FIL1 – Modularité et typage – 2014/2015
Évaluation

2 décembre 2014

Pour répondre, compléter le paquet `eval` de l'archive à récupérer sur Campus. Finalement, déposer l'archive complétée sur Campus.

Remarques

— Le langage de programmation utilisé est Java, version 8 au moins.

— On utilise la bibliothèque `java.util` de l'API (*Application Programming Interface*) de Java. Voir l'annexe à la fin pour les détails utiles.

Dans le paquet `structuresAlgebriques`, on se donne neuf interfaces génériques correspondant à des structures algébriques classiques : semi-groupes, monoïdes, groupes, additifs et multiplicatifs, anneaux et corps. Tous les tests seront réalisés dans la fonction principale de la classe `eval.Test`.

Quelques calculs génériques

Exercice 1 (Un module de calculs) Dans un module de calculs, appelé `eval.Calculs`, on développe des fonctions génériques de calculs opérant sur les interfaces précédentes.

1.1 Définir une fonction `T division(T x, T y)` calculant la division de `x` par `y` (autrement dit, `x` sur `y`). On veillera à donner au paramètre `T` le majorant le plus grand possible (relativement à la relation de sous-typage), afin de rendre la fonction `division` la plus générale possible.

1.2 Généraliser l'opération `produit` en définissant une fonction générique `T produitNAire(List<T> l)`, spécifiée ainsi :

— si la liste `l` est vide, la fonction déclenche une erreur `IllegalArgumentException` [1],

— sinon, si le premier élément de la liste n'est pas l'élément neutre de la multiplication, la fonction déclenche une erreur `IllegalArgumentException`,

— sinon, la fonction renvoie le produit de tous les éléments présents dans la liste `l`.

Pour itérer sur la liste, se reporter à l'annexe. On veillera à donner au paramètre `T` le majorant le plus grand possible (relativement à la relation de sous-typage), afin de rendre la fonction `produitNAire` la plus générale possible.

1. Pour lancer une exception `E` : **throw new** `E();`.

Quelques structures algébriques Toutes les structures implémentées dans les exercices suivants sont immutables.

Exercice 2 (Les entiers naturels) L'ensemble des entiers naturels $\{0, 1, \ldots\}$ peut être considéré comme un monoïde additif et comme un monoïde multiplicatif. On représente cet ensemble par l'interface `Nat`. Pour implémenter cette structure algébrique, on utilise le patron de conception "Composite", qui traduit directement la définition inductive suivante : un entier naturel est soit nul soit le successeur d'un entier naturel. On définit donc une interface héritant de `Nat`, appelée `NatInductif`, utilisée pour factoriser des définitions, et ses deux classes d'implémentation, `Zero`, une classe singleton représentant zéro, et `Succ`, une classe implémentant les entiers naturels ayant un prédécesseur.

2.1 Compléter la définition de l'interface `Nat` de manière à hériter des interfaces génériques `MonoideAdd` et `MonoideMul`, ainsi que de l'interface générique `FabriqueNats`, permettant de fabriquer des entiers naturels. Définir une interface `NatInductif` par héritage de `Nat`. Y définir deux méthodes par défaut, `zero` et `un`, en utilisant les fabriques `creer`. L'interface `NatInductif` sera progressivement complétée par des méthodes par défaut.

2.2 Définir une classe d'implémentation `Zero`, un singleton représentant l'entier nul (zéro) implémenté à l'aide d'une classe `enum` (cf. l'annexe pour la syntaxe). Implémenter dans la classe `Zero` :
- les accesseurs, sachant que la classe n'a pas d'attributs et que la méthode `predecesseur` déclenche une exception `UnsupportedOperationException`,
- les opérations algébriques `somme` et `produit`,
- la méthode `toString`, mais pas la méthode `equals`, déjà définie dans une classe `enum` et impossible à redéfinir.

Implémenter dans l'interface `NatInductif` :
- la méthode par défaut `creer`(), permettant de créer un entier valant zéro.

2.3 Définir la classe d'implémentation `Succ`, dont les instances représentent les entiers non nuls, soit les successeurs, entiers ayant un prédécesseur. Cette classe a un attribut, de type `Nat` et le constructeur associé. Implémenter dans la classe `Succ` :
- les accesseurs,
- les opérations algébriques `somme` et `produit`, de manière récursive,
- la méthode `toString` et
- la méthode `equals`, de manière récursive.

Implémenter dans l'interface `NatInductif` :
- la méthode par défaut `Nat creer(Nat pred)`, permettant de créer le successeur de `pred`.

2.4 Tester les deux classes `Zero` et `Succ` en suivant le scénario suivant.
- Créer une fabrique de type `FabriqueNats`.
- Initialiser trois entiers naturels à zéro, un et deux, en utilisant la fabrique.
- Calculer la somme de deux et deux, et le produit de quatre fois deux.

— Afficher tous les entiers créés.

Par la suite, nous utiliserons une classe générique `Couple<T1, T2>` permettant de représenter des couples. Nous utiliserons aussi une interface générique `Representation`, permettant de représenter les éléments des structures algébriques avec des structures de données particulières (des couples pratiquement).
Les constructions suivantes utilisent toutes la technique de l'agrégation avec délégation.

