Abstraction de données - Arbres binaires

Université Montpellier-II - UFR - FLIN304 - Programmation Applicative -
Cours No 8
C.Dony

15 Abstraction de données

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 : représentation interne de données d'un certain type accessible via un ensemble de fonctions constituant son interface.

Interface : ensemble de fonction de manipulation de données abstraites.

Type : implantation de l'abstraction de donnée.

15.1 Définition de nouveaux types abstraits

Pascal : *Enregistrments - Records*

C : *Structures - Structs*

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

Exemples de types abstraits :

- rationels - représentation : int x int - interface : +-rat, ...
- date - représentation : int x int x int - interface : ¿-date, ¿-date, ...
- personne - représentation : string x string x int ...,
- dictionnaire - représentation : liste de couples clé-valeur - interface : put(key,value), get(key,value)
- arbre binaire (voir plus loin)

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

Interface

Interface :

```
make-rat denom numer rat -rat *rat /rat =rat
```

On peut distinguer dans l'interface la partie *créationnelle*, permettant de créer des donées du type, de la partie *fonctionnelle* permettant d'utiliser les données créées.

La programmation par objet a généralisé la création de nouveaux types de données. L'interface créationnelle est constituée de la fonction `new` paramétrée par un ensemble de fonctions d'initialisation (généralement nommées *constructeurs*).

Implantation de l'interface fonctionnelle

Implanter une interface signifie définir les fonctions qui la constituent.

```
(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))))
```

Représentation No 1 : doublets, sans réduction

```
(define x (cons 1 2))
(car x) = 1
(cdr x) = 2

(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
------- doublets : cons, car, cdr
représentation des doublets
------- ???
```

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))
```

15.3 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.

Arborescence (arbre enraciné) arbre ayant un sommet (**racine**) et une orientation (descendant ou fils, ascendant ou parent)

Arbre binaire :

- soit un arbre vide

- soit un noeud (racine) et deux sous-arbres binaires gauche (fg) et droit (fd).

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.

15.4 Arbres binaires de recherche en Scheme

- Création et accès :
  (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)
  - (afficher a)

15.4.1 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))
```

15.4.2 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)))))
```

15.4.3 Recherche dans un arbre binaire

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

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

15.4.4 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)))
```

15.5 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.

15.5.1 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))
```

15.5.2 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))))))
```