Service Informatique et Mathématiques Appliquées

Description des structures de données

B. MEYER *

1. – INTRODUCTION

La nécessité de séparer, dans la description d'une structure de données, les propriétés abstraites des objets et leur représentation concrète, est reconnue depuis longtemps sur le plan des idées. Dans [7], par exemple, d'Imperio faisait déjà la distinction entre "structures de données" et "structures de mémoire". En pratique, cependant, cette distinction reste souvent un voeu pieux, et la manipulation concrète des structures de données mélange caractéristiques logiques et problèmes de représentation physique.

Il est souvent important, pourtant, de dégager les propriétés "intrinsèques" d'une structure de données des caractéristiques de sa représentation. Dans un traité récent sur la conception et l'analyse des algorithmes [2], les "files" (*queues*) sont introduites initialement (§ 2.1) par le biais d'une représentation formée d'un tableau et de deux pointeurs. Or, le premier algorithme qui utilise des files (figure 3.1 de [2]) effectue l'opération "concaténation de 2 files" dont l'implémentation efficace requiert une représentation chaînée (bien que l'utilisation d'une représentation contiguë ne change pas la complexité globale de l'algorithme en question, qui est $0((m + n)\,k)$). Il semble donc qu'on doive utiliser une notion "abstraite" de file, préexistente à toute représentation physique.

Le but du présent article est de proposer, dans la lignée des travaux de Liskov et Zilles [8] [9], une définition des structures de données reposant sur trois niveaux de description que nous appellerons *spécification fonctionnelle, description logique,* et *représentation physique,* et de montrer la nécessité de cette distinction en l'appliquant à la définition de quelques structures classiques.

NOTATIONS

Nous aurons à manipuler des fonctions bivaluées, de la forme :

$$f : A \rightarrow B \times C$$

Pour $x \in A$, nous noterons $f_B^{(x)}$ et $f_C^{(x)}$ les projections de $f(x)$ sur B et C respectivement.

Soit une fonction f de la forme

$$f : B \times A \rightarrow A$$

pour $x \in A$, et $y_1, y_2, \ldots, y_n \in B$, nous dénoterons par

$$f^{(n)}(y_1, y_2, \ldots, y_n ; x)$$

la valeur de $\quad f(y_1, f(y_2, f(y_3, \ldots, f(y_n, x)\ldots)))$

(*) Ingénieur-Chercheur au Département Méthodes et Moyens de l'Informatique.

E.D.F. – BULLETIN DE LA DIRECTION DES ÉTUDES ET RECHERCHES
SÉRIE C – MATHÉMATIQUES, INFORMATIQUE N° 2 1976 pp. 81-90

2. – SPECIFICATION FONCTIONNELLE

Une "structure de données" est un objet composé vérifiant certaines propriétés. La description d'une telle structure est la définition du domaine des objets vérifiant ces propriétés, c'est-à-dire la définition d'un nouveau *type* de données.

On appellera *spécification fonctionnelle* d'une structure de données la définition externe de ses propriétés, c'est-à-dire l'ensemble des caractéristiques qu'elle doit présenter pour qu'un programme puisse l'utiliser. La spécification fonctionnelle donne une description logiquement complète du domaine D considéré, mais n'implique aucune décision quant à l'organisation interne des objets appartenant à ce domaine.

Une spécification fonctionnelle est formée de :

a) un certain nombre de fonctions $f_i (i = 1, 2, \ldots, n)$:

$$f_i : X_{i_1} \times X_{i_2} \times \ldots \times X_{i_p} \rightarrow X_{i_{p+1}} \times \ldots \times X_{i_q}$$

où l'un au moins des X_{i_j} est le domaine D que l'on cherche à définir, et

b) un certain nombre de relations faisant intervenir ces fonctions et des fonctions définies sur les domaines $X_{i_1} \ldots X_{i_q}$ autres que D. Ces relations seront en général des formules du calcul des prédicats du premier ordre, avec égalité ; elles servent à définir les propriétés abstraites que doivent vérifier les fonctions définies sur D.

On distinguera en pratique trois sortes de fonctions

$$f_i : X_{i_1} \times X_{i_2} \times \ldots \times X_{i_p} \rightarrow X_{i_{p+1}} \times \ldots \times X_{i_q}$$

a) fonctions de création : D n'apparaît qu'à droite de la flèche. La fonction peut ne pas avoir d'arguments ($p = 0$).

b) fonctions d'accès : D n'apparaît qu'à gauche de la flèche.

c) fonctions de modification : D apparaît à gauche et à droite.

3. – DESCRIPTION LOGIQUE

Le niveau suivant de décomposition, que nous appelons *description logique,* peut être pour le programmeur le niveau terminal s'il dispose d'un langage de programmation évolué (ALGOL W, SIMULA 67, PASCAL, ALGOL 68 . . .) ; dans le cas contraire (programmation en langage machine, FORTRAN etc.) un niveau d'affinage supplémentaire sera nécessaire.

