Programmation Fonctionelle

Jean-Luc Falcone

4 juin 2013

Définitions

- La programmation fonctionelle:
  - ~~Programmer avec des fonctions~~
- Langage fonctionel:
  - ~~Langage avec des fonctions~~

Transparence référentielle

Une expression est référentiellement transparente si on peut remplacer chacune de ses occurence avec le résultat de son évaluation sans changer le fonctionement d'un programme.

Exemple (C/java/...)

```
//Referentiellement transparente
double x = PI / 2;
double y = sqrt( sin(x)*sin(x) + cos(x)*cos(x) );

int i = 0;

//Referentiellement opaque
i = 3;
int j = ++i;
```

Transparence référentielle (exemples en Scala)

```
val now = currentTime()

val xs = Array( 0, 0, 0 )
xs(1) = 1

val xml = XML.fromFile( "hello.xml" )
val html = format( xml )
save( html, "hello.html" )
```

Fonction pures

Une fonction pure est une fonction référentiellement transparente.

Exemple (python)

```
#Fonction pure
def isEmpty( lst ):
    return len(lst) == 0

#Fonction impure
emptyNum=0
def countIfEmpty( lst ):
    if isEmpty(lst):
        emptyNum += 1
    return emptyNum
```

Fonctions Pures (exemple en scala)

```scala
def randomNoise( x: Double ) =
  x + rng.nextDouble()/100

def query( db: DataBase, sql: SQL ): Result =
  db.execute( sql )

def sum( is: Array[Int] ): Int = {
  var i = 0
  var sum = 0
  while( i < is.size ) {
    sum += is(i)
    i += 1
  }
  sum
}
```

Définitions

- Programmation fonctionnelle:

  Style de programmation *basé sur l'utilisation d'expression réf. transparentes et de fonctions pures.*

- Langage fonctionel:

  Langage contraingnant le style fonctionel.

Attention

Scala n'est pas un langage fonctionnel (selon cette définition) mais facilite l'utilisation du style fonctionnel.

Avantages

- Pas d'effets de bords
- Composabilité
- Toujours *thread-safe*
- L'ordre de l'évaluation des arguments n'a pas d'importance
- Possibilité d'utiliser un cache
- Facilite l'analyse du code

Désavantages

- Pas d'IO (effets de bord)
- **Peut** être plus lent (p.e. copie conservative)
- Nécéssite des structures de données appropriées
- Les algorithms sont souvent présentés de manière procédurale.
- Le hardware a un fonctionnement impératif.

- Implique un changement d'habitude (apprentissage)

Immutabilité

Utiliser des `val` à la place des `var` !

```
class PointM( var x: Double, y: Double ) {
  def moveHorizontaly( dx: Double ): Unit = {
    x = x + dx
  }
}

case class PointI( x: Double, y: Double ) {
  def moveHorizontaly( dx: Double ): PointI =
    copy( x = x+dx )
}
```

Boucles

Pas moyen d'avoir une boucle sans variable ou sans effet de bord !

```
def sum( is: Array[Int] ): Int = {
  var i = 0
  var sum = 0
  while( i < is.size ) {
    sum += is(i)
    i += 1
  }
  sum
}
```

Récursion (terminale)

```
def sum( is: Array[Int] ): Int = {

  def sumRec( i: Int, sum: Int ): Int =
    if( i == is.size ) sum
    else sumRec( i+1, sum+is(i) )

  sumRec( 0, 0 )
}
```

Procédural: Mettre à jour l'état

```
trait StackM[A] {

  def isEmpty: Boolean

  def push( a: A ): Unit

  def pop: A

}
```

Fonctionnel: Retourner le nouvel état

```
trait StackI[A] {

  def isEmpty: Boolean

  def push( a: A ): StackI[A]

  def pop: (A,StackI[A])

}
```

Exemples

```
def addTop( stack: StackM[Int] ): Unit = {
  val x = stack.pop
  val y = stack.pop
  stack.push( x + y )
}

def addTop( stack: StackI[Int] ): StackI[Int] = {
  val (x,stack1) = stack.pop
  val (y,stack2) = stack1.pop
  stack2.push( x + y )
}
```

