Typy danych

Inspiracja

Materiały do ćwiczeń będą oparte w dużej mierze na materiałach dr. Sławomira Bakalarskiego (do tego samego kursu).

Nowe typy

Używając `type` możemy zdefiniować nową nazwę typu dla typu już istniejącego (w ten sposób zdefiniowany jest też `String`). Przykładowo:

```
type Imie = String
```

Po takiej definicji możemy używać nowego typu, na przykład

```
jan :: Imie
jan = "Jan"
```

Ponieważ wprowadziliśmy tak na prawdę nową nazwę dla istniejącego już typu, w funkcjach używających `String` możemy użyć zdefiniowanego w taki sposób typu `Imie`. Podobnie funkcja, która używa `[Char]` działa dla `String`.

Nowe typy

Używając `newtype` możemy zdefiniować nowy typ za pomocą dokładnie jednego konstruktora przyjmującego dokładnie jeden argument. Na przykład:

```
newtype Imie2 = Imie2 String
```

Zdefiniowanego w ten sposób typu możemy używać np. tak:

```
janek :: Imie2
janek = Imie2 "Janek"
```

W przeciwieństwie do `type` nie możemy używać `Imie2` wymiennie ze `String`. W szczególności nie możemy wypisać nawet powyższej stałej na ekran, ponieważ typ `Imie2` nie jest w klasie `Show`.

Nowe typy

Używając `data` możemy zdefiniować nowy typ używając dowolnej liczby konstruktorów i dowolnej liczby pól. Na przykład:

```
data Rozmiar = S | M | L
maly :: Rozmiar
maly = S
```

lub

```
data Student = Student {imie :: String,
                        nazwisko :: String,
                        nrAlbumu :: Int
                        }
janKowalski = Student {imie="Jan",
 nazwisko="Kowalski", nrAlbumu=1234567}
```

Instancje klasy

Pomocne (chociażby dla możliwości łatwego wyświetlania) jest należenie typu do klasy `Show`. Aby to osiągnąć możemy użyć operatora `instance`:

```
instance Show Rozmiar
    where
    show S = "S"
    show M = "M"
    show L = "L"
```

Aby sprawdzić, jakie funkcje musimy zdefiniować, żeby poprawnie użyć operatora `instance` możemy skorzystać np. z Hoogle (w klasie Show jest prosto, wymagana jest tylko jedna funkcja, w klasie `Num` jest już ich dużo więcej: dodawanie, mnożenie itd.).

deriving

W prostych przypadkach Haskell może "zgadnąć" sensowne implementacje `show` czy ==. Używamy w tym celu operatora `deriving`:

```
data Rozmiar = S | M | L deriving Show
```

W powyższym przypadku `show` zostanie zdefiniowane dokładnie tak, jak na poprzednim slajdzie zrobiliśmy to ręcznie. Możemy też podać więcej, niż jedną klasę, np.:

```
deriving (Show,Eq)
```

W prostych przypadkach, np. dla typu `Student` zdefiniowanego wcześniej Haskell będzie (odpowiednio) wypisywał pola (nazwy i wartości) oraz sprawdzał, czy odpowiadające pola w dwóch podanych instancjach mają te same wartości. Jeśli to, co zostanie przypisane nam nie odpowiada, możemy oczywiście zdefiniować pożądane zachowanie sami.

Przykład: liczby naturalne

Możemy zdefiniować liczby naturalne w następujący sposób:

```
data Naturalne = Zero | Nastepnik Naturalne
 deriving (Show,Eq)
```

Przykładowymi liczbami naturalnymi są wtedy:

```
zero = Zero
jeden = Nastepnik Zero
dwa = Nastepnik (Nastepnik Zero)
```

Zdefiniowanych typów możemy używać w funkcjach, na przykład:

```
natToInt :: Naturalne -> Integer
natToInt Zero = 0
natToInt (Nastepnik x) = (natToInt x) + 1
```

Przykład: drzewo binarne

Możemy zdefiniować drzewo binarne w następujący sposób:

```
data Tree a = Empty | Node a (Tree a) (Tree a)
```

Drzewo zawierające w wierzchołkach liczby całkowite, złożone tylko z korzenia, w którym jest zero możemy wtedy zdefiniować następująco:

```
drzewo :: Tree Integer
drzewo = Node 0 Empty Empty
```