La description logique consiste à fournir une décomposition (éventuellement récursive) des objets du domaine en objets d'autres domaines, et une décomposition (éventuellement récursive) des fonctions intervenant dans les spécifications fonctionnelles en termes d'autres fonctions. Ces décompositions doivent évidemment satisfaire les axiomes de la spécification fonctionnelle.

La description logique fait intervenir un certain nombre de types de base et d'opérations de composition. Plusieurs choix sont possibles pour cet ensemble d'éléments de base :

1) Un premier exemple, proche de ce que proposent des langages comme ALGOL W, PASCAL, SIMULA, ALGOL 68, etc. consiste à prendre un certain nombre de types connus (entiers, réels, caractères. . .), et les trois opérations de composition suivantes :

a) **juxtaposition** (produit cartésien) : t_1 et t_2 étant des types, on notera :

$$t_1 \; ; \; t_2$$

le type dont les éléments sont des couples d'objets de types respectifs t_1 et t_2.

b) **union** : on notera

$$t_1 \mid t_2$$

le type dont les éléments sont, soit de type t_1, soit de type t_2.

c) **énumération** :

$$\alpha_1 , \alpha_2 , \ldots , \alpha_n$$

dénotera le type dont les éléments peuvent être égaux à l'une des constantes

$$\alpha_i , i = 1 , \ldots , n$$

Du point de vue de la notation, on pourra utiliser des parenthèses pour lever les ambiguïtés dans la description de types complexes, et étiqueter les différents composants dans une juxtaposition. Exemples :

type *DATE = an : ENTIER ; mois : ("janvier", . . . , "décembre") ; jour : ENTIER*

type *PERSONNE = nom : TEXTE ; âge : (DATE|ENTIER) ; sexe : LOGIQUE*

Les définitions peuvent être récursives. On utilisera en général dans ce cas une "union" avec un type spécial appelé *VIDE* (contenant un seul élément que nous noterons également *VIDE* par abus de langage) :

type *LISTE D'ENTIERS = VIDE | (premier : ENTIER ; suite : LISTE D'ENTIERS)*

type *ARBRE BINAIRE = VIDE | (racine : INFO ; gauche : ARBRE BINAIRE ; droite : ARBRE BINAIRE)*

Notons à cet égard qu'il est prématuré d'introduire à ce niveau la notion de pointeur [6], de même qu'il serait prématuré d'introduire la notion de branchement dans la description logique d'un algorithme.

2) On peut réduire le nombre d'éléments de base en se limitant, comme [3], à un ensemble de départ contenant :

– un seul type de base (qui est la juxtaposition de 0 types) ;

– 2 opérations de composition : l'union et la juxtaposition.

Avec une interprétation adéquate, on peut alors retrouver de façon simple tous les types usuels (booléens, entiers, . . .) et complexes (listes, arbres. . .).

3) Un troisième ensemble d'éléments de base d'une description logique pourrait être emprunté aux méthodes de descriptions des bases de données, en particulier à un modèle relationnel : modèle de Codd [4] ou modèle binaire d'Abrial [1].

4) A titre de dernier exemple d'outils de description logique, mentionnons l'approche de Gedanken [13], où les structures de données sont décrites comme des fonctions. Les opérations de base sont alors la composition de fonctions et l'expression conditionnelle.

4. – REPRESENTATION PHYSIQUE

La représentation physique d'une structure de données est l'implantation concrète, en machine, d'objets et de sous-programmes conformes à une description logique. Les descriptions logiques étant en général récursives, on peut considérer la représentation physique comme une sorte de "point fixe" de la description logique. Son élaboration sera, selon le langage de programmation utilisé, à la charge du compilateur ou à celle du programmeur.

Nous ne développerons pas dans cet article les problèmes de la représentation physique, où interviennent des méthodes telles que l'utilisation des pointeurs (pour la juxtaposition récursive), de "drapeaux" (pour l'union), les algorithmes de "ramasse-miettes" et de gestion de la mémoire etc. Signalons simplement un point important : l'approche "fonctionnelle" utilisée jusqu'ici n'interdit

en aucune façon de représenter certaines des **fonctions de manipulation** par des "procédures non typées" (*SUBROUTINE*) opérant par effet de bord sur certains arguments, si cela conduit à une implantation plus efficace ; on exigera simplement que les relations intervenant dans la spécification fonctionnelle soient vérifiées entre les valeurs initiale et finale de l'argument correspondant.

5. – UN EXEMPLE

Soit à définir le type "compte en banque", ou *COMPTE*, et le type "ensemble de comptes en banque", ou *BANQUE*. Nous supposons connus les types de base *ENTIER*, *ENTIER POSITIF*, *TEXTE* (= chaîne de caractères), *LOGIQUE* (= booléen), et le type *PERSONNE* (qui peut être un type simple ou complexe). Une spécification fonctionnelle possible pour *COMPTE* et *BANQUE* comprend :

fonctions de création

$$\textit{créer-banque} : \rightarrow \textit{BANQUE}$$
$$\textit{créer-compte} : \textit{PERSONNE} \times \textit{ENTIER POSITIF} \rightarrow \textit{COMPTE}$$

