Cours Structures de Données Avancées

Option G.L.
Semestre 3

Chapitre 2 : Structures de données linéaires

1. Les types abstraits de données

Un Type Abstrait de Données (TAD) est une notation pour décrire les types de données utilisés dans un algorithme, ainsi que l'ensemble des opérations qu'on peut leur appliquer et les propriétés de ces opérations. La conception de l'algorithme se fait en utilisant les opérations du TAD. Donc, un type de données n'est pas seulement un ensemble de valeurs mais un ensemble de valeurs muni d'un ensemble d'opérations (de même qu'en mathématiques un groupe est un ensemble muni d'une opération, un anneau est un ensemble muni de deux opérations, etc.).
On peut même aller plus loin en disant qu'il suffit pour décrire un type de données, de décrire très précisément ses opérations, sans donner explicitement l'ensemble de ses valeurs. On parle dans ce cas de type abstrait. L'avantage de la définition abstraite est qu'elle se concentre sur ce qu'on peut faire avec des objets de ce type, indépendamment de la manière dont ces objets sont représentés. Par exemple, dans la définition abstraite d'un type date on n'a pas besoin de spécifier si les dates sont représentées par des entiers, des triplets (jour, mois, année) ou des chaînes de caractères. Ce n'est qu'au moment de l'implémentation concrète que l'on choisira une représentation en fonction de critères tels que la performance, l'économie de place mémoire, la simplicité de programmation, etc.

1.1 Définition d'un TAD

Un type abstrait de données est donc caractérisé par :

- Son nom,
- Les sortes qu'il manipule,
- Les opérations sur les données,
- Les propriétés de ces opérations.

Les trois premières notions définissent la *signature* d'un TAD, les propriétés sont exprimées généralement sous forme d'*axiomes* dans un TAD.

Signature:
La signature d'un type abstrait est la donnée de son nom, les sortes qu'il utilise et les différentes opérations sur les données.
S**ortes:**
Les sortes ne sont rien d'autre que des noms servant à représenter des ensembles de valeurs sur lesquels vont porter les opérations. Par exemple naturel, booleen, entier, etc.
Profil d'une opération:
Chaque opération est définie par son profil : les sortes de ses paramètres et la sorte du résultat. La forme générale du profil d'une opérations n-aire est:

$$\text{Nom-opération} : \text{sorte}_1, \text{sorte}_2, ..., \text{sorte}_n \rightarrow \text{sorte}_r.$$

Remarques:

1. Pour éviter les notations trop lourdes des opérations (préfixée: trop d'imbrications de parenthéses) nous indiquons à la place des arguments des tirets, nous retrouvons donc les notations habituelles mixfixée, postfixée. Par exemple: i + j (pour éviter + (i, j)).
2. Une opération qui n'a pas d'argument est une opération constante.

1.2 Exemples de signatures

Les signatures suivantes sont extraites des TADs BOOLEEN et NATUREL:

TAD BOOLEEN
Sorte booléen
Opérations
vrai : → booléen
faux : → booléen
_ et _ : booléen, booléen → booléen
_ ou _ : booléen, booléen → booléen

TAD NATUREL
Sorte nat
Opérations
0 : → nat
succ : nat → nat
_ + _ : nat , nat → nat
_ - _ : nat , nat → nat
_ * _ : nat , nat → nat
_ ^_ : nat , nat → nat

1.3 Réutilisation des TADs

Il serait peu pratique d'avoir à donner toute la signature et toutes les propriétés des entiers (par exemple) quand on définit les vecteurs (où les entiers servent à numéroter les éléments). On se donne donc la possibilité quand on définit un type de réutiliser les types déjà définis.
Dans ce cas, la signature du type vecteur est l'union des signatures des types utilisés, enrichie de nouveaux noms de sortes et d'opérations. On introduit par cette construction une hiérarchie entre les types qui permet d'introduire une classification importante entre les sortes et les opérations d'un type de données.
Dans une signature, on appelle *sorte(s) définie(s)* la ou les sortes correspondant aux noms de sortes nouveaux.
On appelle *sorte(s) prédéfinie(s)* la ou les sortes provenant des TADs utilisés.
Une opération est dite *interne constructeur* si elle rend un résultat d'une sorte définie. Ces opérations servent à construire les valeurs d'une sorte. Par exemple, l'opération "succ" dans le TAD NATUREL, permet de construire tous les entiers.
Une opération est dite *interne non constructeur* si elle rend un résultat d'une sorte définie mais ne construit pas de nouvelles valeurs pour cette sorte. Par exemple, l'opération "_+_" dans le TAD NATUREL.
Enfin, une opération est dite *observateur* sur une sorte définie si elle rend un résultat d'une sorte prédéfinie et possède au moins un argument de sorte définie. Par exemple, l'opération "_=_" dans le TAD NATUREL1 ci dessous,

TAD NATUREL1
Utilise BOOLEEN
Sorte nat
Opérations
0 : → nat
succ : nat → nat
_ + _ : nat , nat → nat
_ - _ : nat , nat → nat
_ * _ : nat , nat → nat
_ ^_ : nat , nat → nat

Internes constructeurs

Internes non constructeurs

observateur

_=_: nat, nat → booléen

1.4 Les axiomes

Le problème est de donner une signification (une sémantique) aux noms de la signature, c.à.d. aux sortes et opérations. Si on veut définir un TAD indépendamment de ses implémentations possibles, la méthode la plus connue consiste à énoncer les propriétés des opérations sous forme d'axiomes. La forme générale de l'écriture des axiomes est:

$$terme_g \equiv terme_d$$

Le symbole "≡" doit se lire comme est "équivalent à", il n'a pas de direction privilégiée et signifie qu'on peut remplacer ce qui est à gauche par ce qui est à droite et réciproquement.
Cette forme peut s'étendre à l'expression des axiomes conditionnels comme suit:

$$terme_g \equiv \text{ si condition alors } terme_{d1} \text{ sinon } terme_{d2}$$

Choix d'écriture des axiomes

Généralement, il faut écrire des axiomes pour définir le résultat de la composition des opérations observateurs et internes non constructeurs avec toutes les opérations internes constructeurs.
Pour cela, le nombre d'axiomes nécessaires pour définir les propriétés d'une telle opération dépend de son nombre d'arguments et les opérations constructeurs associées aux sortes de ces arguments.
Dans le cas ou certaines opérations constructuers restent indéfinies, il faut écrire des axiomes pour les définir au moyen d'autres opérations internes.

