Haskell – wprowadzenie c.d.

Inspiracja

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

Klasy typów – wprowadzenie

W Haskellu istnieje możliwość definiowania funkcji (i stałych!) bardziej ogólnie, niż dla jednego wybranego typu. Można się natknąć na to samemu używając `:t`, na przykład na dodawaniu. Możemy przeczytać, że:

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

`Num` oznacza tutaj klasę typów, w tym konkretnych wypadku chodzi o klasę typów, na których można wykonywać działania arytmetyczne. Przykładami typów z klasy `Num` są `Int`, `Integer` czy `Double`. W Haskellu klasy typów mogą wymuszać zdefiniowanie odpowiedniego działania w odpowiednim kontekście. Czasem, mimo, że coś nie jest wymagane jest "oczekiwane", na przykład rozdzielność mnożenia względem dodawania w klasie `Num` (szczegóły dla konkretnych klas można sprawdzić w dokumentacji, np. Hoogle).

Klasy typów – wprowadzenie

Klasy mogą być definiowane w oderwaniu od innych klas lub mogą zachodzić między nimi relacje, np. `Fractional` jest podklasą `Num` (wymaga zdefiniowania operatora "/"), a `Floating` jest podklasą `Fractional` (wymaga dodatkowo zdefiniowania `exp`). Kompilator pilnuje, czy możemy wykonać operację, którą chcemy wykonać w danym kontekście. Gdybyśmy na przykład chcieli skompilować funkcję

```
napiszDodawanie :: Num a => a -> a -> String
napiszDodawanie x y = "Liczba " ++ show x ++
 " powiekszona o " ++ show y ++ " daje nam " ++ show (x+y)
```

dostaniemy błąd, ponieważ od liczby nie wymagamy możliwości przedstawienia jako `String`.

Klasy typów – wprowadzenie

Kompilator podpowie nam też, że możliwym rozwiązaniem problemu jest dorzucenie klasy `Show` w definicji. Istotnie,

```
napiszDodawanie :: (Num a, Show a) => a -> a -> String
napiszDodawanie x y = "Liczba " ++ show x ++
 " powiekszona o " ++ show y ++ " daje nam " ++ show (x+y)
```

jest już poprawną definicją. Na podobne problemy możemy natknąć się porównując, czy obiekty są takie same (klasa `Eq`) lub czy jeden jest większy od drugiego (Klasa `Ord`, jest w szczególności podklasą `Eq`).

Klasy typów – wprowadzenie

Jeśli będziemy definiować własne funkcje, ale nie napiszemy wprost sygnatury, to Haskell będzie "zgadywał" typy, nie zostawi "najbardziej ogólnego przypadku". Np. definicja

```
dodawanie = (+)
```

spowoduje powstanie funkcji

```
dodawanie :: Integer -> Integer -> Integer
```

Jest to zachowanie analogiczne do zgadywania typu stałej. Pisząc funkcje czy stałe samemu dobrym pomysłem może być więc pisanie sygnatury za każdym razem, ponieważ nie zyskujemy automatycznie pełnej ogólności nie pisząc nic. Możemy jednak sami zdefiniować

```
dodawanie :: Num a => a -> a -> a
dodawanie :: Integer -> Integer -> Integer
```

Klasy typów – wprowadzenie

Warto zwrócić jeszcze uwagę, że w danej sygnaturze, jeśli nastąpiło dopasowanie np. do a jakiegoś typu, to musi się ono zgadzać na każdym wystąpieniu a. Na przykład, jeśli napiszemy funkcję

```
wypiszRzeczy :: Show a => a -> a -> String
wypiszRzeczy x y = "Najpierw " ++
 show x ++ ", potem " ++ show y
```

to nie zadziała ona wywołana na argumentach `Int` oraz `Integer` mimo, że oba są w klasie `Show`. Możemy jednak dostać taką funkcjonalność:

```
wypiszRzeczy :: (Show a, Show b) => a -> b -> String
wypiszRzeczy x y = "Najpierw " ++
 show x ++ ", potem " ++ show y
```

Klasy typów – wprowadzenie

Możemy też definiować funkcje używając **zupełnie dowolnego** typu, nie musimy się zawężać do żadnej klasy. Przykładem jest wbudowana funkcja składania

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

o definicji odpowiadającej standardowemu składaniu funkcji, tzn.

$$(f \circ g)(x) = f(g(x))$$

przy standardowych założeniach co do dziedziny i przeciwdziedziny (które widać odzwierciedlone w sygnaturze).

Gdzie

Przy tworzeniu list bardzo pomocny może być operator "gdzie". Za pomocą definicji

```
kwadraty :: [Int]
kwadraty = [x * x | x <- [1..10]]
```

utworzymy listę kwadratów pierwszych 10 dodatnich liczb naturalnych. Odpowiada to oczywiście zapisowi matematycznemu

$$\{x^2 \mid x \in \{1, \ldots, 10\}\}.$$

Podobnie możemy uzyskać listy spełniające więcej warunków, na przykład kwadraty tylko liczb parzystych:

```
kwadraty2 :: [Int]
kwadraty2 = [x * x | x <- [1..10], even x]
```

Inne funkcje na listach

Przykłady innych funkcji związanych z listami:

- `repeat` – tworzy nieskończona listę powtarzając podany argument
- `cycle` – tworzy nieskończona listę powtarzając podaną listę
- `elem <obiekt> <lista>` – sprawdza, czy obiekt jest na liście
- `filter <warunek> <lista>` – zwraca podlistę elementów spełniających warunek