Tutoriel Type Driven Development

Emission du 23/01/2019

Indroduction à la programmation fonctionnelle

La programmation fonctionnelle utilise des langages de programmation tels que :

Lisp ;

Ocaml ;

Haskell

Inspiré du lambda calcul, le principe général de la programmation fonctionnelle est de concevoir des programmes comme des fonctions mathématiques que l'on compose entre elles.

L'un des principes fondamentaux de la programmation fonctionnelle consiste à écrire dans nos applications pour que le noyau soit constitué de fonctions pures plus faciles à raisonner, à combiner, à tester, à déboguer et à paralléliser.

Tandis que les effets secondaires se situe dans une couche externe mince. Les fonctions pures et les programmes fonctionnels sont bâtis sur des expressions dont la valeur est le résultat du programme.

Fonction pure

Une fonction pure est une fonction mathématique simple à raisonner, répétable et composable, où sa sortie ne dépend que de son entrée. Sa valeur de retour est la même pour les mêmes arguments à chaque appel. Son évaluation n'a pas d'effets de bord c'est à dire qu'elle n'utilise que ces paramètres et qu'elle ne change pas les variables globales dans le programme ni les arguments mutables de type référence ou de flux d'entrée-sortie.

Les fonctions pures contribuent également à l'optimisation des performances grâce à leurs caractéristiques de transparence référentielle et de mémorisation.

Exemple :

```
function salutation(prenom) {
  return "Bonjour " + prenom;}
```

En utilisant «.length() » dans une fonction elle reste pure alors que si l'on utilise « .random() » la fonction devient impur.

Fonction impure

Contrairement aux fonctions pures les fonctions impures peuvent utiliser une ou plusieurs variables qui se trouvent hors du contexte de c'est fonction. On aurait donc pu changer leurs valeurs au risque de créer un comportement imprévu dans la suite du programme. Cette manipulation des données crée des effets de bord.

```
Function salutation (){
  Msg = "bonjour" + prenom ;}
```

En entrant plus dans les détails, on peut définir les effets de bord comme l'utilisation (en lecture ou écriture), par une fonction, de toute variable qui est en dehors de son contexte local. Ce contexte se limite

Mais aussi de :

Modifier une structure de données en place ;

Définir un champ sur un objet

Lancer une exception ou arrêter avec une erreur ;

Imprimer sur la console ou lire les entrées de l'utilisateur ;

Lire ou écrire dans un fichier.

1) Néanmoins on peut se poser la question :

Comment allons-nous réussir à implémenter un programme si on s'interdit tout effet de bord ?

La réponse est que la programmation fonctionnelle est une restriction sur la façon dans nous écrivons un programme mais pas sur les programmes que nous pouvons exprimer.

La transparence référentielle

On dit qu'une fonction est préférentiellement transparente si elle peut être remplacer par sa valeur sans changer le programme. On utilise la transparence référentielle en programmation fonctionnelle sur des fonctions pures car leurs valeurs ne changent pas. On remplace chaque fonction par son résultat final dans la suite du programme comme pour la résolution d'une équation algébrique.

Exemple :

```
int plusOne (int x)
{return x+1;}
```

Ou simplement si on a :

scala > val x = 5 + 2

Ici on impose x dans notre programme qui peut être remplacer par 7 sans changer le déroulement du programme ni son résultat final.

Type Driven Development

Définition

Le type Driven Development (TDD) est la conséquence de certaines libertés que s'autorise un développeur Notamment celle d'écrire en fonctionnel pur dans des langages fortement typés tel qu'en :

JAVA ;

C++ ;

SCALA...

Il suffit de comprendre exactement :

Ce que fait la fonction

Ce qu'elle prend comme argument ;

La valeur de retour si elle existe autrement dit connaitre la signature de la fonction afin de produire un code qui respect la notion de pureté et la transparence référentielle.

Exemples du TDD

En java

Habituellement on définit des méthodes tel que :

```
class Player(var score: Int = 0) {¶
  def incrementScore() = score += 1}¶
```

On voit que la variable « score » dans cet exemple va être modifié en faisant appel à la fonction incrementeScore risquant de créer des bugs par la suite sans oublier que la class 'player' hérite probablement d'une autre class ce qui crée du code très peu compréhensible et dur à analyser.

