PROGRAMMATION FONCTIONNELLE

Fabrice Legond-Aubry et Pascal Poizat

Juillet-Août 2024

- Introduction
- Pureté
- Immutabilité
- Ordre supérieur
- Programmes séquentiels
- Absence de valeur et erreurs
- Types de données algébriques
- Effets de bord
- Flux et évaluation paresseuse
- Concurrence
- Test
- Synthèse des notations

PROGRAMMES SÉQUENTIELS

PIPELINING

très utilisé, ici sur programmes séquentiels

Scala
```scala
case class Book(title: String, authors: List[String])
val books = List(
  Book("FP in Scala", List("Chiusano", "Bjarnason")),
  Book("The Hobbit", List("Tolkien")),
  Book("Modern Java in Action", List("Urma", "Fusco", "Mycroft"))
)
```

nombre de livres sur Scala ?

[●,●,●] : *List*[*Book*] – *map* → [▫,▫,▫] : *List*[*String*] – *filter* → [▫] : *List*[*String*] – *size* → 1

Scala
```scala
def numberHaveName(pattern: String)(books: List[Book]): Int = {
  books.map(_.title).filter(_.contains(pattern)).size
}
println(numberHaveName("Scala")(books))
```

RECOMMANDATIONS V1

```scala
case class Movie(title: String)
def bookAdaptations(author: String): List[Movie] = {
  if (author == "Tolkien")
    List(Movie("An Unexpected Journey"), Movie("The Desolation of Smaug"))
  else
    List.empty
}
```

recommandations à partir d'une liste de livres ?

```scala
def recommendationFeed(books: List[Book]): List[Movie] = {
  ???
}
```

EN JAVA

Java

```java
static List<String> recommendationFeed(List<Book> books) {
    List<String> result = new ArrayList<>();
    for (Book book: books)
        for (String author: book.authors())
            for (Movie movie: bookAdaptations(author))
                result.add(String.format("You may like %s, because you liked %s's
    return result;
}
System.out.println(recommendationFeed(books));
```

- collection mutable
- lisibilité des boucles imbriquées
- instruction *vs* expression dans les boucles

VERS UNE SOLUTION SCALA

utilisons le pipelining

1. on part de la liste des livres
2. `map(_.authors)` pour obtenir les auteurs
3. `map(bookAdaptations)` pour obtenir les adaptations
4. et voilà ! (ou pas ...)

Scala
```
scala> books.map(_.authors)
val res0: List[List[String]] = List(List(Chiusano, Bjarnason),
                                    List(Tolkien),
                                    List(Urma, Fusco, Mycroft))
```

`List[String] != List[List[String]]`

VERS UNE SOLUTION SCALA

idée

- utiliser `flatten`

  `List[A].flatten[B](implicit toIterableOnce: List[A] => IterableOnce[B]): List[B]`

  List[*List*[*B*]] ↪ () → *List*[*B*]

Scala

```
scala> books.map(_.authors).flatten
val res2: List[String] = List(Chiusano, Bjarnason, Tolkien, Urma, Fusco, Mycroft)
```

ça reste verbeux …

VERS UNE SOLUTION SCALA

- utiliser `flatMap`

  `List[A].flatMap[B](f: A => IterableOnce[B]): List[B]`

  List[*A*] ↪ (*A*→*List*[*B*]) → *List*[*B*]

  - `map` + `flatten` = `flatMap`

Scala

```
scala> books.flatMap(_.authors)
val res3: List[String] = List(Chiusano, Bjarnason, Tolkien, Urma, Fusco, Mycroft)
```

principe : `flatMap` est l'une des fonctions les plus importantes en PF

SOLUTION SCALA

Scala

```scala
def recommendationFeed(books: List[Book]): List[Movie] = {
  books.flatMap(_.authors).flatMap(bookAdaptations)
}
```

à comparer à

Java

```java
static List<String> recommendationFeed(List<Book> books) {
    List<String> result = new ArrayList<>();
    for (Book book: books)
        for (String author: book.authors())
            for (Movie movie: bookAdaptations(author))
                result.add(String.format("You may like %s, because you liked %s's
    return result;
}
```