Exercice 3 (Les entiers relatifs) À partir des entiers naturels, on peut définir les entiers relatifs. Une implémentation évidente consiste à rajouter à un entier naturel un signe. Nous étudions ici une implémentation plus raffinée, fondée sur une opération dite de symétrisation. Un entier relatif z est alors représenté par un couple d'entiers naturels (p, n), dont la différence vaut l'entier relatif : $z = p - n$. Comme il existe une infinité de tels couples, on doit quotienter l'ensemble des couples : deux couples sont équivalents s'ils correspondent aux mêmes différences.

$$(p, n) \equiv (p', n') \Longleftrightarrow p + n' = p' + n.$$

Cette équivalence sera implémentée par la méthode `equals` en Java.

3.1 Compléter l'interface `Z` de manière à hériter des interfaces suivantes :
- l'interface `Representation<Couple<Nat, Nat>>` permettant de représenter un entier relatif par un couple d'entiers naturels,
- l'interface `FabriqueRelatifs<Z, Nat>` permettant de fabriquer des entiers relatifs à partir de deux entiers naturels,
- l'interface générique `AnneauUnitaire`.

3.2 Compléter la classe `Relatif` implémentant l'interface `Z`.

3.3 Tester la classe `Relatif` suivant le scénario suivant.
- Créer une fabrique de type `FabriqueRelatifs<Z, Nat>`.
- Initialiser trois entiers relatifs à zéro, moins un et moins deux, en utilisant la fabrique.
- Calculer le produit de moins deux fois moins deux.
- Afficher tous les entiers relatifs créés.

3.4 Il apparaît que l'interface `Z` et la classe `Relatif` sont génériques en `Nat` : elles ne dépendent que des propriétés de monoïdes de cette structure. C'est en réalité un exemple d'une construction tout à fait générale, dite de symétrisation, permettant de définir un anneau unitaire à partir d'une structure qui est à la fois un monoïde additif et multiplicatif (un pré-anneau précisément). Compléter la déclaration de l'interface générique `Symetrise<T>` définissant le symétrisé d'un type `T` correspondant à un monoïde additif et à un monoïde multiplicatif. Cette interface hérite de l'interface `FabriqueRelatifs` et aussi de l'interface `Representation`, afin d'exprimer qu'un relatif est représenté par un couple de deux `T`. Compléter enfin la classe `Diagonale`, classe générique implémentant l'interface `Symetrise` et utilisant un couple d'attributs correspondant à une différence.
Remarque : la méthode `equals`, telle que spécifiée, engendre un avertissement.

3.5 Tester la classe `Diagonale<Nat>` suivant le scénario précédent.

- Créer une fabrique de type `FabriqueRelatifs`.
- Initialiser trois entiers relatifs à zéro, moins un et moins deux, en utilisant la fabrique.
- Calculer le produit de moins deux fois moins deux.
- Afficher tous les entiers relatifs créés.

Variance On s'intéresse au système de types de `Java`, et particulièrement à la relation entre la généricité et le sous-typage. Pour cet exercice, compléter l'unité de compilation `eval.Variance`.

Exercice 4 (Typage et variance) On considère l'interface générique `List` du paquet `java.util`. Dans le système de types de `Java`, les deux règles suivantes sont fausses.

Règle de covariance

> Si `B` est un sous-type de `A`, alors `List<B>` est un sous-type de `List<A>`.

Règle de contravariance

> Si `B` est un sous-type de `A`, alors `List<A>` est un sous-type de `List<B>`.

4.1 Compléter les fonctions `erreurCovariance` et `erreurContravariance` de manière à produire une erreur à l'exécution. Dans la fonction `erreurCovariance`, l'erreur doit provenir de l'application de la règle de covariance, simulée par l'application de la fonction `covariance` ; dans la fonction `erreurContravariance`, l'erreur doit provenir de l'application de la règle de contravariance, simulée par l'application de la fonction `contravariance`. Noter que les deux fonctions `covariance` et `contravariance` sont à la fois bien typées et exemptes d'erreurs à l'exécution, du fait de l'effacement des paramètres de types lors de la compilation.
Indication – Les erreurs doivent se produire lors de l'application de la méthode `f` à un objet de type réel `A` et de type déclaré `B`. Pour les produire, utiliser les instructions suivantes, dans le bon ordre :

- initialisation d'une liste chaînée,
- ajout d'un élément à la liste,
- récupération de l'élément placé dans la liste,
- conversion par variance, obtenue en appelant les fonctions `covariance` ou `contravariance`,
- appel de la méthode `f`.

4.2 Définir les versions génériques `covarianceG` et `contravarianceG` de `covariance` et `contravariance`. Elles seront paramétrées par un type, et un seul, et utiliseront le joker (noté ?) exactement une fois. Compléter les fonctions correspondantes `erreurCovarianceG` et `erreurContravarianceG`, variantes de `erreurCovariance` et `erreurContravariance` utilisant respectivement `covarianceG` et `contravarianceG`.

Annexe

Le paquet `java.util` contient l'interface générique `List` et une classe générique d'implémentation `LinkedList`, possédant un constructeur sans paramètre. Pour parcourir une liste `l` de type `List<T>`, on procède ainsi.

```
for(T e : l){
  @@@ // Code utilisant e
}
```

Une classe `enum` se déclare ainsi, dans une version simplifiée mais suffisante.

```
enum ClasseEnumeration implements InterfaceImplementee {
  CONSTANTE1, @@@ ; // Déclaration des constantes
  @@@ // Implémentation des méthodes de l'interface
```