Une des manières de coder proprement cette fonction est de l'écrire en fonctionnel :

```
val incrementMe = (initialScore: Int) => initialScore + 1¶
```

IncrementMe est une fonction qui prend un entier en argument (initialScore) et retourne une fonction, prenant elle-même un entier en paramètre (increment) et sommant le tout.

Ce qui nous permet d'éviter de créer des liens entre la méthode et la fonction et surtout on voit bien que la fonction ne change pas les paramètres de la méthode contrairement à la programmation impérative ou orienté l'objet et on peut la rappeler autant qu'on veut dans le programme en utilisant simplement « incremetMe() ».

En scala

Additionner et multiplier des variables en scala en utilisant la pureté des fonctions :

```
val h = (f: Int => Int, g: Int => Int) => g(f(x))
```

```
scala> def add(a:Int,b:Int) = a + b
add: (a: Int, b: Int)Int
scala> def multiply(a:Int,b:Int) = a * b
multiply: (a: Int, b: Int)Int
scala> add(5,8) + multiply(5,8)
res0: Int = 53
scala> multiply(5,8) + add(5,8)
res1: Int = 53
```

1) Composition de deux fonction g(f(x)) :

   Autre méthode pour écrire la composition de fonctions :

```
s cala> def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
sumOfSquares: (Double,Double)Double
```

```
def compose[A, B, C](f: A => B, g: B => C): (A => C) = x => g(f(x))
```

   Calculer (X∗ X) + (Y∗ Y)

```
private def formatAbs(x: int) = {
def abs (n: int)°: int = 
if (n < 0) -n
else n
val msg = "la valeur absolue de %d est %d"
msg.format(x, abs (x))
```

   Calculer la valeur absolue d'un nombre n avec la fonction « abs » et l'afficher en utilisant la méthode 'format' définit dans la librairie standard 'ONSTRING'

```
Def factoriel (n°: int){
      Def go(n°: int, acc°: int)°: int =
      if (n <= 0) acc
      else go(n-1, n*acc)
      go(n, 1)
}
```

   Calculer la factoriel d'un nombre n:

   Afficher le résultat d'une fonction en utilisant une fonction d'affichage :

```
Private def formatFactoriel (n: int){
   Val msg = "factoriel de %d est %d"
   Msg.format(n, factoriel(n))
}
```

   Utilisation des listes en scala pur

```
def sum(ints: List[Int]): Int = ints match {
case Nil => 0
case Cons(x,xs) => x + sum(xs)
}
def product(ds: List[Double]): Double = ds match {
case Nil => 1.0
case Cons(0.0, _) => 0.0
case Cons(x,xs) => x * product(xs)
}
```

Les exceptions en scala pur :

```
def failingFn(i: Int): Int = {
   try {
       val x = 42 + 5
       x + ((throw new Exception("fail!")): Int)
       }
   catch {
       case e: Exception => 43
       }
```

2) Ou :

```
def mean(xs: Seq[Double]): Double ={
if (xs.isEmpty)
       throw new ArithmeticException("mean of empty list!")
else
       xs.sum / xs.length
```

Ou:

```
def Try[A](a: => A): Option[A] ={
try Some(a)
catch { case e: Exception => None
       }
Finally{
       //do instruction
       }
```

Conclusion

Ce tutoriel m'a permis d'approfondir mes connaissances sur le langage fonctionnel que j'avais acquise lors de mon cursus. J'ai pu expliciter sa définition et sa fonction notamment

pour les fonctions pures, et la transparence référentielle. Par ailleurs j'ai pu intégrer certaines notions de base en Scala et d'assimiler le Type Driven Development.

Ainsi pour la suite j'utiliserai mon tutoriel afin de produire un code fonctionnel palliant les erreurs de bases tout en fournissant un code ergonomique.

Vous trouverez le code de mon projet sur mon github en cliquant ici