Université Montpellier-II
UFR des Sciences - Département Informatique - Licence Informatique
Programmation Applicative et Récursive

Cours No 8 : Abstraction de données, Arbres binaires, Dictionnaires

Notes de cours
Christophe Dony

15 Abstraction de données

15.1 Définitions

Donnée : entité manipulée dans un programme, définie par un type.

ex : Les entiers 1, 3 et 5 sont des données d'un programme réalisant des calculs.

Abstraction de données : processus permettant de rendre des données abstraites, c'est à dire utilisables via un ensemble de fonctions constituant leur **interface** sans que l'on ait à connaître ou sans que l'on puisse connaître (selon la philosophie "prévention/répression" du langage) leur représentation en machine (dite **représentation interne**).

Type Abstrait : Définition abstraite (et par extension implantation) d'un type de données sous la forme de l'ensemble des opérations que l'on peut appliquer à ses éléments indépendamment de l'implantation de ces opérations et de la représentation interne de ces éléments.

Interface : ensemble des fonctions que l'on peut appliquer aux éléments d'un type abstrait.

On peut éventuellement distinguer dans l'**interface**

- la partie *création*, permettant de créer de nouveaux éléments du nouveau type,
- la partie *accès* pour accéder aux propriétés (lorsque ces fonctions d'accès appartiennet à l'interface) des éléments
- la partie *utilisation* ou *fonctionnelle* ou encore *métier* permettant de les utiliser.

La programmation par objet a généralisé la création de nouveaux types de données. L'interface de création est constituée de la fonction `new` et d' un ensemble de fonctions d'initialisation (généralement nommées *constructeurs*). On y parle également d'*accesseurs* et de fonctions "métier".

16 Définition de nouveaux types abstraits

Pascal : *Enregistrement - Record*

C, Scheme : *Structure - Struct*

JAVA : *Classe - Class* - Réalisation de l'encapsulation "données - fonctions".

16.1 Un exemple : les nombres rationnels (d'après Abelson Fano Sussman

Interface

création : make−rat
accès : denom numer
métier : +rat −rat ∗rat /rat =rat

Implantation

```
(define (+rat x y)
    (make-rat
        (+ (∗ (numer x) (denom y))
            (∗ (denom x) (numer y)))
        (∗ (denom x) (denom y))))

(define (∗rat x y)
    (make-rat
        (∗ (numer x) (numer y))
        (∗ (denom x) (denom y))))
```

Interface de création, Représentation No 1 : doublets, sans réduction

```
(define (make-rat n d) (cons n d))
(define (numer r) (car r))
(define (denom r) (cdr r))
```

Les différents niveaux d'abstraction

```
utilisation des rationnels
−−−−−−−− +rat −rat make-rat numer denom
représentation des rationnels, Utilisation des doublets
−−−−−−−− doublets : cons, car, cdr
représentation des doublets
−−−−−−−− ???
```

Interface de création, Représentation no 2, vecteurs et réduction

Vecteur : Collection dont les éléments peuvent être rangés et retrouvés via un index, qui occupe moins d'espace qu'une liste contenant le même nombre d'éléments.

Manipulation de vecteurs.

```
(make-vector taille) : création vide
(vector el1 ... eln) : définition par extension
(vector-ref v index) : accès indexé en lecture
```

(vector-set! v index valeur) : accès indexé en écriture

```
(define (make-rat n d)
  (let ((pgcd (gcd n d)))
    (vector (/ n pgcd) (/ d pgcd))))

(define (numer r) (vector-ref r 0))

(define (denom r) (vector-ref r 1))
```

17 L'exemple des Arbres binaires

Graphe : ensemble de sommets et d'arêtes

Graphe orienté : ensemble de sommets et d'arcs (arrête "partant" d'un noeud et "arrivant" à un noeud)

Arbre : graphe connexe sans circuit tel que si on lui ajoute une arrête quelconque, un circuit apparaît. Les sommets sont appelés **noeuds**.

Exemple d'utilisation : Toute expression scheme peut être représentée par un arbre.

Arbre binaire :

- soit un arbre vide

- soit un noeud ayant deux descendants (fg, fd) qui sont des arbres binaires

Arbre binaire de recherche : arbre dans lesquels une valeur$v$ est associé à chaque noeud n et tel que si $n1 \in fg(n)(resp. fd(n))$ alors $v(n1) < v(n)$ (resp. $v(n1) > v(n)$)

Arbre partiellement équilibré : la hauteur du SAG et celle du SAD diffèrent au plus de 1.

