

Scala : un langage de programmation multi-paradigme

cours 2 : programmation fonctionnelle

Jonathan Lejeune

Sorbonne Université/LIP6-INRIA

DATA CLOUD – Master 2 SAR 2020/2021



source : cours de Ph. Narbel, LaBRI

Objectifs

- Concevoir des programmes comme des fonctions mathématiques que l'on compose entre elles.
- Exploiter des fonctions avec des propriétés particulières augmentant la composition/décomposition de fonctions

Propriétés des fonctions

- La **pureté** : les résultats de la fonction ne dépendent strictement que de leurs arguments, sans autre effet externe
⇒ cloisonnement, localisation, stabilité, déterminisme
- La **1ère classe** : Les fonctions ont un statut de valeur
⇒ flexibilité d'utilisation, compositionnalité

Les langages purement fonctionnels

Haskell, Standard ML, Ocaml

Apports

- 1 Indépendance par rapport au contexte de l'application
- 2 Indépendance de l'ordre des applications des fonctions dans les expressions constitués de fonctions pures
ex : `f(g(3, h("ici",9.1), log(1), r('a', 'c'))), 10)`

Conséquences de 2

- Une fonction peut être une règle de réécriture (pas de notion d'état)
- Typage plus complet et plus représentatif des comportements
- Parallélisation possible
- Tests facilités

Difficultés

- Gestion des structures de données
- Gestion de la mémoire explicite (ramasses-miettes)

Conséquences

Pouvoir être :

- | | |
|---|--|
| 1 nommé, affecté (et typé) | <code>x := sin</code> |
| 2 défini et créé à la demande | <code>x := (function a -> a+1)</code> |
| 3 passé en argument d'une fonction | <code>f(sin)</code> |
| ⇒ Possibilité de généralisation/abstraction fonctionnelle | |
| 4 le résultat d'une fonction | <code>(f(3))(5)</code> |
| 5 stocké dans une structure de données | <code>array := {log, exp, tan}</code> |

Apports de 2 et 3

- Intégration facile de nouveau code dans le code existant.
- Flexibilité du code, maintenabilité, lisibilité

Exemple : généralisation d'un traitement itératif (map, reduce, filter, ...) :

```
map([1,2,3,4], (x -> x*2)) --> [2,4,6,8]
```

Fonctionnel vs. impératif

Critère	Impératif	Fonctionnel
Focus programmeur	Comment effectuer des tâches et comment assurer le suivi des modifications d'état	informations souhaitées et transformations requises
Modifications d'état	Importantes	Non-existantes
Ordre d'exécution	Important	Peu important
Contrôle de flux principal	Boucle, condition, appel de fonctions	Appel de fonctions + récursivité
Unité de manipulation principale	Instance de structures ou classes	Fonctions

Les fonctions en Scala

Déclaration

```
def functionName ([list of parameters]) : [return type] = {  
    function body  
    return [expr]  
}
```

Appel si déclaré dans un *object* (= méthode de classe)

```
<name_object>.functionName( list of parameters )  
//ou  
functionName( list of parameters ) // si dans le même bloc object
```

Appel si déclaré dans une *class* (= méthode de d'instance)

```
<instance>.functionName( list of parameters )
```

Paramètres par défaut pour une fonction

Déclaration

```
def addInt( a:Int = 5, b:Int = 7 ) : Int = {  
    var sum:Int = 0  
    sum = a + b  
    return sum  
}
```

Appels possibles

```
addInt() // a=5 et b=7  
addInt(b=2)//a=5 et b=2  
addInt(a=3)//a=3 et b=7  
addInt(1,0) // a=1, b=0
```

```
//On peut inverser les paramètres en les nommant  
addInt(b=1,a=2)// a=2, b=1
```

Fonction à nombre d'arguments variable

Déclaration

```
def printStrings( args:String* ) = {  
    var i : Int = 0;  
    for( arg <- args ){  
        println("Arg_value[" + i + "]_=" + arg );  
        i = i + 1;  
    }  
}
```

Appel

```
printStrings("Hello", "Scala", "Python");
```


Définition

Une fonction récursive est une fonction qui fait appel à elle même.

