La récursivité (épisode 2)

NFA032

Présentation n°8

Les types récursifs

- Préambule
  - Séance passée : les méthodes récursives
    - La récursivité se traduit par le fait qu'une méthode s'appelle elle-même
  - Cette séance : les types (classes) récursifs
    - Dans la définition d'une classe *R* récursive, au moins l'un de ses attributs (variables non statiques) :
      - a pour type cette même classe *R* (récursion *directe*)
      - a pour type une classe *C*, qui elle a un composant de type *R* (récursion *indirecte*), etc…

Un détour vers la théorie des types

- En théorie des types :
  - Soit un type est primitif
    - Il n'est pas décomposable
  - Soit un type est construit (non primitif) :
    - Il s'appuie sur des types existants (primitifs ou pas)
    - Il les combine selon des opérations appelées *constructions de types*
      - Constructeurs génériques (soient A et B des types) :
        - » La somme de types : $A+B$
          - un type énuméré est une somme de types
        - » Le produit de types : $A \times B$
          - un type tableau est un produit d'un même type
          - un type enregistrement est un produit de types quelconques
        - » L'exponentiation de type : B^A, aussi notée $A \rightarrow B$
          - le type d'une méthode ou d'une fonction est de ce genre
      - Constructions spécifiques (ou ad hoc) :
        - » La notion de pointeur (nombreux langages) :
          - $\star A$ (C/C++) ou **`access`** *A* (Ada) ou **`ref`** *A (OCaml)*
        - » La notion de classe finale en Java : **`final`** *A*

Un détour vers la théorie des types

- Assimilons un type *T* à un ensemble : celui de ses valeurs
  - Hypothèse : tous les types sont disjoints
    - une valeur n'a qu'un seul type
      - donc deux types ne peuvent avoir une valeur en commun
  - Somme des types *A* et *B* :
    - *A*+*B* assimilable à leur union ensembliste $A \cup B$
    - traduit la notion naturelle de choix exclusif (*ou*)
      - un seul existe à la fois
  - Produit des types *A* et *B* :
    - $A \times B$ assimilable à leur produit cartésien ensembliste $A \times B$
    - traduit la notion naturelle de composante (*et*)
      - les deux (ou plus) existent à la fois

Théorie des types et récursion

- Un type récursif
  - est un type construit : il est donc défini par une *expression de type*
  - un des types qui composent son expression est lui-même
- Exemple (ce n'est pas du Java, juste inspiré) :
  - Types non récursifs :
    - False
      - type n'ayant qu'une seule valeur : *false*
    - True
      - type n'ayant qu'une seule valeur : *true*
    - Boolean = False + True
      - type ayant deux valeurs, *true* et *false* ; une valeur booléenne ne peut donc être au choix (exclusif) que soit *true*, soit *false*.
    - tout type énuméré est une somme de types primitifs (un par littéral)
    - Zéro
      - Type n'ayant qu'une seule valeur : l'entier naturel *0*
  - Types récursifs :
    - Naturel = Zéro +(Zéro $\times$ Naturel)
      - L'entier naturel 0 est associé à la valeur *0*
      - L'entier naturel 1 est associé à la valeur (*0*, *0*)
      - L'entier naturel 2 est associé à la valeur (*0*, (*0*, *0*))
    - $\text{Tableau}[n, T] = \begin{cases} T \times \text{Tableau}[n-1, T] & \textbf{si } n > 0 \\ \text{Vide}[T] & \textbf{si } n = 0 \end{cases}$

Théorie des types et récursion

- En informatique, de nombreux types sont récursifs
  - N'importe quel type peut être construit à partir d'une combinaison appropriée des types True et False
  - Tout type ayant un nombre potentiellement infini dénombrable de valeurs est nécessairement récursif
    - Les nombres
    - Les collections
      - Les piles, les files
      - Les listes, les arbres, les graphes
      - Les tableaux, les dictionnaires
      - Les ensembles, les multi-ensembles
  - Qui dit *type récursif* dit en général *fonctions récursives*
    - Une fonction qui manipule une valeur d'un type récursif exploite *naturellement* la récursivité du type
- On montre que la construction de type B^A est suffisante :
  - Somme et produit peuvent être définis à partir de l'exponentielle de types
  - La théorie des types est une forme particulière (celle liée aux aspects calculables) de la *théorie des catégories*.

