Programmation fonctionnelle: notes de cours 3

S5 – 2023/2024

1 Fonctions à plusieurs arguments

1.1 Curryfication

Les lambda-abstractions permettent de définir des fonctions à un argument. On peut toutefois se demander comment définir les fonctions à plusieurs arguments, comme par exemple l'addition. Pour définir une fonction n-aire, le lambda-calcul offre deux possibilités :

- écrire une fonction *décurryfiée* : une lambda-abstraction qui prend pour seul argument un n-uplet et retourne le résultat attendu,
- écrire une fonction *curryfiée* : une lambda-abstraction qui prend le premier argument de la fonction et retourne une fonction sur les $n-1$ autres arguments.

Imaginons que nous souhaitions définir un lambda-terme représentant une fonction qui à trois arguments x, y et z associe la valeur $(x + y) \times z$. Si nous choisissons d'écrire une fonction décurryfiée, nous aurons

$$\lambda(t : (\mathbb{N} \times \mathbb{N} \times \mathbb{N}))\,.\,((\mathit{first}\,t) + (\mathit{second}\,t)) \times (\mathit{third}\,t)$$

où t est un triplet, et *first*, *second* et *third* sont des fonctions qui retournent respectivement le premier, deuxième et troisième élément d'un triplet.

Si au contraire nous choisissons d'écrire une fonction curryfiée, celle-ci sera :

$$\lambda(x : \mathbb{N}).\lambda(y : \mathbb{N}).\lambda(z : \mathbb{N})\,.\,(x + y) \times z$$

Notez que le type des deux fonctions diffère. La fonction décurryfiée est de type $(\mathbb{N} \times \mathbb{N} \times \mathbb{N}) \to \mathbb{N}$, où $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ dénote le type d'un triplet de nombres entiers. La fonction curryfiée est quant à elle de type $\mathbb{N} \to \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, ou en affichant explicitement les parenthèses, $\mathbb{N} \to (\mathbb{N} \to (\mathbb{N} \to \mathbb{N}))$.

La curryfication, qui tire son nom de celui du mathématicien Haskell Curry, est l'opération qui permet de transformer une fonction, par exemple pour passer de la première définition de l'addition à la seconde.

1.2 Application partielle

Le principal avantage d'une fonction curryfiée est qu'elle permet l'*application partielle*. Si f est une fonction n-aire curryfiée, et qu'on l'applique à un nombre

$m < n$ d'arguments, alors on obtient une autre fonction, qui prend $n - m$ arguments. Par exemple, si on définit la fonction binaire d'addition

$$add \equiv \lambda(n : \mathbb{N}).\lambda(m : \mathbb{N}).\, n + m$$

on peut utiliser l'application partielle pour définir une autre fonction

$$succ \equiv add\ 1$$

En développant la définition de *add* et en appliquant la β-réduction, on constate que *succ* est équivalent à $\lambda m.\, 1 + m$, c.-à-d. une fonction unaire qui à tout entier associe son successeur.

1.3 Curryfication en Scala

Scala permet de définir simplement une fonction décurryfiée ou curryfiée, suivant la syntaxe utilisée pour déclarer les arguments :

```
def additionUncurry(x: Int, y: Int): Int = x + y

def additionCurry(x: Int)(y, Int): Int = x + y
```

La syntaxe pour appliquer ces fonctions diffère elle aussi

```
val u = additionUncurry(3,5)

val v = additionCurry(3)(5)
```

On peut appliquer partiellement une fonction curryfiée

```
val f = additionCurry(1)
```

L'application partielle n'est pas directement possible avec la fonction décurryfiée. En revanche il est possible de curryfier une fonction automatiquement :

```
val g = additionUncurry.curried

val w = g(2)
```

2 Fonctions d'ordre supérieur

Une fonction est dîte d'*ordre supérieur* si elle prend une fonction comme argument, ou si elle retourne une fonction. Une fonction curryfiée est donc un exemple de fonction d'ordre supérieur. Puisque dans le lambda-calcul, toute expression est une fonction (une constante n'est rien d'autre qu'une fonction sans argument), on pourrait même dire que toutes les fonctions sont d'ordre supérieur ! Pour être plus précis, nous dirons qu'une fonction est d'ordre supérieur si elle prend un argument ou si elle retourne un élément de type $T_1 \to T_2$, où T_1 et T_2 sont des types.

