tylko elementy typów, które są instancjami klasy **Show**:

```
> Tak === Nie
No instance for (Show TakNie)
 arising from a use of ‘print’ at ...
Possible fix: add an instance declaration for
  (Show TakNie)
```

powinniśmy więc dodać definicję, która powie w jaki sposób typ **TakNie** może spełnić protokół **Show**, ale jest prostszy sposób…

Automatyczne instancje

Instancje dla klas typu **Eq** czy **Show** są przeważnie nudne i przewidywalne, kompilator mógłby je sam wygenerować...

Automatyczne instancje

Instancje dla klas typu **Eq** czy **Show** są przeważnie nudne i przewidywalne, kompilator mógłby je sam wygenerować...

...i potrafi, możemy go poinstruować w tym kierunku przy pomocy dyrektywy **deriving**, np.

```
data TakNie = Tak | Nie
  deriving (Eq, Show)
```

Ten mechanizm ma zastosowanie tylko do klas standardowych (istnieją bogatsze mechanizmy, ale nie będziemy się tu nimi zajmować).

Klasa Num

```
class (Eq a, Show a) => Num a where
  (+) :: a -> a -> a
  (*) :: a -> a -> a
  (-) :: a -> a -> a
  negate :: a -> a
  abs :: a -> a
  signum :: a -> a
  fromInteger :: Integer -> a

instance Num Int
instance Num Integer
instance Num Double
instance Num Float

Prelude> :t 0
0 :: (Num t) => t
```

Klasa Read

Klasa **Read** jest poniekąd dualna do klasy **Show**: pozwala na odczytanie wartości z jej reprezentacji napisowej. Jest to na ogół trudniejsze niż **show**, na razie wystarczy nam znać funkcję

```
read :: (Read a) => String -> a
```

```
Prelude> (+) (read "2")  (read "3")
5
```

Wskazywanie typu

W przypadku funkcji przeciążonych nie zawsze można rozstrzygnąć, o jaki typ nam chodzi, np

```
Prelude> :t read
read :: (Read a) => String -> a
Prelude> read "1"

<interactive>:1:0:
Ambiguous type variable ‘a’ in the constraint:
  ‘Read a’ arising from a use of ‘read’ at <interactive>:1
    Probable fix: add a type signature that fixes these
     type variable(s)
Prelude> (read "1")::Int
1
```

Moduły

Program w Haskellu jest kolekcją modułów

```
module Lub where
import TakNie

lub :: TakNie -> TakNie -> TakNie
Tak `lub` coś = coś
Nie `lub` coś = Nie
```

w tym przykładzie widzimy moduł **Lub** importujący moduł **TakNie**.

Moduł powinien być umieszczony w pliku o odpowiedniej nazwie (np. moduł **Lub** w pliku **Lub.hs**)

Moduły hierarchiczne

Moduły mogą być (i często są) organizowane hierarchicznie

```
module Utils.Char where
import Data.Char

decDigit :: Char -> Int
...
```

Moduł **Utils.Char** powinien być w pliku **Utils/Char.hs**
(względem korzenia drzewa naszego programu).

Selektywny eksport i import

Domyślnie moduł eksportuje (a import importuje) wszystkie definicje modułu. Możemy to oczywiście zmienić:

```
module Utils.Char2(decDigit,hexDigit,isDigit) where
import Data.Char(isDigit,ord,toLower)

decDigit :: Char -> Int
...
```

W tym przykładzie z **Data.Char** importujemy definicje **isDigit,ord,toLower**,
eksportujemy własne funkcje **decDigit, hexDigit**.
Reeksportujemy zaimportowaną definicję **isDigit**

Import kwalifikowany i ukrywanie

Do nazw możemy się też odwoływać, kwalifikując je nazwą modułu

```
Prelude.map (Prelude.+1) [1..9]
```

Jeśli chcemy, żeby zaimportowane nazwy nie mieszały się z lokalnymi, możemy użyć **import qualified**

```
import qualified Data.Vector as V
import qualified Data.List as L
L.map $ toList $ V.map (+1) (V.enumFromTo 1 10)
```

Z kolei jeśli chcemy zaimportować wszystkie nazwy oprócz kilku, możemy użyć **hiding**:

```
import Prelude hiding(map,filter)
map = ...
filter = ...
```

Kompilacja programu wielomodułowego

Wśród modułów programu jeden musi być główny:

```
module Main(main) where
...
main :: IO ()
```

Najprościej zbudować program przez

```
ghc -o nazwa --make main.hs
```

Moduł **Main** stanowi wyjątek od reguły nazywania plików, może się nazywać jakkolwiek (zwykle tak jak cały program)

```
ghc --make nazwa.hs
```