17.1 Arbres binaires de recherche

— Création :
(make-arbre v fg fd)
— Accès aux éléments d'un noeud n :
(val-arbre n), (fg n), (fd n)
— Test
(arbre-vide ? n)
— interface métier :
  — (inserer valeur a) : rend un nouvel arbre résultat de l'insertion de la valeur n dans l'arbre a. L'insertion est faite sans équilibrage de l'arbre.
  — (recherche v a) : recherche dichotomique d'un élément dans un arbre
  — (afficher a) : affichage des élements contenus dans les noeuds de l'arbre, par défaut affichage en profondeur d'abord.

17.1.1 Représentation interne - version 1

Consommation mémoire, feuille et noeud : 3 doublets.

Simple à gérer.

```
(define (make-arbre v fg fd)
  (list v fg fd))

(define (arbre-vide? a) (null? a))

(define (val-arbre a) (car a)) ;;

(define (fg a) (cadr a))

(define (fd a) (caddr a))
```

17.1.2 Insertion sans duplication

```
(define (insere n a)
  (if (arbre-vide? a)
      (make-arbre n () ())
      (let ((valeur (val-arbre a)))
        (cond ((< n valeur) (make-arbre valeur (insere n (fg a)) (fd a)))
              ((> n valeur) (make-arbre valeur (fg a) (insere n (fd a))))
              ((= n valeur) a)))))
```

```
(define a (make-arbre 6 () ()))
(insere 2 (insere 5 (insere 8 (insere 9 (insere 4 a)))))
```

17.1.3 Recherche dans un arbre binaire

Fonction booléenne de recherche dans un arbre binaire. La recherche est dichotomique.

```
(define (recherche v a)
  (and (not (arbre-vide? a))
      (let ((valeur (val-arbre a)))
        (cond ((< v valeur) (recherche v (fg a)))
              ((> v valeur) (recherche v (fd a)))
              ((= v valeur) #t)))))
```

17.1.4 Seconde représentation interne

Coût mémoire, noeud (3 doublets), feuilles (1 doublet).

Nécessite un test supplémentaire à chaque fois que l'on accède à un fils.

```
(define (make-arbre v fg fd)
  (if (and (null? fg) (null? fd))
      (list v)
      (list v fg fd)))
```

```
(define (fg n) (if (null? (cdr n)) () (cadr n)))

(define (fd n) (if (null? (cdr n)) () (caddr n)))
```

Toutes les autres fonctions de manipulation des arbres binaires (interface fonctionnelle) sont inchangées.

18 L'exemple des Dictionnaires

Le type dictionnaire est un type récurrent en programmation, on le trouve dans la plupart des langages de programmation (java.util.Dictionary).

En scheme, il a été historiquement proposé sous le nom de listes d'association. Une liste d'associations est représentée comme une liste de doublets ou comme une liste de liste à deux éléments.

18.0.1 Interface de manipulation

```
>(define l1 '((a 1) (b 2)))
>(assq 'a l1)
(a 1)
>(assq 'c l1)
#f
>(assoc 'a l1)
(a 1)
>(define l2 '( ((a) (une liste de un élément)) ((a b) (deux éléments)) ))
>(assq '(a b) l2)
#f
>(assoc '(a b) l2)
((a b) (deux éléments))
```

18.0.2 Interface : construction et manipulations

Scheme n'a pas prévu d'interface de construction, on construit directement les listes.

Ceci implique que l'on ne peut pas changer la représentation interne des dictionnaires.

Imaginons une interface de construction

```
(define prem-alist car)
(define reste-alist cdr)
(define null-alist? null?)
(define make-alist list)
(define (add clé valeur l)
    ;; rend une nouvelle aliste incluant une nouvelle liaison clé−valeur
    ;; ne vérifie pas si la clé était déjà utilisée
    (cons (list clé valeur) l))
```

```
(define (my-assoc clé al)
```

```
(let loop ((al al))
   (cond ((null-alist? al) #f)
      ((equal? (car (prem-couple al)) clé) (prem-couple al))
      (else (loop (reste-alist al)))))
```