Exemple

Si on veut complèter le TAD NATUREL par des axiomes, Il faut écrire pour chaque opération observateur ou interne non constructeur autant d'axiomes que d'opérations internes constructeurs: 0 et succ, combinées avec les arguments associés, c.-à-d. quatre axiomes pour l'opération _+_.
0 + 0 ≡ 0;
0 + succ(Y) ≡ succ(Y);
succ(X) + 0 ≡ succ(X);
succ(X) + succ(Y) ≡ succ(X + succ(Y));
qui sont combinés deux à deux pour donner les axiomes 1 et 2 du TAD en question suivant.

TAD NATUREL
Sorte Nat
Sorte nat
Opérations
0 : →nat;
succ : nat →nat;
_+_ : nat, nat →nat;
_-_ : nat, nat →nat;
_*_ : nat, nat →nat;
_^_ : nat, nat →nat;
Axiomes
1- X + 0 ≡ X;
2- X + succ(Y) ≡ succ(X + Y);
3- 0 - X ≡ 0;
— convention pour éviter de sortir des entiers naturels—
4- X - 0 ≡ X;
5- succ(X) - succ(Y) ≡ X - Y;
6- X * 0 ≡ 0;
7- X * succ(Y) ≡ X + (X * Y);
8- X ^0 ≡ succ(0);
9- X ^succ(Y) ≡ X * (X ^Y);
Variables
X, Y : nat;

Remarque:
Ils existent des types de données où certaines opérations sont des fonctions partielles, non définies partout. Les préconditions sont destinées à restreindre l'utilisation des opérations à des opérandes appartenant au domaine de définition de cette opération. La syntaxe des préconditions est :

pré(nom_opération(arguments)) définie ssi < condition >

Lorsqu'une opération n'a pas de précondition, le domaine de définition
de cette opération est toute la sorte.

Lorsqu'on écrit des axiomes d'un TAD, on est amené a se poser les questions suivantes:

1. N'y a t-il pas d'axiomes contradictoires (consistance) ?
2. A t-on donné un nombre suffisant d'axiomes pour décrire toutes les propriétés du TAD (complétude) ?

1.5 Implémentation d'un TAD

A l'issue d'une conception descendante on veut obtenir l'algorithme dans un langage de programmation. Dans ce cours, il s'agit du langage JAVA. Il faut donc établir une correspondence entre les types abstraits utilisés pour concevoir l'algorithme, et les types (élémentaires ou objets) du langage JAVA. La réalisation d'un TAD consiste donc à:

• Pour chaque sorte définie, choisir une structure interne de données construite à partir des types simples (primitifs) ou complexes (objets) du langage et/ou des types déjà définis.
• La programmation des opérations internes, en tenant compte de la structure choisie, par des fonctions du langage de programmation (méthodes de classes dans le cas de JAVA).
• La programmation des opérations observateurs ou internes non constructeurs par des fonctions du langage de programmation (méthodes de classes dans le cas de JAVA) en utilisant leurs axiomes.

Exemple:
On se propose d'abord d'écrire le TAD DISQUE décrivant la structure de donnée disque formée à partir des coordonnées du centre et de son rayon. Prévoir l'opération **Surface** qui calcule la surface d'un disque. Ensuite, on propose une implémentation pour ce TAD en JAVA.

TAD DISQUE
Utilise REEL
Sortes Disque
Opérations :
nouveau_disque : Réel, Réel, Réel → Disque
x_du_centre : Disque→ Réel
y_du_centre : Disque→ Réel
rayon : Disque → Réel
surface : Disque→ Reel
Variables :
u,v,r : Réels ; d : disque
Préconditions :
Pré(nouveau-disque(u,v,r)) définie ssi r > 0
Axiomes :
Rayon(nouveau _disque(u, v, r))= r
x_ du_centre(nouveau _disque(u, v, r)) = u
y_du_centre(nouveau _disque(u, v, r)) = v
surface(d) = rayon(d) * rayon(d) * 3.14

Une implémentation possible en JAVA :

```
public class Disque {
        private double  rayon ;
        private double  x_du_centre ;
        private double  y_du_centre ;
        public Disque()
         {
                System.out.println("Création d'un disque !");
                rayon = 0;
                x_du_centre  = 0;
                y_du_centre  = 0;
         }

        public Disque(double pRayon, double pXcentre, double pYcentre)
         {
                System.out.println("Création d'un disque avec des paramètres !");
                rayon = pRayon;
                x_du_centre  = pXcentre;
                y_du_centre  = pYcentre;
         }

        public double rayon()
         {
                return rayon;
         }
        public double  x_du_centre ()
        {
          Return x_du_centre;
        }
        public  double  y_du_centre ()
        {
          return y_du_centre;
        }
        public double  surface ()
        {
          return rayon* rayon * 3.14  ;
         }
```