

Programowanie funkcyjne i współbieżne

Lista 7

Algebraiczna specyfikacja kolejki nieskończonej

Sygnatura

```
empty      : -> Queue
enqueue    : Elem * Queue -> Queue
first      : Queue -> Elem
firstOption: Queue -> Option[Elem]
dequeue    : Queue -> Queue
isEmpty    : Queue -> bool
```

Aksjomaty

```
For all q:Queue, e1,e2: Elem
isEmpty (enqueue (e1,q))      = false
isEmpty (empty)               = true
dequeue (enqueue (e1,enqueue (e2,q))) =
    enqueue (e1, dequeue (enqueue (e2,q)))
dequeue (enqueue (e1,empty))  = empty
dequeue (empty)               = empty
first (enqueue (e1,enqueue (e2,q))) = first (enqueue (e2,q))
first (enqueue (e1,empty))     = e1
first (empty)                  = ERROR
firstOption (enqueue (e1,enqueue (e2,q))) = firstOption (enqueue (e2,q))
firstOption (enqueue (e1,empty))      = Some (e1)
firstOption (empty)                  = None
```

1. Zdefiniuj klasę generyczną dla kowariantnej kolejki niemodyfikowalnej, reprezentowanej przez dwie listy.

W ten sposób reprezentowane są kolejki niemodyfikowalne w językach czysto funkcyjnych, a także w Scali (patrz dokumentacja).

Wskazówka. Wzoruj się na klasie dla stosu z wykładu 7 (str. 8 i 27) oraz dokumentacji `scala.collection.immutable.Queue` (zaimplementuj tylko metody z powyższej specyfikacji).

Zdefiniuj obiekt towarzyszący z metodami `apply` i `empty`.

Utworzenie nowej kolejki ma być możliwe na cztery sposoby:

```
new MyQueue  MyQueue()  MyQueue.empty  MyQueue('a', 'b', 'c')
```

Para list $([x_1; x_2; \dots; x_m], [y_1; y_2; \dots; y_n])$ reprezentuje kolejkę $x_1 x_2 \dots x_m y_n \dots y_2 y_1$. Pierwsza lista reprezentuje początek kolejki, a druga – koniec kolejki. Elementy w drugiej liście są zapamiętane w odwrotnej kolejności, żeby wstawianie było wykonywane w czasie stałym (na początek listy). $enqueue(y, q)$ modyfikuje kolejkę następująco: $(xl, [y_1; y_2; \dots; y_n]) \rightarrow (xl, [y; y_1; y_2; \dots; y_n])$. Elementy w pierwszej liście są pamiętane we właściwej kolejności, co umożliwia szybkie usuwanie pierwszego elementu. $dequeue(q)$ modyfikuje kolejkę następująco: $([x_1; x_2; \dots; x_m], yl) \rightarrow ([x_2; \dots; x_m], yl)$. Kiedy pierwsza lista zostaje opróżniona, druga lista jest odwracana i wstawiana w miejsce pierwszej: $([], [y_1; y_2; \dots; y_n]) \rightarrow ([y_n; \dots; y_2; y_1], [])$. Reprezentacja kolejki jest w postaci normalnej, jeśli nie wygląda tak: $([], [y_1; y_2; \dots; y_n])$ dla $n \geq 1$. **Wszystkie operacje kolejki mają zwracać reprezentację w postaci normalnej**, dzięki czemu pobieranie wartości pierwszego elementu nie spowoduje odwracania listy. Odwracanie drugiej listy po opróżnieniu pierwszej też może się wydawać kosztowne. Jeśli jednak oszacujemy nie koszt pesymistyczny (oddzielnie dla każdej operacji kolejki), ale koszt zamortyzowany (uśredniony dla całego czasu istnienia kolejki), to okaże się, że zamortyzowany koszt operacji wstawiania i usuwania z kolejki jest stały. Para list to najefektywniejsza reprezentacja kolejki niemutowalnej.

2.

```
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]
```

Dla drzew binarnych, zdefiniowanych powyżej, napisz funkcję `breadthBT[A] : BT[A] => List[A]` obchodzącą drzewo binarne wszerz i zwracającą listę wartości, przechowywanych w węzłach drzewa. Wykorzystaj kolejkę z zadania 1.

=====

Wszystkie definicje oraz proste testy w obiekcie singletonowym z metodą `main` umieść w pliku `Lista7.scala`.

Zalecane jest utworzenie projektu Scala IDEA (a nie sbt). Scala worksheet jest doskonały dla programów funkcyjnych, mniej wygodny dla obiektowych, a zupełnie nie nadaje się dla programów współbieżnych (o których będzie mowa od następnego wykładu).

Struktura pliku `Lista7.scala` ma być dokładnie taka, jak pokazano poniżej.

Lista7.scala

```
// imię i nazwisko
// Zadanie 1
```

```
class MyQueue[+T]
...
```

```
object MyQueue
...
```

```
// Zadanie 2
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]
```

```
object Lista7{
  def main (args: Array[String]): Unit = {
    ... proste testy dla kolejki (utwórz ją na cztery podane sposoby)
    ... obejście wszerz drzewa pustego i dwóch drzew z wykładu 4 (t i tt)
  }
}
```

```
def breadthBT[A]
...
```

```
val t = Node(1,Node(2,Empty,Node(3,Empty,Empty)),Empty)
val tt = ...      drzewo z wykładu 4, str. 17
}
```