Wejście-wyjście

- Funkcje w Haskellu co do zasady nie mają efektów ubocznych
- Przeważnie jednak chcemy, aby nasz program miał jakiś efekt
- Dlatego funkcja **main** jest (poniekąd) wyjątkiem: może mieć efekty, co jest zaznaczone w jej typie: **IO ()** oznacza, że funkcja nie daje interesującego wyniku, za to daje efekt wejścia-wyjścia.
- Więcej na kolejnych wykładach, na razie dwie funkcje IO

Skad się bierze IO

- Poznaliśmy juz funkcję

  ```
  interact :: (String -> String) -> IO ()
  ```

  bierze ona jako argument funkcję operującą na strumieniach znaków i zamienia ją w *interakcję*. Zauważmy, że dzięki leniwości możemy zacząć produkować wyjście zanim wczytamy całe wejście.
- Inną przydatną funkcją jest **print**

  ```
  print :: (Show a) => a -> IO ()
  ```

  potrafi ona wydrukować wszystkie instancje klasy **Show** (wszystko co da się zamienić na **String**).

Klasy konstruktorowe

Typy polimorficzne jak **[a]** czy **Tree a** mogą być instancjami klas (przeważnie pod warunkiem, ze **a** jest też instancją odpowiedniej klasy)...

```
data Tree a = Leaf a | Branch (Tree a) (Tree a)
  deriving Show

instance Eq a => Eq (Tree a) where
  Leaf x == Leaf y = x == y
  Branch l r == Branch l' r' = (l==l')&&(r==r')
```

Klasy konstruktorowe

. . . ale są też klasy, których instancjami są nie typy, a *konstruktory typów*. Na przykład funkcję **map** możemy uogólnić na inne pojemniki:

```
-- Klasa Functor jest zdefiniowana w Prelude
-- class  Functor t  where
--     fmap :: (a -> b) -> t a -> t b

(<$>) :: Functor f => (a -> b) -> f a -> f b
(<$>) = fmap

instance Functor Tree where
 fmap f (Leaf a) = Leaf $ f a
 fmap f (Branch l r) = Branch (fmap f l)(fmap f r)

*Tree> negate <$> Leaf 6
Leaf (-6)
```

Klasy konstruktorowe

- Typy takie jak **Tree** czy listy są pojemnikami przechowującymi obiekty
- Instancja **Eq(Tree a)** mówi o własnościach pojemnika z zawartością
- Instancja **Functor Tree** mówi o własnościach samego pojemnika, *niezależnie od zawartości*

Klasy konstruktorowe

```
import Prelude hiding(Functor(..))

class Functor f where
  fmap :: (a -> b) -> (f a -> f b)

-- [] poniżej oznacza konstruktor *typu* dla list
instance Functor [] where
  fmap = map

instance Functor Maybe where
  fmap f (Just x) = Just (f x)
  fmap f Nothing = Nothing
```

Applicative

Chcemy dodać dwie liczby opakowane w **Maybe**

```
*Applicative> :t (+1) <$> Just 5
(+1) <$> Just 5 :: Num b => Maybe b
*Applicative> :t (+) <$> Just 5
(+) <$> Just 5 :: Num a => Maybe (a -> a)
```

czyli nie możemy napisać `(+) <$> Just 2 <$> Just 3`
W ogólności Functor nie wystarczy, potrzeba

```
class (Functor f) => Applicative f where
    pure  :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

teraz:

```
*Applicative> (+) <$> Just 2 <*> Just 3
Just 5
```

Klasy wieloparametrowe

Powiedzmy, ze chcemy zdefiniować klasę kolekcji

```
class Collection c where
  insert :: e -> c -> c
  member :: e -> c -> Bool

instance Eq a => Collection [a] where
     insert x xs = x:xs
     member = elem
```

to się niestety nie skompiluje (co to jest "`e`" w Collection?)

Klasy wieloparametrowe

```
{-# LANGUAGE MultiParamTypeClasses #-}
{-# LANGUAGE FlexibleInstances #-}
class Collection c e where
  insert :: e -> c -> c
  member :: e -> c -> Bool

instance Eq a => Collection [a] a where
     insert = (:)
     member = elem
```

NB musimy użyć rozszerzeń wykraczających poza standard Haskell 2010, stąd pragmy.

Zależności funkcyjne (functional dependencies)

Typ kolekcji determinuje typ elementu
Można to wyrazić przez *zależności funkcyjne*:

```
{-# LANGUAGE FunctionalDependencies #-}
class Collection c e | c -> e where
```

Skojarzenie z bazami danych jest słuszne.
Inny przykład

```
class Mult a b c | a b -> c where
  (|*) :: a -> b -> c
```

Type Families

Innym rozwiązaniem są rodziny typów

```
{-# LANGUAGE TypeFamilies, MultiParamTypeClasses #-}

class Collection c where
    type Elem c
    insert :: Elem c -> c ->c

class Mul a b where
    type MulResult a b
    mul :: a -> b -> MulResult a b
```