Paradygmaty programowania - ćwiczenia
Lista 10

Na wykładzie 4 polimorficzne niemutowalne drzewa binarne zostały zdefiniowane jak niżej:

```
sealed trait BT[+A]
case object Empty extends BT[Nothing]
case class Node[+A](elem:A, left:BT[A], right:BT[A]) extends BT[A]
```

Wykład 10 pozwolił w pełni zrozumieć znaczenie notacji +A. Dzięki temu zdefiniowane drzewa są kowariantne, czyli jeśli T1 <: T2, to BT[T1] <: BT[T2] oraz Node[T1] <: Node[T2].
W szczególności, ponieważ typ Nothing jest podtypem każdego typu, to BT[Nothing] jest podtypem BT[T] dla każdego typu T i obiekt singletonowy Empty może reprezentować drzewo puste dowolnego typu.
W przykładzie na początku listy 8 zdefiniowaliśmy klasę BT<A> dla polimorficznych niemutowalnych drzew binarnych w języku Java. W celu zdefiniowania unikatowego drzewa pustego EMPTY musieliśmy użyć typu surowego BT, ponieważ w Javie nie ma typu, będącego podtypem każdego typu, np. Nothing.

Popatrzmy, co się stanie, jeśli w definicji typu BT i kasy Node pominiemy znak +.

```
sealed trait BT[A]
case object Empty extends BT[Nothing]
case class Node[A](elem:A, left:BT[A], right:BT[A]) extends BT[A]
```

Ten kod się skompiluje, jednak próba utworzenia jakiegokolwiek drzewa spowoduje błąd typu, np.

```
scala> val t: BT[Int] = Node(1, Empty, Empty)
                              ^
       error: type mismatch;
        found   : Empty.type
        required: BT[Int]
       Note: Nothing <: Int (and Empty.type <: BT[Nothing]), but trait BT is invariant in type A.
       You may wish to define A as +A instead. (SLS 4.5)
```

(To jest komunikat Scali 2, zawierający więcej informacji niż komunikat Scali 3).

Oczywiście (jak pokazano na wykładzie 4) można zdefiniować ten typ bez użycia typu Nothing:

```
sealed trait BT[A]
case class Empty[A]() extends BT[A]
case class Node[A](elem:A, left:BT[A], right:BT[A]) extends BT[A]
```

Teraz jednak drzewa puste dla każdego typu będą reprezentowane przez różne obiekty:

```
scala> Empty[Int]() eq Empty[Int]()
res1: Boolean = false
```

Ponadto składnia jest bardziej uciążliwa (trzeba jawnie podawać typ każdego tworzonego drzewa pustego i używać pustej listy argumentów).

Paradygmaty programowania - ćwiczenia
Lista 10

Wszystkie programy mają być napisane w języku Scala.

1. Klasa GenericCellImm kompiluje się jako klasa inwariantna i kowariantna.

```
scala> class GenericCellImm[T] (val x: T)
// defined class GenericCellImm

scala> class GenericCellImm[+T] (val x: T)
// defined class GenericCellImm
```

Natomiast klasa GenericCellMut kompiluje się tylko jako klasa inwariantna.

```
scala> class GenericCellMut[T] (var x: T)
// defined class GenericCellMut
```

Wersja kowariantna powoduje błąd kompilacji.

```
scala> class GenericCellMut[+T] (var x: T)
-- Error:
1 |class GenericCellMut[+T] (var x: T)
  |                            ^
  |covariant type T occurs in contravariant position in type T of parameter x_=
```

a) Wyjaśnij powód tego błędu (należy **dokładnie** wyjaśnić powyższy komunikat).
b) Czy można się pozbyć tego błędu? Uzasadnij swoją odpowiedź.
c) Czy wersja kontrawariantna skompiluje się? Uzasadnij swoją odpowiedź.
class GenericCellMut[-T] (private var x: T)

2. Poniższa definicja powoduje błąd kompilacji.

```
scala> abstract class Sequence[+A]:
     |   def append(x: Sequence[A]): Sequence[A]
     |
-- Error:
2 |  def append(x: Sequence[A]): Sequence[A]
  |                ^^^^^^^^^^^^^^
  |covariant type A occurs in contravariant position in type Sequence[A] of parameter x
```

Wyjaśnij przyczynę tego błędu. Czy można się go pozbyć?.

3. Zdefiniuj klasę generyczną dla **kowariantnej** kolejki niemodyfikowalnej, reprezentowanej przez parę list (patrz lista 7, zadanie 1b).
Wskazówka. Wzoruj się na klasie dla stosu z wykładu 10 (str. 9 i 28).

4. Zdefiniuj generyczną inwariantną metodę copy dla sekwencyjnych kolekcji modyfikowalnych scala.collection.mutable.Seq na wzór programu napisanego w Javie (wykład 10, str. 31).
Wskazówka. Wykorzystaj metodę foreach oraz metodę update (patrz Scala API).