L'exemple le plus simple de fonction d'ordre supérieur est sans doute l'opérateur de composition de fonctions :

$$\mathit{compose} \equiv \lambda(g : T_2 \to T_3).\lambda(f : T_1 \to T_2).\lambda(x : T_1).\, g\,(f\,x)$$

qui prend comme arguments deux fonctions, et en retourne une nouvelle de type $T_1 \to T_3$. Nous verrons par la suite d'autres exemples de fonctions d'ordre supérieur très utiles, comme la fonction *map* qui permet d'appliquer une fonction aux valeurs contenues dans une structure de données.

L'application partielle et les fonctions d'ordre supérieur offrent des moyens de combiner et réutiliser du code qui n'ont pas d'équivalent dans les autres paradigmes de programmation.

3 Déclarer et manipuler des données

Jusqu'ici nous avons défini des fonctions portant sur les types $\mathbb{N}$ et $\mathbb{B}$ (ou leurs équivalents Scala) ainsi que des types de fonctions. Il est aussi possible de définir nos propres types, en particulier des types de données algébriques.

3.1 Types de données algébriques dans Scala 3

L'utilisation la plus basique pour les types de données algébriques est de créer une énumération. La déclaration suivante crée un type avec 3 valeurs possibles :

```
enum TrafficLightColor:
  case Green
  case Yellow
  case Red
```

Il est aussi possible d'ajouter des arguments à chaque cas. Ici la définition n'inclut qu'un seul cas, mais avec deux arguments, pour former le type des paires d'entiers :

```
enum IntPair:
  case P(u: Int, v: Int)
```

Ainsi chaque cas peut être vu comme une fonction qui crée une valeur du type défini par la déclaration `enum`. Cette fonction (dans cet exemple, `P`) est appelé un *constructeur* du type de données. Les arguments d'un constructeur peuvent être du type défini par la déclaration `enum`, on obtient alors une définition récursive. La définition suivante introduit un type des entiers naturels en représentation unaire : un entier naturel est soit zéro, soit le successeur d'un autre entier naturel.

```
enum Nat:
  case Zero
  case Succ(n: Nat)
```

3.2 Les listes en Scala

Une des structures de données les plus utilisées en programmation fonctionnelle est la liste chaînée. Il est possible d'utiliser un type de données algébrique pour définir récursivement une liste :

```
enum IntList:
  case EmptyIntList
  case ConsIntList(x: Int, l: IntList)
```

Cette déclaration indique qu'une liste est soit la liste vide `EmptyIntList`, soit une liste constituée en chaînant un élément `x` à une autre liste `l` via le constructeur `ConsIntList`.

Scala propose un type prédéfini pour les listes chaînées, qui a l'avantage d'être polymorphe : le type `List[A]` dénote le type des listes contenant des éléments d'un type `A`. On a donc concrètement une infinité de types de listes, tels que `List[Int]`, `List[String]`, ou même `List[Int => Boolean]`. Hormis l'aspect polymorphe, les listes prédéfinies fonctionnent de la même manière que notre type `IntList`. Le constructeur pour la liste vide est `Nil`, et le constructeur pour chaîner un élément à une liste est `::` (utilisé comme un opérateur infixe). On peut ainsi déclarer une liste

```
val l1 = "Athos" :: "Porthos" :: "Aramis" :: Nil
```

dont le type est `List[String]`, et qui contient 3 éléments. On peut réutiliser cette liste pour en construire une autre, en y chaînant un élément supplémentaire.

```
val l2 = "D'Artagnan" :: l1
```

Il est possible d'utiliser la notation `List("Athos", "Porthos", "Aramis")` pour déclarer `l1`, ou `List()` pour la liste vide : cette syntaxe produit exactement le même résultat que l'utilisation explicite des constructeurs `Nil` et `::`.