(on verra un `flatMap` en Java plus tard)

SYNTHÈSE INTERMÉDIAIRE

- `map`: *List*[*A*] ↪ (*A*→*B*) → *List*[*B*]

  ```
  List[A].map(f: A => B): List[B]
  ```

- `flatten`: *List*[*List*[*B*]] ↪ () → *List*[*B*]

  ```
  List[A].flatten[B](implicit toIterableOnce:
  List[A] => IterableOnce[B]): List[B]
  ```

- `flatMap`: *List*[*A*] ↪ (*A*→*List*[*B*]) → *List*[*B*]

  ```
  List[A].flatMap[B](f: A => IterableOnce[B]):
  List[B]
  ```

(ici on cache volontairement le rôle de `IterableOnce`)

EXERCICE

quel est le résultat de ces appels ?

```scala
List(1, 2, 3).flatMap(i => List(i, i + 10))
List(1, 2, 3).flatMap(i => List(i * 2))
List(1, 2, 3).flatMap(i => if (i % 2 == 0) List(i) else List.empty)
```

EXERCICE

Complétez pour obtenir les auteurs recommandés par des amis

```Scala
case class Book(title: String, authors: List[String])
def recommendedBy(friend: String): List[Book] = { ... }
val friends = List("Alice", "Bob", "Charlie")
val recommendedBooks: List[Book] = ???
val recommendedAuthors: List[String] = ???
```

RECOMMANDATIONS V2

En Java nous avions

Java

```java
static List<String> recommendationFeed(List<Book> books) {
    List<String> result = new ArrayList<>();
    for (Book book: books)
        for (String author: book.authors())
            for (Movie movie: bookAdaptations(author))
                result.add(String.format("You may like %s, because you liked %s's
    return result;
}
```

mais en Scala seulement

Scala

```scala
def recommendationFeed(books: List[Book]): List[Movie] = {
  books.flatMap(_.authors).flatMap(bookAdaptations)
}
```

RECOMMANDATION V2

Nous pouvons tenter

```Scala
recommendationFeed(books).map(movie => s"You may like ${movie.title}, " +
                                       s"because you liked $author's ${book.title
```

problème ?

RECOMMANDATION V2

Il faut maintenir les scopes externes
et imbriquer les `flatMap`

Scala
```scala
def recommendationFeed2(books: List[Book]): List[String] = {
  books.flatMap(book =>
    book.authors.flatMap(author =>
      bookAdaptations(author).map(movie =>
        s"You may like ${movie.title}, " +
        s"because you liked $author's ${book.title}"
        )))
}
```

à comparer à

Scala
```scala
def recommendationFeed(books: List[Book]): List[Movie] = {
  books.flatMap(_.authors).flatMap(bookAdaptations)
}
```

EXERCICE

Complétez pour obtenir

`List(Point(1,-2), Point(1, 7))`

```scala
case class Point(x: Int, y: Int)
List( ??? ).flatMap(x =>
  List( ??? ).map(y =>
    Point(x, y)))
```

Combien d'éléments dans la liste résultat avec

- `List(1,2)` et `List(-2)` ?
- `List(1,2)` et `List(-2,7,10)` ?

FOR COMPREHENSIONS

Scala à une notation plus élégante

```scala
for {
  x <- xs
  y <- ys
} yield f(x, y)
```

au lieu de

```scala
xs.flatMap(x =>
  ys.map(y =>
    f(x, y)
    ))
```

EXPRESSIONS VS INSTRUCTIONS

- on retrouve des **list** comprehensions, avec filtrage, dans de nombreux langages
- en Haskell et Scala
  on peut utiliser les **for** comprehension sur les listes et sur d'autres types que nous verrons dans la suite
- ne pas confondre **expression** (retour) et **instruction** (pas de retour)
  - en Scala, `for` est une **expression**
  - le for de Java (par exemple `for(Point p: points){...}`) est une **instruction**
  - on retrouve cela en Java avec `switch` expression et `switch` instruction

RECOMMANDATION V2 (SIMPLIFIÉ)

Scala
```scala
def recommendationFeed2(books: List[Book]): List[String] = {
  books.flatMap(book =>
    book.authors.flatMap(author =>
      bookAdaptations(author).map(movie =>
        s"You may like ${movie.title}, " +
        s"because you liked $author's ${book.title}"
        )))
}
```

avec for comprehension ?

