Aix*Marseille université

Domaine Sciences et Technologies
LICENCE INFORMATIQUE - DU CCIE

Algo. et Structures Discrètes : TP 2

Année 2018-19

Listes inductives

1 Introduction

Dans ce TP, nous allons implémenter la structure de données de *listes* [1] correspondant aux listes définies dans le cours par l'induction :

$$\frac{}{vide} \qquad \frac{l}{a :: l}\, a \in E$$

Une liste est donc soit *vide*, soit composée d'une liste à laquelle on a ajouté un élément en tête. Toute liste est ainsi de la forme $a_1 :: a_2 :: a_3 :: \cdots a_n :: vide$, où $a_1, \ldots, a_n \in E$.

Etant donné une liste non vide $l = a :: l'$, nous appelons a la *tête* de la liste l, et l' la *queue* de la liste l. Pour simplifier, nous n'allons considérer que les listes d'entiers mais ce que nous décrirons s'applique à des listes de n'importe quel type.

2 l'interface `IntList`

L'interface est la base de l'implémentation d'une structure de données. C'est elle qui donne la signature des méthodes utilisables sur la structure de donnée. Il s'agit donc d'une description des méthodes, entièrement indépendante de l'implémentation choisie. Nous démarrons avec l'interface suivante que nous enrichirons par la suite.

```
public interface IntList
{
    /** @return   la tête de la liste si elle est non-vide
     */
    int getHead();

    /** @return    la queue de la liste si elle est non-vide
     */
    IntList getTail();

    /** @return   vrai si la liste est vide et faux sinon
     */
    boolean isEmpty();

    /** @param  n     l'entier à ajouter dans la liste
     *  @return       la liste n :: this
     */
    IntList cons(int n);

     /** @return       une liste vide
     */
    IntList emptyList();

}
```

Remarquez qu'ici, nous avons plusieurs possibilités pour implémenter cette classe. Nous allons choisir pour commencer une implémentation un peu générique, qui peut s'appliquer à la plupart des définitions inductives en définissant une classe pour les axiomes et une classe par règle. Nous utiliserons aussi une classe abstraite qui sera utilisée pour écrire des méthodes supplémentaires.

1. Il existe plusieurs implémentation de cette structure de données dans l'API Java.

3 La classe abstraite `AbstractIntList`

Afin de limiter au maximum les erreurs d'implémentation, et de produire du code facilement modifiable, nous allons séparer l'implémentation des primitives `getHead, getTail, cons, isEmpty, emptyList` de celle des autres méthodes (longueur du liste, affichage d'une liste, etc) qui n'agiront sur les listes **que par l'intermédiaire des primitives**.

Pour cela, on utilise une classe abstraite `AbstractIntList` : c'est elle qui contiendra l'implémentation de ces méthodes supplémentaires. Les primitives seront implémentées dans des classes héritant de `AbstractIntList` .

Cette façon de faire limite grandement le risque d'erreurs : si les primitives sont correctement implémentées, alors les méthodes les utilisant peuvent toujours être ramenés à des algorithmes sur les listes telles que nous les avons définies inductivement, et ce, indépendamment de l'implémentation choisie.

Pour le moment, la classe `AbstractIntList` se contente de restreindre les sorties des méthodes définies dans l'interface au type `AbstractIntList`. Cela permettra de définir dans cette classe des méthodes récursives privées.

```
public abstract class AbstractIntList implements IntList{
        public abstract AbstractIntList cons(int n);
        public abstract AbstractIntList  getTail();
        public abstract AbstractIntList emptyList();
}
```

4 Implémentation inductive des listes

Pour implémenter les listes, nous allons tout simplement suivre la définition inductive [2] des listes : une liste est soit une liste vide, soit une paire (entier, liste). En Java, cela revient à considérer deux classes : `NonEmptyIntList` et `EmptyIntList` qui hériteront de la classe abstraite `InductiveIntList`. Une liste sera une instance de l'une de ces deux classes :

- la classe `NonEmptyIntList` représentera les listes non-vides d'entiers. Ces listes comporteront au moins un élément : elles seront donc définies par leur premier élément (la tête) et la liste constituant leur queue.
- la classe `EmptyIntList` représentera les listes vides d'entiers.

Les classes `InductiveIntList`, `NonEmptyIntList`, `EmptyIntList` implémenteront les fonctions de base sur les listes (les primitives), et seront les seules à pouvoir modifier directement les listes.