Théorie des types et Java

- classe Java
  - est associée à un type
    - Nouveau type intégré à une expression de types (implique les classes de son ascendance et de sa descendance)
- héritage en Java
  - est une construction de type intégrant à la fois somme (règle du polymorphisme) et produit (ajout de propriétés à celles héritées)
    - Langage objet
      - l'héritage suffit pour définir n'importe quel type somme ou produit
    - Langage non objet
      - Les deux constructions de type somme et produit sont explicites :
        - » **`union`** et **`struct`** en C
        - » **`record`** *à choix* et **`record`** en Ada83

Un type récursif typique : les listes

- Simplification : listes d'entiers
  - En principe, les éléments d'une liste sont tous d'un même type *T*, *T* étant quelconque $\Rightarrow$ nécessiterait la notion de *généricité*, pas au programme NFA032
  - Une liste est assimilable à un tableau, sauf que :
    - sa taille peut varier en cours de calcul (peut être potentiellement infinie), au contraire de celle des tableaux, qui est fixée (librement tout de même) à la création de chaque tableau
- Définition informelle :
  - Toute liste est :
    - soit *vide*
    - soit *non vide*, et dans ce cas :
      - elle est associé à un élément (entier) appelé son *premier* (ou sa *tête*)
      - elle est associée à une liste appelée son *reste*

Les listes

- Définitions précises
  - définition de type : $\text{Liste} = \text{Vide} + \text{Entier} \times \text{Liste}$
  - type abstrait :

type :

Liste

opérations :

vide : Liste
ajout : entier × Liste → Liste
estVide : Liste → booléen
premier : Liste → entier
reste : Liste → Liste

axiomes :

en supposant que L est une liste (L : Liste)
et e un entier (e : entier), alors :

premier(ajout(e, L)) = e
reste(ajout(e, L)) = L
estVide(vide) = vrai
estVide(ajout(e, L)) = faux

erreurs :

premier(vide) est indéfini
reste(vide) est indéfini

Réaliser les listes en Java

- Version 1
  - Toute liste a pour type `Liste`
  - La liste vide est représentée par l'objet spécial **`null`**
    - on profite de cette particularité en Java :
      - Toute variable de type *T*, où *T* n'est pas primitif, peut référencer **`null`**
      - C'est l'une des facettes du polymorphisme des variables :
        - » Le type de **`null`**, nommé `NullType`, est compatible avec n'importe quel type non primitif
  - Toute liste non vide est un objet de type `Liste` :

```
class Liste {
  int   _premier;
  Liste _reste;

  ...
}
```

Les listes - version 1

- Détails du code
- Exercices

  `longueur(L)`

  `dernier(L)`

  `compter(L, e)`

  `effacer(L, e)`
- Avantages :
  - Facilement définissable
  - Optimal est terme de gestion mémoire
- Inconvénients :
  - La liste vide n'est pas un objet de type `Liste`
    - Il faut nécessairement définir des fonctions, pas des méthodes

Réaliser les listes en Java

- Version 2
  - posons $\text{Chaînon} = \text{Vide} + \text{Entier} \times \text{Chaînon}$
  - donc $\text{Liste} = \text{Chaînon}$
    - Encapsulation de la notion de liste version 1 dans une nouvelle notion de liste.
- En Java :

```
class Liste {
  Chainon _premierChaînon;

  ...
}
```

```
class Chainon {
  int     _élément;
  Chainon _suivant;

  ...
}
```

Les listes - version 2

- Détails du code, et la classe des chaînons
- Avantages :
  - la liste vide est un objet de type `Liste`

    ```
    vide._premierChaînon == null
    ```

  - Toutes les opérations du type abstrait peuvent être réalisées sous forme purement objet (méthodes)
- Inconvénients :
  - Il peut exister plusieurs objets représentant une liste vide
    - Possibilité de faire en sorte que ce ne soit pas le cas (technique du singleton, étudiée dans la version 3)
  - Une liste vide consomme un peu d'espace mémoire

Réaliser les listes en Java

- Version 3
  - L'héritage est exploité pour exprimer la somme de type
    - $\text{Liste} = \text{ListeVide} + \text{ListeNonVide}$

```
class Liste {
   ... // aucun attribut
}
```

```
class ListeVide extends Liste
{
  // aucun attribut
  ...
}
```

```
class ListeNonVide extends Liste
{
   int    _premier;
   Liste  _reste;

   ...
}
```

Les listes – version 3

- Propriétés attendues
  - Interdire de pouvoir créer un objet de type `Liste`
    - seulement des objets `ListeVide` ou `ListeNonVide`

      $\Rightarrow$ faire de `Liste` une interface ou une classe abstraite
  - un seul objet pour représenter la liste vide (le test de vacuité de `L` pourrait se traduirait aussi bien par `L.estVide()` que par `L==vide`)
    - `vide` est une constante
    - il faut garantir qu'il n'y aura qu'une seule instance de `ListeVide` au cours d'une exécution (patron de conception *Singleton*)
  - offrir quelques fonctions pour construire des listes simples
    - Notion de *fabrique* (patron de conception éponyme)

Les listes – version 3

- Détails du code
- Avantages :
  - conception purement objet
    - tout est objet
    - les opérations sont des méthodes
    - aucun test
      - La liaison dynamique garantit l'appel du bon code
- Inconvénients :
  - conception purement objet !
    - consommation mémoire
    - suppose que la liaison dynamique soit efficacement compilée

Manipuler des listes en Java

- Exercices