Wykład 12

Klasy typów jako mechanizm abstrakcji

Klasy typów jako mechanizm przeciążania

Klasy typów jako mechanizm abstrakcji

Klasy Semigroup

Klasa Monoid i operacja „zwijania”

Klasy Functor, Applicative i Monad

Motto

Matematykiem jest, kto umie znajdować analogie między twierdzeniami; lepszym, kto widzi analogie dowodów, i jeszcze lepszym, kto dostrzega analogie teorii, a można wyobrazić sobie i takiego, co między analogiami widzi analogie.

Stefan Banach

Jak zobaczymy, informatycy również doceniają umiejętność zauważania analogii i tworzenia abstrakcji – nie tylko w teorii, lecz również w praktyce.

Klasy typów jako mechanizm przeciążania

Na poprzednim wykładzie były przedstawione klasy typów, stanowiące pewne uogólnienie typów konkretnych. Na przykład dzięki klasie `Num` można używać takich samych operacji dla różnych konkretnych typów numerycznych. Ogólniej, te same metody klasy mogły być używane dla wartości różnych typów, czyli Haskell w sposób bezpieczny implementuje mechanizm przeciążania (polimorfizm ad-hoc).

- Polimorfizm „ad hoc” umożliwia użycie tej samej nazwy dla różnych fragmentów kodu, których zachowanie może być całkowicie różne dla różnych typów.

  Do najważniejszych odmian polimorfizmu ad hoc należy przeciążenie oraz koercja (ang. coercion). Granica między przeciążeniem a koercją w wielu sytuacjach jest nieostra i zależy od implementacji.

Klasy typów były definiowana względem pewnej zmiennej, przebiegającej typy, np.

```
class Eq a  where  ...
```

gdzie `a :: *`, czyli zmienna `a` przebiega typy (jej konkretyzacją musi być typ).

https://wiki.haskell.org/Typeclassopedia

Klasy typów jako mechanizm abstrakcji

Na dzisiejszym wykładzie będzie mowa o klasach typów, których korzenie tkwią w matematyce (algebry abstrakcyjne i teoria kategorii). Informacje o takich klasach typów zawiera Typeclassopedia: https://wiki.haskell.org/Typeclassopedia

Dzisiaj zapoznamy się z najważniejszymi z nich.

Na wykładzie 7 zauważyliśmy związek między abstrakcyjnymi typami danych a algebrami abstrakcyjnymi.

Algebrą abstrakcyjną nazywamy *zbiór elementów S* (nośnik, dziedzina lub uniwersum algebry), na którym są zdefiniowane pewne *operacje*, posiadające własności zadane przez *aksjomaty* równościowe.

- *Półgrupa* $\langle S, \bullet \rangle$ $\bullet : S \times S \to S$

  $a \bullet (b \bullet c) = (a \bullet b) \bullet c$ (łączność)
- *Monoid* $\langle S, \bullet, 1 \rangle$ jest półgrupą z obustronną jednością (elementem neutralnym) $1 : S$

  $a \bullet 1 = a$ $1 \bullet a = a$

Biblioteka Haskella zawiera klasy typów dla półgrupy (Semigroup) i monoidu (Monoid).

Klasa Semigroup

• *Półgrupa* $\langle S, \bullet \rangle$ $\bullet : S \times S \to S$
$a \bullet (b \bullet c) = (a \bullet b) \bullet c$ (łączność)

```
class Semigroup a where
  (<>) :: a -> a -> a

Prelude> [1,2] <> [4,5]
[1,2,4,5]
Prelude> "Hello " <> "world"
"Hello world"
```

Więcej definicji dla klasy można znaleźć w pakiecie Data.Semigroup, np.:

```
newtype Sum a  =  Sum {getSum :: a}
```

```
newtype Product a  =  Product {getProduct :: a}
```

```
Instance Num a => Semigroup (Sum a)
instance Num a => Semigroup (Product a)
```

```
Prelude> import Data.Semigroup
Prelude Data.Semigroup> Sum 2 <> Sum 3 <> Sum 10
Sum {getSum = 15}
Prelude Data.Semigroup> getProduct (Product 2 <> Product 3 <> Product 10)
60
```

