Rapport — Modules à la ML vs Traits vs Classes de types vs Implicites

Pougala Biko — Cai Chaolei

17 mars 2022

1 Modules à la ML

En guise d'introduction, nous allons vous parler de 3 constructions clés chez les modules à la ML, c'est-à-dire les structures, signatures et foncteurs. Car au final on pourrait comprendre le sujet de notre présentation comme différent choix d'implémentation pour ces constructions.

1.1 Structure

Dans la programmation modulaire, une structure est la partie concrète qui contient les fonctions et les données. Au fait, le paradigme d'Ocaml est que *"tout est module"*, du coup, le code qui est contenu dans le fichier A va être interprété par le compilateur d'Ocaml comme le module A. Il est comparable à une instance d'objet en programmation orienté objet ou encore la partie *valeur* d'une variable.

1.2 Signature

Par opposition à la structure, la signature correspond plutôt à la partie abstraite du module. Elle contient une description du contenue attendu dans la structure. C'est un peu comme le patron d'un objet X, si un objet Y satisfait la description donnée par la signature de X, alors Y est un membre de l'objet X. Prenons la voiture comme exemple un peu plus concret, une voiture réelle est l'implémentation concrète du module *Voiture*, elle a 4 roues, 4 portes, un moteur et elle roule. La signature est alors une fiche descriptive de ce que doit être une voiture, par exemple elle doit avoir des roues, des portes, des moteurs et rouler. Nous pouvons alors appeler une voiture tout object qui satisfait cette fiche. Nous y ajoutons ainsi une couche d'abstraction pour l'utilisateur de l'interface, qui n'a pas besoin de connaître l'implémentation réelle pour faire appel à cette interface. Par exemple, nous pouvons interchanger les structures de données file et pile si nous utilisons la signature *Set*, comment est implémenté la file n'a pas d'importance pour nous, nous savons juste qu'il est possible d'appeler les fonctions ajouter ou encore enlever un élément depuis cette interface.

1.3 Foncteur

Nous pouvons décrire le foncteur comme une fonction applicable à une structure vers une autre structure. Au lieu d'avoir une fonction appliquée à un type de variable donné, le foncteur s'applique à tout module de signature donnée. Par exemple nous pouvons avoir un foncteur *dessiner* qui prend en argument un module de signature *dessinable* et nous renvoie un module *dessin*. Cet exemple illustre la reutilisabilité des modules via le foncteur, ainsi nous avons pas besoin d'écrire une fonction spécifique pour dessiner une voiture, un avion, un bateau. Il suffit juste que les objects que nous souhaitons dessiner respecte la déclaration de la signature *dessinable*.

1.4 Module de premiere classe

Chez les langages de programmation fonctionnels, les fonctions sont valeurs de première classe car nous pouvons les crées par des calculs, passées en argument à des fonctions ou renvoyées, comme n'importe quelles autres valeurs. Cette propriété n'est pas native au module, car c'est plutôt des structures statiques décidées à la compilation. Ainsi les modules de première de classe nous permettent de créer des modules décidés à l'exécution

2 Haskell

Haskell est un langage de programmation fonctionnelle dont la philosophie repose très fortement sur les fonctions mathématiques : le schéma des applications Haskell est relativement basique : des fonctions prenant un certain nombre d'arguments en entrée et renvoyant un certain nombre de valeurs en sortie. Il n'y a pas de notion d'état ou de variables globales. Les variables nulles n'existent pas non plus, toutes les fonctions doivent retourner des valeurs correspondant à la signature de la fonction. Haskell est en outre un langage fortement typé : toutes les erreurs de typage vont être détectées à la compilation. On peut en revanche donner une certaine liberté quant au type des valeurs en entrée et en sortie en introduisant des types génériques. Pour s'assurer en revanche que le fonctionnement de la fonction a du sens, on introduit des **contraintes de classes**. On décide de restreindre le champ des types possibles à une certaine classe de types.