4.1 Implementation de la classe `InductiveIntList`

Pour le moment, cette classe a pour seul objectif l'encapsulation des classes `NonEmptyIntList` et `EmptyIntList`. Elle est donc vide :

```
public abstract class InductiveIntList extends AbstractIntList {
}
```

4.2 Implémentation de la classe `NonEmptyIntList`

Elle doit étendre la classe `InductivetIntList`.

~~Exercice 1~~ En sachant qu'une liste non vide est constituée d'un premier élément (un entier) et d'une liste (vide ou pas) constituant la queue, quels sont les attributs de votre classe et leur type ?

~~Exercice 2~~ Ajouter un constructeur `NonEmptyList(int head, AbstractIntList tail)` permettant de construire une liste non vide composée de la tête `head` et de la queue `tail`.

2. Cette méthode n'est pas forcément la plus directe pour les listes, mais montre une façon générale d'implémenter une structure de données correspondant à une définition inductive.

Exercice 3 Réaliser une implémentation des méthodes accesseurs `getHead()` et `getTail()`.

Exercice 4 Réaliser une implémentation de la méthode `isEmpty()` (pour mémoire, les instances de `NonEmptyIntList` sont des listes non vides).

Exercice 5 La méthode `cons` crée une nouvelle liste dont le premier élément est l'entier passé en paramètre et la queue est la liste sur laquelle la méthode s'applique. Réaliser une implémentation de cette méthode pour la classe `NonEmptyIntList`.

Exercice 6 La méthode `emptyList` crée une nouvelle liste vide. Réaliser une implémentation de cette méthode pour la classe `NonEmptyIntList`.

4.3 Implémentation de la classe `EmptyIntList`

Nous allons maintenant implémenter la classe `EmptyIntList` qui étend `AbstractIntList` et dont les instances sont des listes vides d'entiers. Puisqu'il s'agit d'implémenter une liste vide, aucun attribut n'est nécessaire.

Exercice 7 Réaliser une implémentation des méthodes `isEmpty`, `cons` et `emptyList`.

Exercice 8 Pour mémoire, les opérations `tete` et `queue` ne sont pas définies sur la liste vide (d'entiers). L'implémentation de méthodes `getHead` et `getTail` consistera simplement à lever une exception pour signaler cette non-définition au moyen de l'instruction `throw new NoSuchElementException();`

4.4 Optimisation du code

Vous avez certainement remarqué que les primitives de construction ont la même implémentation dans les deux classes `NonEmptyIntList` et `EmptyIntList`. Déplacez les donc dans la classe `InductiveIntList`.

4.5 Tests de votre implementation

Même si nous ne disposons pas encore de méthodes d'affichage des listes, nous pouvons commencer à effectuer quelques tests de base. Vous réaliserez ces tests dans une classe `Tests` contenant seulement une méthode `main` :

```
public class Tests {

        public static void main(String[] args) {
        // Tests des listes inductives
        // construction d'une liste vide
        AbstractIntList lvide = new EmptyList();
        // construction d'une liste de longueur 1
        AbstractIntList l1 = lvide.cons(1);
        // construction d'une liste de longueur 2
        AbstractIntList l2 = l1.cons(2);
        // construction d'une liste de longueur 3
        AbstractIntList l3 = l2.cons(3);
       }
}
```

Exercice 9 Vérifiez que la liste `lvide` est vide ; c'est-à-dire que l'application de la méthode `isEmpty` cette liste retourne faux.

Exercice 10 Vérifiez que les listes `l1, l2, l3` ne sont pas vides.

Exercice 11 Vérifiez sur les listes `l1, l2, l3` que la tête d'une liste $e :: l$ est e.

Exercice 12 Vérifiez sur les listes `l1, l2, l3` que la queue d'une liste $e :: l$ est l.

5 De nouvelles méthodes

Nous allons enrichir notre classe `IntList` par de nouvelles méthodes qui seront implémentées directement dans la classe `AbstractIntList`. Vous n'avez donc plus à modifier les classes `EmptyIntList` et `NonEmptyIntList`

~~Exercice 13~~ Ajoutez à `AbstractIntList` une méthode `public String toString()` qui retourne une chaîne de caractère représentant la liste (sous la forme $a_1 :: a_2 :: \cdots :: a_n :: vide$). Notez que cette méthode n'a pas besoin d'être ajoutée à `IntList` car il s'agit d'une redéfinition d'une méthode commune à tous les objets Java.