Klasa Monoid

- *Monoid* ⟨S, •, 1⟩ jest półgrupą z obustronną jednością (elementem neutralnym) 1:S
  a•1=a 1•a=a

```
class Semigroup a => Monoid a where
  mempty :: a                  -- element neutralny monoidu
  mappend :: a -> a -> a       -- binarna operacja monoidu


Prelude> mempty :: [Int]
[]
Prelude> mempty :: String
""
Prelude> import Data.Monoid
Prelude Data.Monoid> mempty :: Sum Int
Sum {getSum = 0}
Prelude Data.Monoid> mempty :: Product Int
Product {getProduct = 1}
Prelude Data.Monoid> Sum 2 `mappend` Sum 3 `mappend` Sum 10
Sum {getSum = 15}
```

Operacja „zwijania” w klasie Monoid

Klasa Monoid zawiera też metodę mconcat z domyślną implementacją:

```
mconcat :: [a] -> a
mconcat = foldr mappend mempty
```

Jak widać z definicji, „zwija” ona monoid za pomocą funkcji foldr.

Taka funkcja jest też dostępna ze standardowego preludium:
mconcat :: Monoid a => [a] -> a

Porównaj poniższy kod z przykładami z wykładu 3, str. 16-17.

```
Prelude Data.Monoid> mconcat [Sum 4, Sum 3, Sum 2, Sum 1]
Sum {getSum = 10}
Prelude Data.Monoid> mconcat [Product 4, Product 3, Product 2, Product 1]
Product {getProduct = 24}
Prelude Data.Monoid> mconcat [[5, 6], [1, 2, 3]]
[5,6,1,2,3]
Prelude Data.Monoid> mconcat ["Ala ", "ma ", "kota"]
"Ala ma kota"
```

Wykorzystanie polecenia :info

Zapoznając się z nowym typem danych warto sprawdzić, do jakich klas typów należy. Warto to również zrobić teraz retrospektywnie dla znanych typów. Poniżej mamy odpowiedź dla list.

```
Prelude> :i []
data [] a = [] | a : [a]        -- Defined in ‘GHC.Types’
instance Applicative [] -- Defined in ‘GHC.Base’
instance Eq a => Eq [a] -- Defined in ‘GHC.Classes’
instance Functor [] -- Defined in ‘GHC.Base’
instance Monad [] -- Defined in ‘GHC.Base’
instance Monoid [a] -- Defined in ‘GHC.Base’
instance Ord a => Ord [a] -- Defined in ‘GHC.Classes’
instance Semigroup [a] -- Defined in ‘GHC.Base’
instance Show a => Show [a] -- Defined in ‘GHC.Show’
instance Read a => Read [a] -- Defined in ‘GHC.Read’
instance Foldable [] -- Defined in ‘Data.Foldable’
instance Traversable [] -- Defined in ‘Data.Traversable’
```

Uogólnienie funkcji

Spróbujmy teraz przeprowadzić uogólnienie dla pojęcia fundamentalnego w programowaniu funkcyjnym, mianowicie funkcji.

W rzeczywistości od pierwszego wykładu używaliśmy takiego uogólnienia dla list w postaci funkcjonału map. Porównajmy typy operacji aplikacji ($) i funkcjonału map.

```
($)   :: (a -> b) -> a -> b
map :: (a -> b) -> [a] -> [b]
```

Np. map (+ 1) [1, 2] = [2, 3].

Można łatwo zdefiniować funkcjonał fmap :: (a -> b) -> Maybe a -> Maybe b tak, żeby fmap (+ 1) (Just 5) = Just 6.

A gdyby uogólnić ten pomysł dla innych konstruktorów typów f :: * -> * i zdefiniować

fmap :: (a -> b) -> f a -> f b ?

Na poprzednim wykładzie była mowa o tym, że konstruktor typu dla list może być używany prefiksowo, czyli:

map :: (a -> b) -> [] a -> [] b

Klasa Functor

A gdyby uogólnić ten pomysł dla innych konstruktorów typów f :: * -> * i zdefiniować

fmap :: (a -> b) -> f a -> f b ?

Właśnie to robi klasa Functor.

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

Można sobie wyobrazić, że konstruktor typu f reprezentuje pewien *kontener*, a fmap aplikuje zadaną funkcję do każdego elementu tego kontenera. Taka intuicja dobrze pasuje do list czy drzew.

Ogólniej można spojrzeć na f jako na pewien *kontekst obliczeniowy*, który może umożliwiać powodowanie efektów ubocznych, związanych np. z wejściem/wyjściem (IO).