Il est important de noter que `List` (de même que tous les types de données algébriques) est un type de données *immutable* : une fois la liste créée, il est impossible d'en modifier un élément. Cela signifie que pour changer un élément dans une liste, ou ajouter un élément à la fin de la liste (ce qui reviendrai à remplacer `Nil` par une autre liste), il faut créer une toute nouvelle liste. D'un autre côté, l'immutabilité permet de partager des données en mémoire : par exemple `l1` et `l2` "partagent" trois maillons, sans qu'une modification de `l1` ne puisse affecter `l2`. Les types de données immutables sont largement utilisés dans le cadre de la programmation fonctionnelle pure, puisqu'une fonction pure ne peut pas changer l'état d'un système, mais seulement associer des entrées et des sorties. Scala n'est pas un langage fonctionnel pur, et il propose aussi des types de données mutables, tels que les tableaux, que nous n'aborderons pas dans ce cours.

3.3 Le filtrage par motif

Le filtrage par motif est une construction permettant de spécifier différents cas pour l'évaluation d'une fonction, suivant la valeur prise par une expression.

On peut voir cette construction comme une généralisation de la fonction conditionnelle *if _ then _ else _* qui permet de tester une expression de n'importe quel type (alors que la conditionnelle teste uniquement une expression booléenne) et de définir un nombre arbitraire de cas. Dans la fonction suivante, le filtrage est effectué sur la valeur de l'expression `c`, et trois cas possibles sont décrits, avec pour chacun une expression retournée par la fonction :

```
def instruction(c: TrafficLightColor): String = c match
  case TrafficLightColor.Green ⇒ "go!"
  case TrafficLightColor.Yellow ⇒ "stop!"
  case TrafficLightColor.Red ⇒ "stop!"
```

Le métacaractère (dénoté _) agit comme un joker qui correspond à n'importe quelle expression. La définition suivante est donc équivalente à la précédente.

```
def instruction(c: TrafficLightColor): String = c match
  case TrafficLightColor.Green ⇒ "go!"
  case _ ⇒ "stop!"
```

De manière similaire, on peut utiliser une variable libre : comme le métacaractère, une variable libre correspond à n'importe quelle expression, mais en plus on pourra réutiliser la valeur correspondante dans la définition de l'expression retournée. Ainsi avec cette définition

```
def ordinal(n: Int): String = n match
  case 1 ⇒ "1st"
  case 2 ⇒ "2nd"
  case 3 ⇒ "3rd"
  case m ⇒ m.toString + "th"
```

la valeur de `ordinal(5)` est la chaîne de caractères `"5th"`.

L'utilisation du métacaractère ou d'une variable libre n'est pas limitée à l'expression entière, on peut s'en servir pour filter une sous-expression, et en particulier les arguments d'un constructeur d'un type de donnée algébrique. La fonction suivante utilise ainsi le pattern matching pour "déconstruire" une expression de type `IntPair` et en retourner le premier élément :

```
def first(p: IntPair): Int = p match
  case IntPair.P(x, _) ⇒ x
```

On peut aussi mélanger constructeurs, constantes, variables libre et métacaractère pour définir des filtres complexes :

```
def secondElementIsZero(l: List[Int]): Boolean = l match
  case _ :: 0 :: _ ⇒ true
  case _ ⇒ false
```

4 Ressources supplémentaires

— *La curryfication et ses applications*
Cet article illustre la curryfication en Scala.
https://blog.engineering.publicissapient.fr/2019/09/12/la-curryfication-et-ses-applications/

— *Scala 3 book – Algebraic data types*
Une documentation (en anglais) des types de données algébriques dans Scala 3. Attention la syntaxe pour déclarer ces types de données a été modifiée après Scala 2, qui est encore largement utilisé. Les autres ressources que vous pouvez trouver sur ce sujet risquent d'utiliser l'ancienne syntaxe.
https://docs.scala-lang.org/scala3/book/types-adts-gadts.html

— *Scala - jouer avec le pattern matching*
Le filtrage par motif (*pattern matching*) offre de nombreuses autres possibilités qui n'ont pas été abordés dans ce cours (cas conditionnels, filtrage sur les types...), et qui sont décrites dans cet article.
https://blog.engineering.publicissapient.fr/2012/01/11/scala-jouer-avec-le-pattern-matching/