2.1 Classes de types

Les classes de type sont des interfaces qui définissent un comportement. Si un type appartient à une certaine classe de types, alors il implémente le comportement que cette classe de type décrit. Par exemple, si on créé une fonction `circomference` qui calcule la circonférence d'un cercle étant donné son rayon :

```
circonference r = 2 * pi * r
```

Et qu'on laisse le compilateur déterminer le type de cette fonction, alors on verra en retour :

```
*Main> :t circonference
circonference :: Floating a => a -> a
```

Cette nomenclature signifie que la fonction `circonference` prend un seul argument en entrée de type générique (les types génériques portent de façon standard des noms de lettres) et renvoie en sortie une valeur du même type que le type en entrée. On ne souhaite pas en revanche calculer le rayon d'une chaîne de caractères ou d'un booléen, car cela n'aurait aucun sens. Le compilateur impose alors une contrainte de classe sur ce type générique et force ce dernier à appartenir à la classe de types `Floating`. Comment le compilateur parvient-il à ce résultat ? Il regarde les opérations qui sont effectuées par cette fonction : une multiplication entre un entier (2), une variable de type générique `r` et `pi`. Dans *Prelude*, `pi` est une fonction qui ne prend pas d'argument et renvoie en sortie une valeur appartenant à la classe de type `Floating`. L'opérateur `*` est implémenté par toutes les classes de types numériques (`Floating`, `Num`, `Fractional`..) mais puisque cet opérateur est utilisé avec la fonction `pi`, c'est bien la classe de type `Floating` qui est sélectionnée par le compilateur. Concernant l'entier, il existe une fonction `fromIntegral` appelée implicitement qui permet de convertir à la volée les nombres appartenant à la classes de types `Fractional` (les nombres entiers) vers n'importe quel type numérique, et en l'occurence vers `Floating`.
Chaque classe de types impose d'implémenter un certain nombre de fonctions et d'opérateurs, ou, alternativement, propose une implémentation par défaut pour les fonctions qui n'auraient pas été implémentées. On peut prendre par exemple la classe de type `Eq` :

```
class Eq a where
  (==), (/=)      :: a -> a -> Bool
  x /= y          =  not (x == y)
```

Les types appartenant à cette classe doivent implémenter les deux opérateurs mathématiques `==` et `/=`, qui sont par ailleurs eux-mêmes des fonctions. `Prelude` ne permet pas de poser de contraintes sur la sémantique de l'implémentation de ces fonctions : le programmeur peut choisir d'implémenter ces opérateurs comme bon lui semble. En revanche, la classe `Eq` impose la contrainte suivante : quelle que soit la manière dont on implémente l'opérateur `/=`, il faut qu'il renvoie l'opposé de ce que renvoie l'opérateur `==`.

3 Scala

3.1 Traits

Les Traits sont utilisés en Scala pour partager des interfaces et des champs entre différentes classes. Chaque trait encapsule des méthodes (qu'elles soient abstraites ou non) et des champs. Ils sont compilés par la JVM sous forme d'interfaces Java donc leur comportement est totalement inopérable. Les Traits permettent de créer des types qui vont être utilisés et implémentés par différentes classes.
Les Traits ont un comportement particulier en plus des simples interfaces Java : il est possible de composer des classes Scala avec une combinaison d etraits et de classes abstraites. C'est ce qu'on appelle les **mixins**. Par exemple avec l'extrait de code suivant :

```
abstract class A {
  val message: String
}
class B extends A {
val message = "Je suis une instance de la classe B"
}
trait C extends A {
  def messageFort = message.toUpperCase()
}
class D extends B with C
```

La nomenclature `extends X with Y` permet de composer des classes en héritant d'une classe `X` tout en y injectant des méthodes et comportements issus du Trait Y. Ceci a pour avantage de permettre de réutiliser des méthodes et des propriétés dans des contextes très différents ; en l'occurrence en les injectant dans des classes différentes implémentant des comportements divers. Ceci permet de contourner la restriction sur l'héritage multiple, que ni Scala ni Java ne prennent en charge nativement. Ceci permet de créer des classes au comportement autrement plus riche que ce qui est possible de réaliser avec les classes de types dans Haskell, notamment grâce au fait que les Traits et mixins permettent d'injecter des propriétés en plus de méthodes.

