Polymorphisme

6 juin 2017

Nous explorons aujourd'hui les deux facettes du polymorphisme en Scala, qui permettent de développer du code modulaire : le polymorphisme paramétrique (ou généricité) et le polymorphisme ad-hoc (ou héritage).

1 Paramétricité

1. Donner l'implémentation des fonctions génériques suivantes :

```
def apply[A,B](f:A=>B, x:A): B
def compose[A,B,C](f:A=>B, g:B=>C): A=>C
def curry[A,B,C](f:(A,B)=>C): A=>(B=>C)
def uncurry[A,B,C](f:A=>B=>C): (A,B)=>C
def distrib[A,B,C](f:A=>B, g:A=>C): A=>(B,C)
def swap[A,B](x:(A,B)): (B,A)
def triplet[A,B,C](x:((A,B),C)): (A,B,C)
```

   Y a-t-il d'autres implémentations pour ces signatures ?

2. Donner *toutes* les implémentations ayant pour signature :

```
def choose[A](x:(A,A)): A
def swap2[A](x:(A,A)): (A,A)
def apply2[A](f:A=>A, x:A): A
```

3. Définir une classe fonctionnelle `Stack[A]` des piles d'objets de type `A`. Elle aura les méthodes `def push(x:A): Stack[A]`, et `def pop: (A, Stack[A])` ; on utilisera de simples `List` pour les représenter. Evaluer les déclarations :

```
val s : Stack[Int] = (new Stack(List())).push(1).push(2).push(3)
val t : Stack[Any] = s.push("hello")
```

   Pourquoi la deuxième échoue-t-elle ?

2 Héritage

1. Définir deux fonctions `f: Int => String` et `g: String => Int` quelconques. Définir une fonction `switch` à un argument qui appelle soit `f` soit `g` aléatoirement [1]. Quel est sa signature la plus précise ?
2. Définir une liste d'entiers `l=(1,2,3)` ; ajoutez-y l'élement `"hello"`. Quel est le type de la liste résultante ? Même question avec les tableaux d'entiers (`Array[Int]`).
3. Définir une fonction `def reverse(xs:List[Any]) = xs.reverse` et l'appliquer à `l`. Même qustion avec les tableaux d'objets (`Array[Any]`).
4. Reprendre le code de la classe `Stack` et la rendre co-variante. Que se passe-t-il si vous essayez de la rendre contra-variante ?

1. `new scala.util.Random()).nextBoolean`

3 Expression intrinsèquement typée

Nous allons maintenant définir une structure de données arborescente pour représenter des expressions arithmétiques (ex : $(2+2) \leq 4$ ou $\mathit{true} \wedge (1 \leq 0)$), comme au TP2 Exercice 2. À la différence de celui-ci, on se servira du système de types de Scala pour interdir la formation d'*expressions mal typées*, au sens du typage standard des expressions booléennes et entières ; par exemple $2 + \mathit{true}$ ou $42 \wedge (2 \leq 3)$ sont mal typées.

1. On définit dans un premier temps une classe `Expr` des expressions, constituée des constantes entières (1,2,3...) et booléennes (*true* et *false*), ainsi que les opérateurs binaires + (addition entière), $\wedge$ ("et" booléen), $\leq$ (comparaison d'entiers), et les opérateurs unaires $-$ (opposé d'un nombre entier) et $\neg$ (négation booléenne) :

```
sealed abstract class BinOperator
case object Add extends BinOperator
case object And extends BinOperator
case object Le extends BinOperator

sealed abstract class UnOperator
case object Minus extends UnOperator
case object Not extends UnOperator

sealed abstract class Expr
case class Num(n:Int) extends Expr
case class Bool(n:Boolean) extends Expr
case class UnOp(op:UnOperator, arg:Expr) extends Expr
case class BinOp(op:BinOperator, left:Expr, right:Expr) extends Expr
```

   Définir les valeurs de type `Expr` qui représente les expressions $2 + 2 \leq 4$ et $2 + \mathit{true} \leq \mathit{false}$.

2. Représentons le type d'une expression par un type Scala : `Boolean` pour les expressions booléennes, `Int` pour les expressions entières.
   (a) Ajouter deux paramètres de type O et R à la classe `UnOperator` ; O est le type de la sous-expression attachée à l'opérateur, R est le type de l'expression construite avec cet opérateur. Compléter la déclaration de chaque opérateur unaire selon sa sémantique.
   (b) Faire de même pour la classe `BinOperator` : y ajouter trois paramètres de type O_1 (type de la sous-expression de gauche), O_2 (type de la sous-expression de droite) et R (type de l'expression résultante).
   (c) Enfin, ajouter à `Expr` un paramètre de type R représentant le type de l'expression (Boolean) ; Compléter les déclarations des sous-classes `Num`, `Bool`, `UnOp`, `BinOp` en conséquence.
3. Adapter les exemples définis en 1. ; Le deuxième ne devrait plus être accepté, car il est mal typé.
4. Définir les fonctions qui représentent les connecteurs $=$, $\vee$, $>$, grâce aux équivalences :
   — $a > b$ est équivalent à $\neg(a \leq b)$
   — $a = b$ est équivalent à $a \leq b \wedge b \leq a$
   — $a \vee b$ est équivalent à $\neg(\neg a \wedge \neg b)$
5. Définir une méthode `eval` dans la classe `Expr[A]` qui calcule l'expression courante et renvoit sa valeur de type `A`, c'est-à-dire soit `Boolean` soit `Int`. [2]

2. Attention : il faut employer le style qui consiste à définir chaque cas de la fonction dans sa sous-classe. Définir `eval` entièrement dans `Expr` par pattern-matching ne fonctionnera pas (pourquoi ? essayez !).