Лабораторная работа № 6

Исследование особенностей реализации параметризованных классов и функторов в языке Scala

1. Цель работы

Исследовать способы реализации и использования параметризованных классов и функторов в языке Scala, получить практические навыки их применения.

2. Общие положения

2.1. Параметризованные классы

Параметризованные классы – классы, которые могут оперировать над значениями различных типов.

В качестве примера реализуем класс, который получает в конструкторе значение и имеет метод для печати полученного значения. Особенность класса заключается в том, что он создан не для работы с каким-то конкретным типом, а может работать со всеми типами.

```
case class MyPrinter[T](attribute: T) {
 def print() = println(attribute)
}

// Пример 1
MyPrinter(1).print()          // 1
// Пример 2
MyPrinter("test").print()     // test
// Пример 3
MyPrinter[Double](1).print()  // 1.0
```

При создании класса не обязательно указывать конкретный тип параметра (примеры 1 и 2), но, если очень хочется, то можно (пример 3).

Добавим в класс MyPrinter метод, который будет печатать два значения - атрибут класса и аргумент метода:

```
case class MyPrinter[T](attribute: T) {
 def print()        = println(attribute)
 def print2(arg: T) = println(attribute + " and " + arg)
}

MyPrinter("test").print2("testtest")  // test and testtest
```

Метод print2() может принимать аргумент только того типа, которому принадлежит атрибут класса (и там и там значения имеют один и тот же тип T). Попытка распечатать вместе String и Double с помощью MyPrinter(“test”).print2(3.0) является ошибкой.

Помимо классов, можно также параметризовать и функции, так что вышеуказанную ошибку можно решить следующим способом:

```
case class MyPrinter[T](attribute: T) {
 def print()           = println(attribute)
 def print2(arg: T)    = println(attribute + " and " + arg)
 def print3[R](arg: R) = println(attribute + " and " + arg)
}

// Пример 1
MyPrinter("test").print3(3)                 // test and 3
// Пример 2
MyPrinter("test").print3[Double](3)         // test and 3.0
// Пример 3
MyPrinter("test").print3("another string")   // test and another
string
```

Как видно из примера 3, типы T и R не обязательно должны быть различны.

2.2. Ковариативный функтор

С точки зрения синтаксиса, ковариативный функтор это такой тип X, который параметризирован типом T и имеет метод (назовём его map) с особой сигнатурой:

```
trait X[T] {
 def map[R](f: T => R): X[R]
}
```

Этот тип можно представлять себе как источник, откуда можно **что-то** взять, и это **что-то** имеет тип T. Но, если надо получить не тип T, а тип R, то мы на выходе этого источника ставим преобразующую функцию из T в R и получаем новый источник, который даёт тип R.

Примерами ковариативных функторов являются Scala классы Option и List.

Для того, чтобы тип X считался ковариативным функтором, также должны соблюдаться два правила.

Правило №1: Закон идентичности (Identity Law)
Пусть fun[T] является ковариативным функтором, тогда

fun.map(identity) ≡ fun

где identity - функция, которая возвращает свой аргумент (def identity[A](x: A) = x).

Другими словами: fun.map(identity) не должно ничего менять внутри функтора.

Например, следующий класс не является ковариативным функтором, т.к. после применения fun.map(identity) изменится значение переменной inc.

```
class Example[T](value: T, inc: Int = 0) {
 def map[R](f: T => R): Example[R] = new Example[R](f(value), inc
+ 1)
}
```

Правило №2: Закон композиции (Composition Law)
Пусть fun[T] является ковариативным функтором, f - функция типа T => R, g - функция типа R => Q, тогда:

fun.map(f).map(g) ≡ fun.map(f andThen g)

где f andThen g возвращает функцию, которая аналогична последовательному применений функций f и g. То есть (f andThen g)(x) то же самое что и g(f(x)).

Иными словами, функтор, который отображают функцией f, а затем g, эквивалентен функтору, который отображают композицией функций f и g.

Рассмотрим оба правила на примере бинарного дерева:

```
trait Tree[T] {
 def map[R](f: T => R): Tree[R]
}
case class Node[T](value: T, fst: Tree[T], snd: Tree[T]) extends
Tree[T] {
 def map[R](f: T => R) = Node(f(value), fst.map(f), snd.map(f))
}
case class Leaf[T](value: T) extends Tree[T] {
 def map[R](f: T => R) = Leaf(f(value))
}
```

В данной реализации при отображении дерева с помощью произвольной функции сохраняется его структура и обновляется значения в узлах и листьях. Изменим метод map следующим образом:

```
case class Node[T](value: T, fst: Tree[T], snd: Tree[T]) extends
Tree[T] {
 def map[R](f: T => R) = Node(f(value), snd.map(f), fst.map(f))
}
```

С данной реализацией метода map бинарное дерево не является ковариативным функтором, т.к. с каждым отображением правое и левое поддерево узлов меняется местами.