Fonctions anonymes

```
(i: Int) => i+1                 // (Int)=>Int

(i: Int, j: Int ) => i+j // (Int,Int)=>Int

(_:Int) * 3                     // (Int)=>Int

(_:Int) / (_:Int)               // (Int,Int)=>Int
```

Utilisation

```
val inc = (i: Int) => i+1
val add = (i: Int, j: Int ) => i+j

val triple = (_:Int) * 3
val div = (_:Int) / (_:Int)

val f = inc.andThen(triple)
val g = (i:Int,j:Int) => add( f(i), div(i,j) )
```

Evaluation de fonction

```
f(3)          //=> 12
add(2,3)      //=> 5
div(12,2)     //=> 6
g(4,2)        //=> 17
```

Problème

- Pour calculer le montant total d'une commande
  - On multiplie le nombre d'unités par le coût
  - On applique un rabais
- Le rabais peut être:
  - Inexistant
  - Une somme fixe (pe: -25 CHF)
  - Un pourcentage (pe: -15 %)
  - Un nombre d'unités offertes à partir d'un certain volume (pe: la onzième offerte)

Solution Orientée-Objet

```
trait Rebater {
  def apply( d: Double ): Double
}
object NoRebate extends Rebater{
  def apply( d: Double ) = d
}
case class Percent( p: Double ) extends Rebater {
  val factor = 1 - p/100
  def apply( d: Double ) = d * factor
}
def total( units: Int, cost: Double, rebate: Rebate ) = {
  val total = units*cost
  rebate.apply( total )
}
total( 3, 105.00, Percent(15) )
```

Solution Fonctionelle

```
def total( uts: Int, c: Double, rebate: Double=>Double ) =
  val total = units*cost
  rebate( total )
}

total( 3, 105.00, (d:Double) => d-25 )
total( 3, 105.00, (_:Double) * (1-0.15) )

total( 3, 105.00, d => d-25 )
total( 3, 105.00, _ * 0.85 )

total( 3, 105.00, identity  )
```

Solution Fonctionelle: Multi param lists

```
def total( uts: Int, c: Double)( reb: Double=>Double ) = {
  val total = units*cost
  rebate( total )
}

val customer: Customer = ...

total( 3, 105.00 ){ t =>
  val amount = customer.pastTotalOrders
  if( amount > 5000 ) t * 0.9
  else t
}
```

Solution Fonctionelle: exemples (2)

```
val noRebate = (d:Double) => d
total( 3, 105.00, noRebate )

def percent( p: Percent ): Double=>Double = {
  val factor = 1 - p/100
  d => d*factor
}
total( 3, 105.00, percent(15) )
```

Solution Fonctionelle: exemples (3)

```
def whenHigh( treshold: Double,
              rebate: Double=>Double): Double=>Double =
  total => if( total >= threshold ) rebate(total)
           else total

val vipOnly = whenHigh( 500, percent(10) )

total( 3, 105.00, vipOnly )
```

Base de donnée (style impératif)

```
object DataBase {
  def connect(...): DataBase
}
trait DataBase {
  def transaction: Transaction
  def disconnect: Unit
}
trait Transaction {
  def query( sql: SQL ): Result
  def update( sql: SQL ): Boolean
  def commit: Unit
}
```

Utilisation (style impératif)

```
val db = DataBase.connect( ... )

val tx1 = db.transaction
val i = tx1.query(...) //READ
tx1.update( ... )      //WRITE
tx1.commit

val tx2 = db.transaction
val j = tx2.query(...)  //READ
tx2.update( ... )       //WRITE
tx2.commit

db.disconnect
```

Base de donnée (style hybride)

```
object DataBase {
  def connect(...): DataBase
  def withConnection(...)( body: DataBase=>Unit) {
    val db = connect(...)
    body(db)
    db.close
  }
}
trait DataBase {
  def transaction: Transaction
  def disconnect: Unit
  def withTransaction( ops: Transaction=>Unit ) {
    val tx = transaction
    ops(tx)
    tx.commit
  }
}
```

Utilisation (style hybride)

```
DataBase.withConnection( ... ){ db =>

  db.withTransaction { tx =>
    val i = tx.query(...) //READ
    tx.update( ... )      //WRITE
  }

  db.withTransaction { tx =>
    val i = tx.query(...) //READ
    tx.update( ... )      //WRITE
  }

}
```