Operator infiksowy (<$>) :: Functor f => (a -> b) -> f a -> f b jest synonimem dla fmap. Można go używać analogicznie do operatora aplikacji ($),

```
(+ 1) $ 3 = 4
(+ 1) <$> [1, 2, 3] = [2, 3, 4]
```

Klasa Functor

Klasa Functor jest zdefiniowana w standardowym preludium Haskella.

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

Instancjami klasy Functor mogą być typy, na wartościach których można wykonać mapowanie. Listy, IO i Maybe są instancjami tej klasy.
Instancje klasy Functor powinny spełniać następujące aksjomaty:

fmap id = id

fmap (f . g) = fmap f . fmap g

Wszystkie instancje klasy Functor zdefiniowane w preludium spełniają te aksjomaty.

Zauważ, że typ f w definicji klasy Functor jest gatunku * → * (jednoargumentowy konstruktor typu).

```
*Main> fmap (+1) [1,2]          -- [] jest instancją klasy Functor
[2,3]

*Main> fmap (+1)(Just 5)        -- Maybe jest instancją klasy Functor
Just 6
```

Klasa Functor cd.

Zdefiniujmy własny typ Maybe' jako instancję klasy Functor.

```
data Maybe' a = Just' a | Nothing'
                deriving (Show)

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

fmap (^2) (Just' 5)      -- Just' 25

fmap show (Just' 5)    -- Just' "5"
```

Klasa Functor pozwala na uogólnienie aplikacji funkcji jednoargumentowych na dowolne typy (instancje klasy Functor).

```
Prelude> length <$> getLine
abcd
4
```

IO też jest instancją klasy Functor.

Klasa Functor cd.

Tree też może być instancją klasy Functor.

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

instance Functor Tree  where
    fmap f (Leaf x)          = Leaf (f x)
    fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)

*Main> fmap (+1) (Branch (Leaf 1) (Leaf 2))
Branch (Leaf 2) (Leaf 3)
```

Zwróć uwagę na informacje o błędach gatunku.

```
instance Functor (Tree Int) where
   fmap f (Leaf x)       = Leaf (f x)
   fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)


    * Expected kind '* -> *', but 'Tree Int' has kind '*'
    * In the first argument of 'Functor', namely '(Tree Int)'
      In the instance declaration for 'Functor (Tree Int)'
```

Klasa Applicative

Klasa Functor pozwala nam „podnieść” (ang. to lift) zwykłą funkcję jednoargumentową i zaaplikować ją bezpośrednio do wartości typu Maybe (lub innej instancji klasy Functor).

```
fmap (+2) (Just 3)              -- Just 5
```

fmap nie umożliwia jednak zaaplikowania funkcji do kilku wartości, np.

```
fmap (+) (Just 3) (Just 2)      -- nie da się; dlaczego?
Prelude> :t  fmap (+) (Just 3)
fmap (+) (Just 3) :: Num a => Maybe (a -> a)
```

Funkcja, którą chcemy zaaplikować do wartości z kontekstu Maybe sama należy do tego kontekstu.

Powyższy problem rozwiązuje klasa Applicative dziedzicząca z klasy Functor.

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

Maybe jest instancją klasy Applicative, więc:

```
pure (+) <*> Just 3 <*> Just 2      -- Just 5
```

Klasa Applicative cd.

Porównaj typy:

```
($)              ::                          (a -> b) ->   a ->  b
(<$>), fmap :: Functor f      =>   (a -> b) -> f a -> f b
(<*>)            :: Applicative f => f (a -> b) -> f a -> f b
```

- ($) jest aplikacją funkcji
- (<$>) jest aplikacją funkcji „podniesioną” do funktorów
- (<*>) jest aplikacją funkcji „podniesioną” do funktorów, gdzie sama funkcja jest również zagłębiona w kontekście f

Klasa Applicative cd.

Zadeklarujmy nasz typ Maybe' jako instancję klasy Applicative.

```
instance Applicative Maybe' where
  pure f = Just' f
  Nothing'  <*> _        =  Nothing'
  _         <*> Nothing' = Nothing'
  (Just' f) <*> (Just' x) = Just' (f x)
```

Teraz też możemy wykonać:

```
pure (+) <*> Just' 2 <*> Just' 3      -- Just' 5
```

W analogiczny sposób można „podnosić" inne funkcje:

```
pure (,) <*> Just' 2 <*> Just' 3      -- Just' (2,3)
```

To samo możemy zrobić ze złożeniem funkcji:

```
pure (.) <*> Just' (+2) <*> Just' (+3) <*> Just' 1    -- Just' 6
```

lub inaczej:

```
Just' (.) <*> Just' (+2) <*> Just' (+3) <*> Just' 1    -- Just' 6
```

Klasa Applicative cd.

Jak jest ewaluowane ostatnie wyrażenie?

