Programmation Objet : Concepts Avancés

Programmation typée avancée en Scala

Cours: Gustavo **Petri** TD: Guilhem **Jaber**

gpetri@informatique.univ-paris-diderot.fr

Slides: Yann **Régis-Gianas**

IRIF – Université Denis Diderot – Paris 7

Plan de la partie *“Construire des objets”*

Retour sur `equalizable[A]`

```
trait equalizable[A <: equalizable[A]] {
   def equal (x : A) : Boolean
}
```

est un prédicat qui représente les types « dont l'égalité est testable ».

C'est un **idiome** de Scala que l'on retrouve dans de nombreuses bibliothèques.

Usage dans la bibliothèque de collections de Scala

Quelles contraintes sur les types des identificateurs sont imposées par le fragment de code suivant ?

```
def minors_and_adults (people : ?) =
  people partition (_.age < 18)
```

(Voir http://docs.scala-lang.org/overviews/collections/overview.html.)

Le trait `traversable[A]`

Y-a-t-il une contradiction entre :

```
def map[B](f: (A) => B): Traversable[B]
```

et le type de la valeur `res0` dans l'exemple qui suit :

```
scala> Set(1, 2, 3) map (_ * 2)
res0: Set[Int] = Set(2, 4, 6)
```

?

Étude la classe `List` de Scala

Questions

1. Quelles sont les avantages des structures de données persistentes ?
2. Commentez le code de `map` dans `List.scala`.
3. Déterminez la provenance des méthodes de cette classe.

Plan de la partie *"Construire des objets"*

Définition par compréhension

Comment s'évalue l'expression suivante ?

```
for (enumerators) yield e
```

Sémantique par traduction

Voir http://docs.scala-lang.org/tutorials/FAQ/yield.html

Vues

Commentez la définition suivante :

```
def lazyMap[T, U](coll: Iterable[T], f: T => U) = new Iterable[T] {
        def iterator = coll.iterator map f
}
```

Les structures de données non-strictes

- Quelle est la différence entre un `Stream`, un `Iterator` et une vue ?

Plan de la partie *“Construire des objets”*

Les traits comme prédicats

Définitions par compréhension

Collections pour le traitement parallèle

Inférence de programmes dirigée par les types

Types dépendants

Contrôle du polymorphisme

Des collections pour le parallélisme

[Effort minimal]

- Pour passer d'une évaluation séquentielle à une évaluation parallèle : `.par`
- Les calculs en arrière plan : `future`

(Voir http://docs.scala-lang.org/overviews/parallel-collections/overview.html.)
(Voir http://docs.scala-lang.org/overviews/core/futures.html.)

Plan de la partie *"Construire des objets"*

Plan de la partie *"Construire des objets"*

Si on a :

```
implicit def fromAtoB (x : A) : B = ...
```

alors la fonction :

```
def f[T <% B] (x : T) = ...
```

peut être appelée sur une instance de `A`.

La quantification se lit “pour tout `T` qui peut se voir comme un `B`”.

Interaction avec le polymorphisme

Soit le trait suivant :

```
trait memoryAware[T] {
   def size (x : T) : Int
}
```

permettant de déterminer la taille en mémoire d'un objet de type `T`.

Comment *automatiquement* construire des objets de ce type par induction sur la structure des types ?

Plan de la partie *"Construire des objets"*

Objets implicites

```
abstract class SemiGroup[A] {
  def add(x: A, y: A): A
}
abstract class Monoid[A] extends SemiGroup[A] {
  def unit: A
}
object ImplicitTest extends App {
  implicit object StringMonoid extends Monoid[String] {
    def add(x: String, y: String): String = x concat y
    def unit: String = ""
  }
  implicit object IntMonoid extends Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
  def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
    if (xs.isEmpty) m.unit
    else m.add(xs.head, sum(xs.tail))
  println(sum(List(1, 2, 3)))
  println(sum(List("a", "b", "c")))
}
```

Plan de la partie *"Construire des objets"*

Exemple de classe imbriquée inspiré de http://scala-lang.org/node/115

```
01 class Graph {
02    var nodes: List[Node] = Nil
03    class Node {
04       var connectedNodes: List[Node] = Nil
05       def +→ (node: Node) {
06          if (connectedNodes.find(node.equals).isEmpty) {
07             connectedNodes = node :: connectedNodes
08          } else { println ("Already connected.") }
09       }
10       nodes = this :: nodes
11    }
12 }
```

