Programowanie funkcyjne
(wykład 4.)

Roman Dębski

Instytut Informatyki, AGH

14 listopada 2024

Plan wykładu

Plan wykładu

Typy w Haskellu, *type* vs. *newtype* vs. *data*

Typy w Haskellu
- typy proste: ***Int***, ***Double***, ...
- typy złożone: *krotki*, *listy*, *funkcje*, ...
- typy algebraiczne (created by 'algebraic' operations): *tworzone konstrukcją* ***data ...***
- a także:

type [synonim, alias]

```
type Name = String
capitalizeName :: Name -> Name
capitalizeName = map toUpper
```

newtype [constr. of a newtype must have exactly one field!]

```
newtype FirstName = FirstName String
formatFstName :: FirstName -> String
formatFstName (FirstName s) = case s of
  (x:xs) -> toUpper x : map toLower xs
  []     -> []
```

```
ghci> :t capitalizeName
capitalizeName :: Name -> Name

ghci> capitalizeName "t-1000"
"T-1000"
```

```
ghci> formatFstName(FirstName "apolinary")
"Apolinary"
ghci> formatFstName("apolinary")-- error*
```

* *expected type 'FirstName', actual type '[Char]'*

Algebraiczne typy danych: *algebra typów*

Wyrażenie algebraiczne	Odpowiednik w *algebrze typów*
0	*Void*
1	*()* - *'Unit'*
$a+b$	*data Add a b = AddL a | AddR b* (*data Either a b*)
$a*b$	*(a,b)* lub *data Mul a b = Mul a b*
$2=1+1$	*data Bool = True | False*
$1+a$	*data Maybe a = Nothing | Just a*
b^a	*a → b*
$0+a=a$	*Add Void a ∼ a*
$0*x=0$	*Mul Void a ∼ Void*
$1*x=x$	*Mul () a ∼ a*
$a*(b+c)=a*b+a*c$	*Mul a (Add b c) ∼ Add (Mul a b) (Mul a c)*
$L(a)=1+aL(a)=1+a+a^2+\ldots$	*data List a = EmptyL | Cons a (List a)*
$T(a)=1+aT(a)^2=1+a+2a^2+\ldots$	*data Tree a = EmptyT | Node a (Tree a) (Tree a)*

Algebraiczne typy danych: *product and sum types*

Product type [single constructor, ∼record]

```
data AandB a b = AandB_Con a b
```

```
data Person n s = Person n s -- *

ghci> :i Person
data Person n s = Person n s

ghci> :t Person
Person :: n -> s -> Person n s

ghci> p1 = Person "Inigo" "Montoya"
ghci> :t p1
p1 :: Person [Char] [Char]
```

*the same names for the type constructor and a data/value constructor

Sum type [co-product, disjoint union]

```
data ABorB a b=AB_Con a b | B_Con b
```

```
data Either a b = Left a | Right b

ghci> :t Left
Left :: a -> Either a b
ghci> :t Right
Right :: b -> Either a b
ghci> let r = Right 3
ghci> let l = Left 'a'

-- enum (special case of sum type)
data ThreeColors = Blue |
                   White |
                   Red
ghci> :t Blue
Blue :: ThreeColors
```


Algebraiczne typy danych: *record syntax*

```
data Person = Person String String
              deriving (Show)

name :: Person -> String
name (Person n _) = n

surname :: Person -> String
surname (Person _ sn) = sn
```

Record syntax

```
data Person = Person
              { name :: String
              , surname :: String
              } deriving (Show)
```

```
ghci> let swordMaster =
  Person "Inigo" "Montoya"

ghci> swordMaster
Person "Inigo" "Montoya"

ghci> name swordMaster
"Inigo"
ghci> surname swordMaster
"Montoya"
```

```
ghci> let swordMaster =
  Person { surname = "Montoya" ,
           name = "Inigo" }
ghci> swordMaster
Person { name = "Inigo",
         surname = "Montoya" }
ghci> name swordMaster
"Inigo"
ghci> surname swordMaster
"Montoya"
```


Algebraiczne typy danych: *parametric types, patterns in record syntax*

```
data Car = Car
           { company :: String
           , model :: String
           , year :: Int
           } deriving (Show)
```

```
data Car a b c = Car
                 { company :: a
                 , model :: b
                 , year :: c
                 } deriving (Show)
```

Przykł.: dopasowanie wzorca (record syntax)

```
carInfo :: (Show a) => Car String String a -> String
carInfo (Car {company = c, model = m, year = y}) =
    "This " ++ c ++ " " ++ m ++ " was made in " ++ show y
```

```
ghci> let focus = Car {company="Ford", model="Focus I", year=2004}

ghci> carInfo focus
"This Ford Focus I was made in 2004"
```

Algebraiczne typy danych: przykład *(structural recursion)*

```
data Tree a = Nil |
              Node a (Tree a) (Tree a)
              deriving (Eq, Ord, Show, Read) -- Eq, Ord ???

depth :: Tree a -> Int
depth Nil = 0
depth (Node n lt rt) = 1 + max (depth lt) (depth rt)

collapse :: Tree a -> [a]
collapse Nil = []
collapse (Node n lt rt) = collapse lt ++ [n] ++ collapse rt

mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f Nil = Nil
mapTree f (Node n lt rt) = Node (f n) (mapTree f lt) (mapTree f rt)
```