```
    pure (.) <*> Just' (+2) <*> Just' (+3) <*> Just' 1
=   Just' (.) <*> Just' (+2) <*> Just' (+3) <*> Just' 1
=   Just' ((.)  (+2))          <*> Just' (+3) <*> Just' 1
=   Just' ((.)  (+2) (+3))                    <*> Just' 1
=   Just' ((.)  (+2) (+3) 1)
=   Just' (((+2) . (+3)) 1)
=   Just' ((+2)  ((+3) 1))
=   Just' ((+2)  4)
=   Just' 6
```

Klasa Monad

Klasa Monad od wersji 7.10 GHC dziedziczy z klasy Applicative.

```
class  (Applicative m) => Monad m  where
   (>>=)   :: m a -> (a -> m b) -> m b
   (>>)    :: m a -> m b -> m b
   return  :: a -> m a
   fail    :: String -> m a

   m >> k  =  m >>= \_ -> k
   fail s  = error s
   return  = pure       -- od wersji 7.10 GHC
```

Typ m w definicji klasy Monad jest gatunku * → *. Domyślne implementacje metod (>>) i return praktycznie nigdy nie wymagają zmiany. Instancje klasy Monad muszą tylko zaimplementować (>>=). Operator bind (>>=) łączy wartość monadyczną (m a) z funkcją (a -> m b), nazywaną **funkcją monadyczną** (ang. monadic function).
Notacja "do", omawiana na wykładzie poświęconym typowi IO, który jest instancją klasy Monad, odnosi się do wszystkich instancji tej klasy.

Klasa Monad cd.

Oprócz IO, w standardowym preludium również Maybe oraz listy zostały zadeklarowane jako instancje klasy Monad. Metoda fail dla list zwraca pustą listę [], dla Maybe zwraca Nothing, a dla IO zgłasza wyjątek user error.

Instancje klasy Monad powinny spełniać następujące aksjomaty (por. wykład 10, str.27):

return x >>= f = f x

m >>= return = m

m1 >>= (\x -> m2 >>= \y->m3) = (m1 >>= \x->m2) >>= \y->m3
jeśli x nie jest zmienną wolną w m3

Instancje obu klas: Monad i Functor powinny dodatkowo spełniać aksjomat :

fmap f xs = xs >>= return . f

Wszystkie instancje klasy Monad zdefiniowane w preludium spełniają te aksjomaty.

Klasa Monad cd.

Zadeklarujmy nasz typ Maybe' jako instancję klasy Monad.

```
instance Monad Maybe' where
-- return  =  pure         -- domyślna implementacja
   Nothing'  >>=  _     =  Nothing'
   (Just' x)  >>=  f     =    (f x)
```

Teraz też możemy wykonać np.:

```
Just' 10 >>= \x -> Just' (show (x::Int))    -- Just' "10"

Nothing' >>= \x -> Just' (show (x::Int))   -- Nothing'
```

Porównanie

```
($)      :: (a -> b) ->  a -> b
(<$>) :: Functor f => (a -> b) -> f a -> f b          lub fmap
pure   :: Applicative f => a -> f a
(<*>) ::  Applicative f => f (a -> b) -> f a -> f b
(>>=) :: Monad m => m a -> (a -> m b) -> m b     (łączność lewostronna)
(=<<) :: Monad m => (a -> m b) -> m a -> m b     (łączność prawostronna, kolejność argumentów
                                                  odpowiada aplikacji)
```

- Functor umożliwia mapowanie funkcji w kontekście obliczeniowym.
- Applicative umożliwia aplikację funkcji z kontekstu do wartości z kontekstu (<*>) i wkładanie wartości do kontekstu obliczeniowego (pure).
- Monad umożliwia składanie (sekwencjonowanie) obliczeń (funkcji monadycznych) w kontekście obliczeniowym, przy czym kolejny krok obliczeń zależy od wyniku poprzedniego kroku.

We wszystkich przypadkach w kontekście obliczeniowym (ang. computational context) możliwe są efekty obliczeniowe (ang. effects, computational effects). W ten sposób czysto funkcyjny język programowania, jakim jest Haskell, umożliwia kontrolowane programowanie z efektami (ang. effectful programming).