Exemple

```
def fact(n: Int): Int = {  
    if (n <= 1)  
        return 1  
    else  
        return n * fact(n - 1)  
}
```

ou alors

```
def fact(n: Int): Int = n match {  
    case 0 => 1  
    case x => x * fact(x-1)  
}
```

Définition

Fonction définie dans une autre fonction. Sa visibilité est locale à la fonction englobante.

Exemple

```
def factorial(i: Int): Int = {  
  def fact(i: Int, accumulator: Int): Int = {  
    if (i <= 1)  
      accumulator  
    else  
      fact(i - 1, i * accumulator)  
  }  
  fact(i, 1)  
}
```

Fonction call-by-name

Définition

Une fonction call-by-name est une fonction qui prend en paramètre une expression qui ne sera évaluée que lorsqu'elle est appelée à l'intérieur de la fonction.

Exemple

```
def plus( a: => Long , b: => Long ) = {  
    println("debut_␣plus")  
    a + b // a et b sont évaluées ici  
}
```

```
scala> def foo = { println("debut_␣foo") ; 5 }  
foo: Int
```

```
scala> plus(foo,foo)  
debut plus  
debut foo  
debut foo  
res3: Long = 10
```

Fonction d'ordre supérieur

Définition

Une fonction d'ordre supérieur est une fonction qui peut prendre une ou plusieurs fonctions en arguments ou/et qui peut renvoyer une fonction.

Exemples

```
scala> def twice(f: Int=>Int, i: Int) = f(f(i))  
twice: (f: Int => Int, i: Int)Int
```

```
scala> def myforeach[A](t: Traversable[A], f: (A) => Unit): Unit =  
  | for(x <- t) f(x)  
myforeach: [A](t: Traversable[A], f: A => Unit)Unit
```

Exemples d'appel

```
scala> def plusUn(a: Int): Int = a + 1  
plusUn: (a: Int)Int
```

```
scala> def fois2(a: Int): Int = a + a  
fois2: (a: Int)Int
```

```
scala> val x = twice(plusUn, 2)  
x: Int = 4
```

```
scala> val y = twice(plusUn, plusUn(10))  
y: Int = 13
```

```
scala> val z = twice(fois2, plusUn(1))  
z: Int = 8
```

```
scala> myforeach(List(3, 4, 5), println)  
3  
4  
5
```

Exemple de fonctions d'ordre supérieur de l'API scala

Dans `trait TraversableOnce[+A]` :

- `def exists(p: (A) => Boolean): Boolean`
renvoie vrai si le prédicat `p` est vérifié sur au moins un élément
- `def filter(p: (A) => Boolean): TraversableOnce[A]`
renvoie une nouvelle collection où tous les éléments respectent le prédicat `p`
- `def map[B](f: (A) => B): TraversableOnce[B]`
créé une nouvelle collection en appliquant à chaque élément de la collection appelante la fonction `f`
- `def reduce[A1 >: A](op: (A1, A1) => A1): A1`
Réduit les éléments avec l'opérateur associatif `op`.
- `def foreach(f: (A) => Unit): Unit`
Parcours de la collection en appliquant `f` à chaque élément.

Fonctions anonymes

Définition

Fonctions sans nom qui sont des littéraux de fonction et se déclare ainsi :

```
([identifiant: type, ... ]) =><expression>
```

Déclaration dans une variable

```
scala> val maximize = (a: Int, b: Int) => if (a > b) a else b  
maximize: (Int, Int) => Int = <function2>
```

```
scala> val m = maximize(84, 96)  
m: Int = 96
```

Passage de paramètre d'une fonction d'ordre supérieure

Exemple avec `def twice(f: Int => Int, i: Int) = f(f(i))`

```
scala> val x = twice((a: Int) => a + 1, 2)  
x: Int = 4
```

```
scala> val y = twice((a: Int) => a + a, plusUn(10))  
y: Int = 44
```

Sucres syntaxiques dans l'affectation d'une fonction anonyme

- Les typages des paramètres de la fonction anonyme n'est pas obligatoire
⇒ Inférence des types depuis le type attendu de la fonction

```
val x = twice(a=>a+1 ,2) // type inféré de a = Int
```

```
val y = twice(a=>a+a ,plusUn(10))
```