Typy wyższego rzędu (*kinds*) [typy vs. konstruktory typów vs. konstruktory danych]

```
ghci> :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
instance Bounded Int
...
```

```
ghci> :k Int
Int :: *
ghci> :k Int -> Int
Int -> Int :: *
```

```
ghci> :i Maybe
data Maybe a = Nothing | Just a

ghci> :k Maybe
Maybe :: * -> *
ghci> :k Maybe Int
Maybe Int :: *

ghci> :i Either
data Either a b = Left a | Right b
ghci> :k Either
Either :: * -> * -> *
ghci> :k Either Int
Either Int :: * -> *
```

```
data CDT a b =
  CDT { e :: Either a b
      , f :: Either a b -> Maybe a }
ghci> :k CDT
CDT :: * -> * -> *

ghci> :t CDT
CDT :: Either a b ->
  (Either a b -> Maybe a) -> CDT a b

ghci> :t f
f :: CDT a b -> Either a b -> Maybe a
```

```
{-#LANGUAGE KindSignatures #-}
```


Plan wykładu

Klasy typów w Haskellu vs. klasy w językach obiektowych

Klasy typów w Haskellu to mechanizm realizacji **polimorfizmu *ad-hoc*** ('przeciążanie' nazw).

A class – collection of types (its instances) over which function(s) is (are defined). One way we can think of a class is as an adjective: any particular type IS or IS NOT in the class.

Other languages such as C++ or Java make a type and a class the same thing.

A type is made a member or instance of a class by defining the interface functions for the type.

Instances in Haskell are global; it is not possible to make instances local to a module or a set of modules (note: 'wrapped' type is often a solution to this problem).

```
class [cx =>] <ClassName> tv where
  -- signature (interface, contract) involving the type variable tv

instance [cx =>] <ClassName> <InsData> where -- e.g. InsData = Int
  -- implementation of the methods declared in ClassName
```


Klasy typów i ich instancje: przykład

```
newtype Vec2Dnt a = Vec2Dnt (a, a)
instance Eq a => Eq (Vec2Dnt a) where
  (==) (Vec2Dnt (x1,y1)) (Vec2Dnt (x2,y2)) = x1 == x2 && y1 == y2
-- ghci> Vec2Dnt (2,2) == Vec2Dnt (2,2) -- True
```

```
class VectorLike (t :: * -> *) where
  (|==|) :: Eq a => t a -> t a -> Bool
  (|+|), (|-|) :: (Num a) => t a -> t a -> t a
  (|*|) :: (Num a) => t a -> t a -> a
  ... -- {-#LANGUAGE KindSignatures #-} needed to use (t :: * -> *)

instance VectorLike Vec2Dnt where
    (|==|) (Vec2Dnt (x1,y1)) (Vec2Dnt (x2,y2)) = ...
    (|+|) (Vec2Dnt (x1,y1)) (Vec2Dnt (x2,y2))  = ...
    ... -- ghci> Vec2Dnt (1,-1) |+| Vec2Dnt (2,2) -- Vec2Dnt (3,1)

(|-?) :: (VectorLike t, Num a, Eq a) => t a -> t a -> Bool
(|-?) v1 v2 = v1 |*| v2 == 0 -- is this 'safe' for all Num types?
```


Plan wykładu

Moduły i importy

Program w Haskellu = **kolekcja modułów** (w tym *Main* zawierający funkcję *main*)

Rola/zastosowanie modułów

tworzenie przestrzeni nazw (i zarządzanie nimi), mechanizm tworzenia abstrakcyjnych typów danych (*ADT*), jednostka struktury kodu (jednostka kompilacji)

```
module MName (<exportNames>) where
...

import MName (<importNames>)
-- vs.
import MName hiding (<lst>)
-- vs.
import qualified MName as MyN
-- vs.
import qualified MName as NyN(x,y)
```

```
module Stack (Stack, push, pop) where

import Data.List
import Text.Printf(printf)
import qualified Data.Map as Map
import Prelude hiding (zip)

push :: a -> Stack a -> Stack a
...

module Stack (Stack(..),push,pop) --!
```


Moduły: mechanizm definicji abstrakcyjnych typów danych (*ADT*)

Stos jako *ADT* [przykład]

```
module Stack (Stack, empty, isEmpty, push, top, pop) where

-- interface (signature, contract)
empty :: Stack a
isEmpty :: Stack a -> Bool
push :: a -> Stack a -> Stack a
top :: Stack a -> a
pop :: Stack a -> (a,Stack a)

-- implementation
newtype Stack a = StackImpl [a] -- hidden constructor

empty = StackImpl []
isEmpty (StackImpl s) = null s
push x (StackImpl s) = StackImpl (x:s)
top (StackImpl s) = head s
pop (StackImpl (s:ss)) = (s,StackImpl ss)
```

Organizacja kodu źródłowego: narzędzie *Stack*

"**Stack*** is a cross-platform program for developing Haskell projects"

— https://docs.haskellstack.org/en/stable/README/

```
$ stack new my-project
$ cd my-project
$ stack setup
$ stack build
$ stack exec my-project-exe

$ stack test
$ stack ghci
```

```
$  my-project tree .
.
|-- LICENSE
|-- Setup.hs
|-- app
|   |-- Main.hs
|-- my-project.cabal
|-- src
|   |-- Lib.hs
|-- stack.yaml
|-- test
    |--- Spec.hs

3 directories, 7 files
```

* "*Stack is a project of the Commercial Haskell group, spearheaded by FP Complete*"

Bibliografia

- Simon Thompson, Haskell: The Craft of Functional Programming, Addison-Wesley Professional, 2011
- Graham Hutton, Programming in Haskell, Cambridge University Press, 2007
- https://wiki.haskell.org
- http://learnyouahaskell.com
- http://book.realworldhaskell.org/read
- http://chris-taylor.github.io/blog/2013/02/10/the-algebra-of-algebraic-data-types
- https://docs.haskellstack.org