Scala
```scala
def recommendationFeed3(books: List[Book]): List[String] = {
  for {
    book <- books
    author <- book.authors
    movie <- bookAdaptations(author)
  } yield s"You may like ${movie.title}, " +
          s"because you liked $author's ${book.title}"
}
```

EXERCICE

Donnez une version sans
et une version avec for comprehension

```Scala
case class Point3D(x: Int, y: Int, z: Int)
???
```

qui donnent

```Scala
List(
  Point3D(-1, -1, -1), Point3D(-1, -1, 0), Point3D(-1, -1, 1),
  Point3D(-1, 0, -1), Point3D(-1, 0, 0), Point3D(-1, 0, 1),
  Point3D(-1, 1, -1), Point3D(-1, 1, 0), Point3D(-1, 1, 1),
  Point3D(1, -1, -1), Point3D(1, -1, 0), Point3D(1, -1, 1),
  Point3D(1, 0, -1), Point3D(1, 0, 0), Point3D(1, 0, 1),
  Point3D(1, 1, -1), Point3D(1, 1, 0), Point3D(1, 1, 1),
)
```

CERCLES V1

```scala
case class Point(x: Int, y: Int)
def isInside(point: Point, radius: Int): Boolean = {
    radius * radius >= point.x * point.x + point.y * point.y
}
val points = List(Point(5,2), Point(1,1))
val radiuses = List(2, 1)
```

quels points sont dans le cercle
de centre `(0,0)` et rayon `r`?

idées

- `checkInside(points: List[Point], radiuses: List[Int]): List[Point]`
- for comprehension + filtrage nécessaire
- (amélioration) curryfier `isInside`

SOLUTION 1 CERCLES V1

- utilisation de **garde**
- approche moins générale que les 2 suivantes

Scala

```scala
case class Point(x: Int, y: Int)
def isInside(point: Point, radius: Int): Boolean = {
    radius * radius >= point.x * point.x + point.y * point.y
}
def checkInside1(points: List[Point], radiuses: List[Int]): List[Point] = {
    for {
        radius <- radiuses
        point <- points
        if isInside(point, radius)
    } yield point
}
```

SOLUTION 2 CERCLES V1

- utilisation de `flatMap` implicite
- approche un peu overkill ici

Scala

```scala
case class Point(x: Int, y: Int)
def isInside(point: Point, radius: Int): Boolean = {
    radius * radius >= point.x * point.x + point.y * point.y
}
def insideFilter(point: Point, radius: Int): List[Point] = {
    if isInside(point, radius) then List(point) else List.empty
}
def checkInside2(points: List[Point], radiuses: List[Int]): List[Point] = {
    for {
        radius <- radiuses
        point <- points
        inPoint <- insideFilter(point, radius)
    } yield point
}
```

SOLUTION 3 CERCLES V1

- utilisation de `filter`
- on a currifié `isInside` au passage

Scala

```scala
case class Point(x: Int, y: Int)
def isInside(radius: Int)(point: Point): Boolean = {
    radius * radius >= point.x * point.x + point.y * point.y
}
def checkInside3(points: List[Point], radiuses: List[Int]): List[Point] = {
    for {
        radius <- radiuses
        point <- points.filter(isInside(radius))
    } yield point
}
```

EXERCICE

- on souhaite éviter les données invalides

```scala
val points = List(Point(5, 2), Point(1, 1))
val radiuses = List(-10, 0, 2)
```

- que donne le code actuel avec ces données ?
- corrigez cela
  en utilisant les trois techniques de filtrage
  - garde
  - `flatMap` implicite
  - `filter`