3.2 Implicites

Il est assez simple de comprendre l'usage des implicites en Scala, tout d'abord nous pouvons utiliser le mot-clé *implicit* afin d'obtenir des paramètres qui seront déterminés par le compilateur. Dans l'exemple suivant, dans le scope de ce morceau de code, il y a une variable name de type chaîne de caractère avec le mot-clé *implicit* puis une fonction qui affiche sur la console les 2 paramètres donnés(dont un implicite). Le compilateur à la lecture de l'appel de la fonction log, va alors chercher dans le scope le plus proche une variable de type chaîne de caractère et le lier à cet appel, ici, dans le scope le plus proche il y a bien une variable de ce type.

```
implicit val name: String = "default"
def log(msg: String)(implicit name: String): Unit = println(s"[$name] $msg")
log("init") // [default] init
```

Je n'ai pas trouvé les paramètres implicites intéressants car au final est-ce que c'est si utile d'avoir "des paramètre de défaut intelligent" ? En revanche, le cas des fonctions implicites l'est beaucoup plus car nous pouvons les voir comme des foncteurs intelligents. Imaginons que nous avons une classe *Data*, et que nous appelons la fonction *println* sur cette classe. Évidemment, le compilateur va jeter une erreur car la fonction println prend en paramètre une chaîne de caractères, cependant, avec les implicites, il est possible d'éviter cette erreur. En effet, il suffit de définir une fonction Data2String

```
implicit def Data2String(data: Data): String = ...
```

Ainsi, lors du traitement par le compilateur, même si println de Data n'existe pas, il existe une fonction de conversion de Data vers String, ce qu'il fait que le compilateur va appeler cette fonction de conversion avant la fonction println. Cela nous revient à écrire *println(Data2String(myData))* à la seule différence que c'est le compilateur qui va faire le boulot à notre place. Néanmoins, les implicites ne sont pas parfaites, voici quelques points que nous pouvons soulever :

— Il est évident que l'usage d'implicite demande l'existence de variable, fonction ou de classe implicite, si vous ne spécifier pas l'existence de telles implicites, le compilateur ne peut pas les détecter, logique.
— Les variables, fonction ou classe implicites doivent être clairement visible sans préfix, c'est-à-dire, vous pouvez nommer un implicit x mais pas $foo.x$
— Le compilateur effectue la recherche d'implicite avec priorité, par exemple s'il ne trouve rien dans le contexte actuel, il va aller dans les imports explicites, puis les imports implicit(import avec wildcard) ainsi de suite...
— La conversion d'implicite ne s'active qu'en cas d'erreur, si la compilation se termine sans erreur, alors l'implicite n'est pas utilisée. Par exemple dans notre cas, si la méthode toString() existe pour notre classe, l'erreur de println(Data) ne sera pas jeté.
— L'implicite a une profondeur de porté faible, par exemple, si nous effectuons un appel $x+y$, le compilateur peut déterminer *convertx2y(x) + y* mais il ne saura déterminer *convertx2y(convertz2x(z)) + y*
— L'implicite doit être singulière au sein d'un contexte, c'est-à-dire que dans un même contexte, nous pouvons avoir des implicites d'entier, de String, de Foo, de fonction Data-¿ String, Data-¿ Foo mais il ne devra y avoir qu'un seul par type. Sinon compilateur doit faire face à un conflit qu'il ne saura déterminer.

Les implicites sont agréables à utiliser car ils permettent d'avoir d'écrire du code plus concis et simple, il faudra juste faire attention au contexte de l'implicite à laquelle nous y referons afin de garder la lisibilité du code.

Références

[DL05] Derek Dreyer and Peter Lee. Understanding and evolving the ml module system (thesis summary), 2005.

[Drea] The haskell 98 language report. https://www.haskell.org/onlinereport/. Consulté le : 2022-03-09.

[Dreb] Tour of scala. https://docs.scala-lang.org/tour/traits.html. Consulté le : 2022-03-10.

[Lip11] Miran Lipovaca. *Learn You a Haskell for Great Good ! A Beginner's Guide*. No Starch Press, 2011.

[OSV08] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala*. Artima, 2008.