~~Exercice 14~~ Exprimer la longueur d'une liste non vide en fonction de la longueur de sa queue. En déduire une implémentation (récursive) de la méthode `int length()` retournant la longueur d'une liste.

~~Exercice 15~~ Ecrivez une méthode récursive renvoyant la somme des éléments d'une liste.

~~Exercice 16~~ L'opérateur `==` ne permet pas de tester que deux listes possèdent le même contenu (c'est-à-dire qu'elles sont égales du point de vue des listes d'entiers). Pour pallier ce problème, nous allons implémenter une méthode `equals` pour tester l'égalité de deux listes d'entiers. À noter que :

— si les deux listes sont vides alors elles sont égales
— si l'une est vide et l'autre non alors les deux listes sont différentes
— si les deux sont non-vides alors elles sont égales si leurs têtes et leurs queues sont égales.

Proposer une implémentation de la méthode `equals(AbstractIntList list)` .

~~Exercice 17~~ Ecrivez une méthode qui réalise la concaténation de deux listes.

~~Exercice 18~~ Ecrivez une méthode qui renverse une liste. Si vous utilisez une fonction auxiliaire, vous pouvez normalement la rendre privée puisqu'elle n'a pas à être vue par l'utilisateur. Elle n'apparait donc pas dans l'interface.

6 Une autre implémentation des listes

Après mûre réflexion, nous pensons que nous n'avons pas choisi la meilleur implémentation pour les listes. Heureusement, nous avons pris soin de bien séparer l'implémentation des primitives, de l'implémentation des autres méthodes, ainsi, nous pouvons changer d'implémentation sans modifier `AbstractIntList` et donc sans avoir à réécrire toutes les méthodes ajoutées à la section 5.

La nouvelle implémentation représentera une liste sous forme d'une *liste chaînée*. Que sont donc les listes chaînées ? Pour les décrire, il nous faut rentrer plus profondément dans la représentation des objets. Chaque objet est identifié par une référence ; cette référence désigne donc l'objet en question. La liste chainée est alors définie par un certain nombre de maillons, chacun contenant une donnée (un entier pour une liste d'entiers) mais également un lien vers un autre maillon. On peut alors passer d'un maillon à une autre, égrainant ainsi les éléments de la liste. Cette suite de maillons tous différents deux à deux permet de retrouver l'idée d'une séquence ordonnée de données de taille arbitraire.

6.1 Implémentation des maillons

Les maillons seront représentés par la class `Cell`. Les objets de cette classe posséderont deux attributs. Un attribut `data` de type `int` qui contiendra la donnée du maillon et un attribut `nextCell` de type `Cell` qui sera le maillon suivant. Pour le dernier maillon de cette suite, cet attribut `nextCell` sera la référence `null`.

~~Exercice 19~~ Implémenter la classe `Cell` en donnant notamment son constructeur qui prend en paramètre un entier et un objet de type `Cell`.

~~Exercice 20~~ Compléter la classe `Cell` en implémentant les assesseurs et mutateurs suivants : `getDataCell()`, `getNextCell()`, `setDataCell()`, `setNextCell()`.

6.2 Implémentation des listes chaînées

Une liste (chainée) va simplement être représentée par un maillon, à savoir le premier maillon de la liste. Le type de ces objets sera `LinkedIntList`, cette classe héritant de la classe abstraite `AbstractIntList`. On définit ainsi

```
public class LinkedIntList extends AbstractIntList
{
    // variables d'instance
    private Cell first;

    /** Constructeur d'objets de classe LinkedIntList
     */
    public LinkedIntList()
    {
        // initialisation des variables d'instance
        first = null;
    }

    ......

}
```

Le constructeur de la classe `LinkedIntList` produit ainsi la liste vide.

Exercice 21 Compléter la définition de cette classe en implémentant les méthodes `cons`, `getHead`, `getTail`, `isEmpty` et `emptyList`.

La méthode `getTail` retournera une nouvelle liste égale à la queue de la liste sur laquelle elle est invoquée.

Exercice 22 Testez à nouveau les méthodes construites dans la section 6, sur des listes de type `LinkedIntList`.