Следовательно, tree.map(identity) вернёт дерево с тем же значением элементов, но поменяными поддеревьями (не соблюдается первое правило). В свою очередь, tree.map(f).map(g) изменит структуру дерева дважды (по одному разу на каждом вызове map), а tree.map(f andThen g) - всего один раз (не соблюдается второе правило).

Чтобы увидеть пользу от применения ковариативных функторов, рассмотрим следующий код:

```
val f = (x: Double) => x*2 + 3
val g = (x: Double) => x/5 - 7

// Список из Double
val list = (1 to 1000000 toList) map (_.toDouble)

// Пример 1
list.map(f).map(g)
// Пример 2
list.map(f andThen g)
```

В первом примере мы 2 раза проходим по всему списку чисел и 2 раза применяем функцию. Во втором примере мы 1 раз проходим по всему списку чисел и 1 раз применяем функцию. Зная то, что List является ковариативным функтором и зная правила, которым должен удовлетворять ковариативный функтор, мы можем оптимизировать первый пример и быть уверенными, что результат останется прежним.

2.3. Контравариативный функтор

С точки зрения синтаксиса, контравариативный функтор это такой тип X, который параметризирован типом T и имеет метод (назовём его contramap) с особой сигнатурой:

```
trait X[T] {
 def contramap[R](f: R => T): X[R]
}
```

Если ковариативный функтор можно представить как источник, то контравариативный функтор можно представить как приёмник, который принимать тип T. Если на входе (не на выходе, как это было с ковариативным функтором) поставить преобразующую функцию из R в T, то получится новый приёмник, который принимает тип R.

Для того, чтобы тип X считался контравариативным функтором, также должны соблюдаться два правила.

Правило №1: Закон идентичности (Identity Law)

Пусть fun[T] является контравариативным функтором, тогда

fun.contramap(identity) ≡ fun

где identity - функция, которая возвращает свой аргумент (def identity[A](x: A) = x).

Другими словами: fun.contramap(identity) не должно ничего менять внутри функтора.

Правило №2: Закон композиции (Composition Law)
Пусть fun[T] является контравариативным функтором, f - функция типа Q => R, g - функция типа R => T, тогда:

fun.contramap(g).contramap(f) ≡ fun.contramap(f andThen g)

где f andThen g возвращает функцию, которая аналогична последовательному применений функций f и g. То есть (f andThen g)(x) то же самое что и g(f(x)).

Иными словами, последовательное отображение контравариантного функтора парой функций эквивалентно отображению инвертированной композицией функций.

2.4. Инвариативный функтор

Инвариативный функтор это объединение ковариантивного и контравариативного функтора (и источник и приёмник одновременно). С точки зрения синтаксиса он представляет собой следующее:

```
trait X[T] {
 def xmap[R](f: T => R, g: R => T): X[R]
}
```

Для получение нового инвариативного функтора с типом R необходимо предоставить функции преобразования на входе (g: R => T) и на выходе (f: T => R).

Инвариативный функтор также должен соблюдать два правила.

Правило №1: Закон идентичности (Identity Law)
Пусть fun[T] является инвариативным функтором, тогда

fun.xmap(identity, identity) ≡ fun

где identity - функция, которая возвращает свой аргумент (def identity[A](x: A) = x).

Другими словами: fun.xmap(identity) не должно ничего менять внутри функтора.

Правило №2: Закон композиции (Composition Law)
Пусть fun[T] является инвариативным функтором, тогда для данного функтора и функций f1: T => R, g1: R => T, f2: R => Q, g2: Q => R должно выполняться следующее соотношение:

fun.xmap(f1, g1).xmap(f2, g2) ≡ fun.xmap(f2 compose f1, g1 compose g2)

где f compose g возвращает функцию, которая аналогична последовательному применений функций g и f. То есть (f compose g)(x) то же самое что и f(g(x)).

3. Порядок выполнения работы

3.1. Написать типизированный класс Box[T] который позволяет сохранять в себя значение типа T (метод save) и получать сохранённое значение (метод get).

3.2. Преобразовать класс Box из п.1 в ковариативный функтор. Доказать, что он является ковариативным функтором.

3.3. Преобразовать класс Box в контравариативный функтор. Доказать, что он является контравариативным функтором.

3.4. Преобразовать класс Box в инвариативный функтор. Доказать, что он является инвариативным функтором.

4. Содержание отчета

4.1. Цель работы.
4.2. Задание на работу.
4.3. Исходный код программы.
4.4. Результаты работы программы.
4.5. Выводы по работе.