Exemple de classe imbriquée inspiré de http://scala-lang.org/node/115

```
scala> val g = new Graph
g: Graph = Graph@20442c19

scala> val n1 = new g.Node
n1: g.Node = Graph$Node@36edc33d

scala> val n2 = new g.Node
n2: g.Node = Graph$Node@1a6313e1

scala> val n3 = new g.Node
n3: g.Node = Graph$Node@f5e12

scala> n1 +→ n2

scala> n3 +→ n1

scala> n1 +→ n2
Already connected.
```

```
01 class Graph {
02   private var nodes: List[Node] = Nil
03   class Node {
04     private[Graph] var connectedNodes: List[Node] = Nil
05     def +→ (node: Node) {
06       if (connectedNodes.find(node.equals).isEmpty) {
07         connectedNodes = node :: connectedNodes
08       } else { println ("Already connected.") }
09     }
10     nodes = this :: nodes
11   }
12 }
13 scala> val g1 = new Graph
14 scala> val n1 = new g1.Node
15 n1: g1.Node = Graph$Node@25997c
16 scala> val n2 = new g1.Node
17 n2: g1.Node = Graph$Node@84c1f9
18 scala> n1 +→ n2
19 scala> val g2 = new Graph
20 scala> val n3 = new g2.Node
21 n3: g2.Node = Graph$Node@11e8d5c
22 scala> n3 +→ n1
23 <console>:11: error: type mismatch;
24  found   : g1.Node
25  required: g2.Node
26          n3 +→ n1
```

Typage dépendant

Typage dépendant

Un système de type est dépendant (de valeurs) si la syntaxe des types peut faire référence à des expressions (calculatoires).

- Scala offre une forme restreinte de type dépendant pouvant faire mention de **résultats nommés** de calcul arbitraire, appelés identificateurs stables.

Identificateurs stables

Chemin

En Scala, un chemin est de la forme :

- « `C.this` » où C est une classe.
- « `p.x` » où p est un chemin et x est un membre stable, c'est à dire les membres qui sont des paquetages, des objets ou des valeurs introduites par des définitions (qui n'ont pas un type volatile (i)).
- « `C.super[M].x` » où C est une classe et x un membre stable de la classe mère M de C. (ii)

(Nous reviendrons sur les points (i) et (ii).)

Identificateur stable

En Scala, un identificateur stable est un chemin qui se termine par un identificateur.

Limite d'utilisation des chemins dépendants

```
01 scala> def id (g : Graph) : Graph = g
02 id: (g: Graph)Graph
03
04 scala> val g3 = id (g1)
05 g3: Graph = Graph@1226fe1
06
07 scala> val n4 = new g3.Node
08 n4: g3.Node = Graph$Node@93e86f
09
10 scala> n1 +→ n4
11 <console>:12: error: type mismatch;
12  found : g3.Node
13  required: g1.Node
14          n1 +→ n4
```

Limite d'utilisation des chemins dépendants

```
01 scala> val n5 = n4.asInstanceOf[g1.Node]
02 n5: g1.Node = Graph$Node@93e86f
03 scala> n1 +→ n5
04 scala> n3
05 res20: g2.Node = Graph$Node@11e8d5c
06 scala> val n6 = n3.asInstanceOf[g1.Node]
07 n6: g1.Node = Graph$Node@11e8d5c
08 scala> n6 eq null
09 res21: Boolean = false
10 scala> n1 +→ n6
11
```

⇒ On retrouve alors le typage des classes imbriquées de JAVA.

Solution : utiliser le type le plus précis d'un objet

```
01 scala> val also_g1 = (id (g1)).asInstanceOf[g1.type]
02 also_g1: g1.type = Graph@1226fe1
03
04 scala> val n7 = new also_g1.Node
05 n7: also_g1.Node = Graph$Node@1123968
06
07 scala> n1 +→ n7
08
09 scala> n3 +→ n7
10 <console>:13: error: type mismatch;
11  found : also_g1.Node
12  required: g2.Node
13          n3 +→ n7
```

- Le type le plus précis que l'on peut donner à l'identificateur « `also_g1` » est «`g1.type` ».
- Ce type caractérise exactement la valeur $g1$, c'est un type singleton.

Plan de la partie *"Construire des objets"*

Les traits comme prédicats

Définitions par compréhension

Collections pour le traitement parallèle

Inférence de programmes dirigée par les types

Types dépendants

Contrôle du polymorphisme

Annotation de variances

```
abstract class OutputChannel[-A] {
   def write(x: A): Unit
}
abstract class InputChannel[+A] {
   def read : A
}
```

Ici, on indique que `OutputChannel` est **contravariant** sur son premier paramètre tandis que `InputChannel` est **covariant** sur son premier paramètre.

Affaiblissement du type à l'aide d'une borne inférieure

Si on concatène à une séquence de `A` une autre séquence d'éléments d'un sur-type `B` de `A`, on obtient une séquence de `B`.

```
abstract class Sequence[+A] {
  def append[B >: A](x: Sequence[B]): Sequence[B]
}
```