Programmation multiparadigme avec Scala
Le problème de l'expression et les traits

Jacques Noyé

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

17 janvier 2017

1 Objectif

L'objectif de ce TP est de se familiariser avec l'utilisation des *case classes* et des *traits* dans le contexte du *problème de l'expression*. Le problème de l'expression apparaît lorsqu'on considère un type de données avec des variantes et plusieurs opérations qui s'appliquent sur ces données. On va prendre dans ce TP, l'exemple d'expressions arithmétiques qu'il est possible de vérifier, d'évaluer, d'afficher...

Dans ce genre de situation, il y a classiquement deux organisations de base :

— une organisation *par fonction* qui décrit séparément chaque opération en traitant pour chaque opération toutes les variantes possibles ;

— une organisation *par objet* qui décrit séparément chaque variante du type de données en traitant pour chaque variante toutes les opérations possibles.

L'organisation *par fonction* favorise l'ajout de nouvelles fonctions, qui peut se faire sans toucher le code existant. Par contre, l'ajout de nouvelles variantes du type de données requiert de modifier chaque fonction existante.

L'organisation *par objet* facilite l'ajout de nouvelles variantes du type de données sous la forme de nouvelles classes. Par contre, l'ajout d'une nouvelle opération requiert de modifier chaque classe existante.

C'est cette apparente nécessité de choisir une de ces organisations et donc de favoriser une des deux dimensions de l'extensibilité, type de données ou opérations, qui a été nommée *problème de l'expression*. *Problème* parce qu'on aimerait laisser le choix ouvert. Il s'avère que les traits permettent de résoudre ce problème.

On va regarder un petit exemple qu'on va développer *par fonction* (le développement correspondant *par objet* devrait être clair) avant de voir comment résoudre le *problème de l'expression* avec les traits.

2 Point de départ

Récupérez sur Campus le projet Scala `InterpreterV0`.

Le projet définit un type d'expression `Exp` sous la forme d'un trait et un certain nombre de variantes concrètes de ce type sous la forme de *case classes* :

```
trait Exp
case class Plus(e1: Exp, e2: Exp) extends Exp // addition
case class Sub(e1: Exp, e2: Exp) extends Exp // subtraction
```

```
case class Minus(e: Exp) extends Exp // unary minus
case class IntLit(v: Int) extends Exp // integer literal
```

Une *case class* est une classe, distinguée par le préfixe `case`, munie d'un certain nombre de facilités syntaxiques :

- les paramètres du constructeur sont, sans qu'il soit nécessaire de le déclarer, des variables d'instance imutables ;
- le compilateur crée automatiquement un objet compagnon et l'usine associée au constructeur primaire ;
- il crée aussi une méthode `toString` qui affiche les objets comme des termes dont le foncteur est le nom de la classe et les arguments les paramètres du constructeur ;
- il crée des *extracteurs* qui permettent d'utiliser ces termes pour faire du *pattern matching* ; *i*l crée finalement des méthodes gérant l'égalité (deux instances sont égales si leurs paramètres respectifs sont égaux).

Ceci permet d'associer très simplement une fonction d'évaluation aux expressions de la manière suivante :

```
object Exp {
  def eval(e: Exp): Int = e match {
    case IntLit(v) => v
    case Minus(e1) => - eval(e1)
    case Plus(e1, e2) => eval(e1) + eval(e2)
    case Sub(e1, e2) => eval(e1) - eval(e2)
  }
}
```

Il s'agit là d'une amorce d'organisation par fonction.

3 Extension dans une organisation par fonction

Pour commencer, on va étendre cette première version en suivant l'organisation par fonction.

3.1 Ajout d'une opération d'affichage

La méthode `toString` disponible par défaut permet d'afficher une expression sous une forme abstraite. Ecrire une nouvelle opération `toString(e: Exp): String` permettant d'afficher une expression en utilisant la syntaxe concrète habituelle (avec des opérateurs infixes).

Attention à l'associativité des opérateurs, par exemple lors de l'affichage de l'expression :

```
Sub(IntLit(2),
    Sub(IntLit(1), IntLit(3)))
```

3.2 Ajout de booléens

On veut maintenant ajouter des booléens :

- des constantes booléennes `BoolLit` ;
- les opérations booléennes `Not`, `And`, `Or` ;
- l'opération de comparaison `Compare` qui compare deux expressions quelconque et s'évalue en un booléen ;
- la conditionnelle `If`.

On donne à ces constructions le même sens qu'elles peuvent avoir en Java ou en Scala.

Dans un premier temps, il faut ajouter un nouveau type de données `Val` définissant le type de données retourné par `eval` qui ne peut plus simplement être `Int` puisqu'il est aussi possible de retourner des booléens.

Rajoutez ensuite les nouvelles constructions et mettez à jour `toString` et `eval`. Les modifications ne sont pas localisées, il faut toucher à toutes les opérations.

3.3 Ajout d'une opération de vérification sémantique des constructions

La définition des expressions permet de construires des expressions qui n'ont pas de sens, par exemple une expression ajoutant deux booléens. L'évaluation de telles expressions lève une exception.

On veut préalablement à l'évaluation éliminer les constructions incorrectes à l'aide d'une nouvelle opération `check(e: Exp): T` qui retourne le type `T` d'une expression (`B` pour un booléen, `I` pour un entier).

Ajouter cette fonction. Cette fois, la modification est localisée.

4 Résolution du problème de l'expression

Reprendre la version initiale de l'interpréteur (avec des entiers) en adoptant une approche par objet mais en utilisant des traits plutôt que des classes, ainsi qu'un représentation des résultats de l'évaluation sous la forme du type de données `Val` afin qu'il soit extensible.

Par exemple, pour `IntLit` :

```
trait IntLit extends Exp { val v: Int }
val four = new IntLit { val value = 4 }
```

Il est maintenant possible de rajouter l'évaluation comme une nouvelle opération sans modifier le code existant en définissant une extension de `Exp` avec une méthode d'évaluation :

```
trait ExpE extends Exp { def eval: Val }
```

Et en étendant chacune des constructions avec sa définition de la méthode :

```
trait PlusE extends Plus with ExpE {
  val e1, e2: ExpE

  def eval = ...
}
```

Notez :

— la nécessité pour `PlusE` d'étendre à la fois `Plus` pour notamment disposer des opérations déjà définies et `ExpE` pour pouvoir être utilisé en postion d'une expression munie d'une méthode d'évaluation, par exemple comme opérande d'une addition elle-même ;

— la nécessité de raffiner le typage des attributs de `Plus` afin de pouvoir aussi les évaluer.

On peut faire de même avec la vérification :

```
trait ExpC extends Exp { def check: Val }
trait PlusC extends Plus with ExpC {
  val e1, e2: ExpC

  def check = ...
}
```

Complétez cette version avec les booléens et créez une version des expressions `ExpEC` qui permette à la fois de vérifier et d'évaluer une expression.