- Possible de désigner le n-ième argument au n-ième underscore ssi chaque argument de la fonction anonyme est utilisé 1 fois alors

```
val x = twice(_+1,2)//équivalent à twice(a=>a+1,2)
twice(a=>a+a ,plusUn(10)) // impossible d'utiliser l'underscore ici
scala> def truc( f: (Int,Int)=>Int ) : Int = 2 * f(2,4)
truc: (f: (Int, Int) => Int)Int
scala>val z = truc(_+_ )
z: Int = 12
```


Relation fonction anonyme et objet scala

Fonction anonyme = instance d'un objet implémentant un trait de l'API std.

⇒ Fonction anonyme = instance d'une classe anonyme

```
trait Function0[+R] extends AnyRef
trait Function1[-T1, +R] extends AnyRef
trait Function2[-T1, -T2, +R] extends AnyRef
...
//until trait Function22
```

```
val max = new Function2[Int, Int, Int] {
  def apply(x: Int, y: Int): Int = if (x < y) y else x
}
```

est équivalent à

```
val max = (x: Int, y: Int) => if (x < y) y else x
```

Compatibilité des variables typées par une fonction anonyme

Soit une variable f de type $X_f \Rightarrow Y_f$.

`val g:Xg =>Yg = f` est possible si

$X_f = X_g$ **ou** X_g est un sous type de X_f (contra-variance)

Et $Y_f = Y_g$ **ou** Y_f est un sous type de Y_g (covariance)

```
scala> val egal = (a:Any,b:Any) => a == b
egal: (Any, Any) => Boolean = <function2>
```

```
scala> val egalInt:(Int,Int) => Boolean = egal
egalInt: (Int, Int) => Boolean = <function2>
```

Application partielle d'une fonction

Définition

L'application partielle d'une fonction permet de définir une nouvelle fonction en passant en paramètre une partie de ses arguments.

⇒ arguments non spécifiés remplacés par des underscores

Exemple

```
scala> def log(date: Date, mess: String)= println(date + ":_" + mess)
log: (date: java.util.Date, mess: String)Unit
```

```
scala> val now = new Date
now: java.util.Date = Mon Oct 23 19:46:32 CEST 2017
```

```
scala> val log_with_predefined_date = log(now, _: String)
log_with_predefined_date: String => Unit = <function1>
```

```
scala> log_with_predefined_date("mess_1")
Mon Oct 23 19:46:32 CEST 2017: mess 1
```

```
scala> log_with_predefined_date("mess_2")
Mon Oct 23 19:46:32 CEST 2017 : mess 2
```

Curryfication

Transformation d'une fonction à plusieurs arguments en une fonction à un argument qui retourne une fonction sur le reste des arguments.

Exemples de curryfication

```
scala> def uncurried_add(x : Int, y : Int) = x + y
uncurried_add: (x: Int, y: Int)Int
```

```
// Versions curryfiées :
```

```
scala> def curried_add0(x : Int)(y : Int) = x + y
curried_add0: (x: Int)(y: Int)Int
```

```
scala> def curried_add1 (x : Int) = (y : Int) => x + y
curried_add1: (x: Int)Int => Int
```

```
scala> def curried_add2 = (x : Int) => (y : Int) => x + y
curried_add2: Int => (Int => Int)
```

Définition

Arguments passés implicitement lors de l'appel à la fonction

```
def f(a: Int)(implicit argimpl1: Int, argimpl2: String) = ....
```

Un argument de type T est passé implicitement à la fonction ssi :

- il existe **exactement une** variable ou un `object`
- déclaré `implicit`
- de type ou de sous-type de T
- visible depuis le code client.

```
def foo(hours: Int)(implicit amount: BigDecimal, curName: String) =  
  (amount * hours).toString() + "␣" + curName
```

```
implicit val hourlyRate = BigDecimal(34)  
implicit val currencyName = "Dollars"
```

```
scala> val x = foo(3)  
x: String = 102 Dollars
```