QUELQUES PRINCIPES

principe : on construit des programmes
à partir de petites fonctions

principe : on se base sur des abstractions réutilisables
(y compris multi-langage)

par exemple le fait de supporter `map` :

Scala

```scala
def mappables(): Unit = {
    case class Etudiant(nom: String, prenom: String)
    val bob = Etudiant("Sponge", "Bob")
    val pat = Etudiant("Star", "Patrick")
    val etudiantsL = List(bob, pat)
    val etudiantsS = Set(bob, pat)
    val etudiantsT = Node(Leaf(bob), Leaf(pat))
    println(etudiantsL.map(_.prenom)) // List(Bob, Patrick)
    println(etudiantsS.map(_.prenom)) // Set(Bob, Patrick)
    println(etudiantsT.map(_.prenom)) // Node(Leaf(Bob), Leaf(Patrick))
```

COMPREHENSION MULTI-TYPES

abstraction `flatMap` utilisable pour différents types

```scala
for {
    a <- List(1,2)
    b <- List(2,1)
} yield a * b
List(2, 1, 4, 2)
```

```scala
for {
    a <- Set(1,2)
    b <- Set(2,1)
} yield a * b
Set(2, 1, 4)
```

mais aussi multi-types pour énumérateurs (le premier définit le type final)

```scala
for {
    a <- List(1,2)
    b <- Set(2,1)
} yield a * b
List(2, 1, 4, 2)
```

```scala
for {
    a <- Set(1,2)
    b <- List(2,1)
} yield a * b
Set(2, 1, 4)
```

EXERCICE

complétez

Scala
```scala
for {
    x <- List(1, 2, 3)
    y <- Set(1)
} yield x * y
???( ??? )
```

Scala
```scala
for {
    x <- ???
    y <- List(1)
} yield x * y
Set(1, 2, 3)
```

Scala
```scala
for {
    x <- ???(1, 2, 3)
    y <- Set(1)
    z <- Set( ??? )
} yield x * y * z
List(0, 0, 0)
```

SCALA VS JAVA : FLATMAP

Scala

```scala
def recommendationFeed(books: List[Book]): List[Movie] = {
  books.flatMap(_.authors).flatMap(bookAdaptations)
}
```

Java

```java
public static List<Movie> recommendationFeed(List<Book> books) {
  return books.stream()
    .flatMap(b -> b.authors().stream())
    .flatMap(a -> bookAdaptations(a).stream())
    .toList();
}
```

noter :

- la verbosité
- la présence de `.stream()` au niveau de `flatMap`

SCALA VS JAVA : FLATMAP

Scala

```scala
def recommendationFeed2(books: List[Book]): List[String] = {
  books.flatMap(book =>
    book.authors.flatMap(author =>
      bookAdaptations(author).map(movie =>
        s"You may like ${movie.title}, " +
        s"because you liked $author's ${book.title}"
        )))
}
```

Java

```java
public static List<String> recommendationFeed2(List<Book> books) {
  return books.stream().flatMap(book ->
    book.authors().stream().flatMap(author ->
      bookAdaptations(author).stream().map(movie ->
        String.format("You may like %s, because you liked %s's %s",
                      movie.title(), author, book.title())
      ))).toList();
}
```

SCALA VS JAVA : FLATMAP

Scala

```scala
def recommendationFeed3(books: List[Book]): List[String] = {
  for {
    book <- books
    author <- book.authors
    movie <- bookAdaptations(author)
  } yield s"You may like ${movie.title}, " +
        s"because you liked $author's ${book.title}"
}
```

Java

```java
public static List<String> recommendationFeed2(List<Book> books) {
  return books.stream().flatMap(book ->
    book.authors().stream().flatMap(author ->
      bookAdaptations(author).stream().map(movie ->
        String.format("You may like %s, because you liked %s's %s",
                      movie.title(), author, book.title())
    ))).toList();
}
```

RÉSUMÉ

- pipelining
- expressions vs instructions
- Scala
  - `flatten`, `flatMap`
  - `flatMap` imbriqués / comprehensions
  - notations `???` et `for {} yield`
- Java
  - `flatMap`