fonctions d'accès

$$\textit{titulaire} : \textit{COMPTE} \rightarrow \textit{PERSONNE}$$
$$\textit{solde} : \textit{COMPTE} \rightarrow \textit{ENTIER}$$
$$\textit{trouver-compte} : \textit{BANQUE} \times \textit{PERSONNE} \rightarrow \textit{COMPTE} \cup \{\textit{VIDE}\}$$

fonctions de modification

$$\textit{inscrire-compte} : \textit{BANQUE} \times \textit{COMPTE} \rightarrow \textit{BANQUE}$$
$$\textit{ajouter} : \textit{COMPTE} \times \textit{ENTIER POSITIF} \rightarrow \textit{COMPTE}$$
$$\textit{retirer} : \textit{COMPTE} \times \textit{ENTIER POSITIF} \rightarrow \textit{COMPTE}$$

relations

$\forall b \in \textit{BANQUE}, \forall c, c' \in \textit{COMPTE}, \forall n, n' \in \textit{ENTIER POSITIF}, \forall p, p' \in \textit{PERSONNE}$:

(C_1) $\textit{titulaire (créer-compte } (p, n)) = p$

(C_2) $\textit{solde (créer-compte } (p, n)) = n$

(C_3) $c = c' \Leftrightarrow \textit{titulaire } (c) = \textit{titulaire } (c')$

(C_4) $\textit{trouver-compte (créer-banque, } p) = \textit{VIDE}$

(C_5) $\textit{trouver-compte (inscrire-compte (b, créer-compte } (p, n)), p) = \textit{créer-compte } (p, n)$

(C_6) $\textit{inscrire-compte (inscrire-compte } (b, c), c') = \textit{inscrire-compte (inscrire-compte } (b, c'), c)$

(C_7) $\textit{solde (ajouter } (c, n)) = \textit{solde } (c) + n$

(C_8) $\textit{solde (retirer } (c, n)) = \textit{solde } (c) - n$

(C_9) $\textit{titulaire (ajouter } (c, n)) = \textit{titulaire } (c)$

(C_{10}) $\textit{titulaire (retirer } (c, n)) = \textit{titulaire } (c)$

De ces propriétés, on peut déduire un grand nombre d'autres. Par exemple :

(C_1) et (C_3) entraînent

$$(C_{11})\ p \neq p' \Rightarrow \textit{créer-compte } (p, n) \neq \textit{créer-compte } (p', n')$$

de même, on peut montrer à partir de $(C_1), (C_3)$, (C_4), (C_5) et (C_6) que

$$(C_{12})\ p \neq p' \Rightarrow \textit{trouver-compte } (b, p) = \textit{trouver-compte (inscrire-compte } (b, \textit{créer-compte } (p', n')), p)$$

à condition toutefois de supposer que les seules opérations effectuées sur les *COMPTES* et les *BANQUES* sont celles qui entrent dans la définition fonctionnelle (démonstration par récurrence sur le nombre d'opérations *inscrire-compte* effectuées).

Une relation du type (C_3) indique que l'un des "composants" d'un type (ici le titulaire) est une "clé primaire" [4].

Une description logique des types *COMPTE* et *BANQUE* (conduisant sans doute à une représentation physique peu efficace) pourrait être :

type *BANQUE = VIDE | (premier : COMPTE ; suite : BANQUE)*

type *COMPTE = possesseur : (PERSONNE | SOCIETE); numéro : ENTIER POSITIF ; solde : ENTIER*

type *PERSONNE = PERSONNE PHYSIQUE | SOCIETE*

type *PERSONNE PHYSIQUE = (nom : TEXTE ; adresse : TEXTE)*

type *SOCIETE = nom : TEXTE ; siège-social : TEXTE ; capital : ENTIER POSITIF*

avec pour les fonctions associées des descriptions logiques de la forme :

sous-programme *trouver-compte (p : PERSONNE ; b : BANQUE) : COMPTE*

```
si   b = VIDE alors
                VIDE
sinon si titulaire (premier [b]) = p alors
                premier [b]
sinon    trouver-compte (p , suite [b])
```

Notons qu'il s'agit bien là d'une description logique, et que la représentation physique ne fera pas nécessairement intervenir des sous-programmes récursifs. Un autre point important est qu'à une spécification fonctionnelle donnée peuvent correspondre plusieurs descriptions logiques, et réciproquement ; ainsi, une même structure logique de "banque" pourra être "vue" fonctionnellement de diverses façons pour des raisons de protection (certaines opérations étant absentes de certaines spécifications). De la même manière, une structure logique peut avoir plusieurs représentations physiques adéquates.

6. – SPECIFICATION FONCTIONNELLE DE QUELQUES STRUCTURES CLASSIQUES

Nous allons préciser la notion de spécification fonctionnelle à propos de quelques structures usuelles : pile, listes, tableaux, ensembles, etc. La simplicité de ces spécifications, qui permettent cependant de démontrer toutes les propriétés habituelles des structures considérées, nous paraît un argument important en faveur de l'utilisation de spécifications fonctionnelles avant toute description détaillée de l'organisation des données d'un programme.

Dans ce qui suit, nous considérons des structures complexes bâties à partir d'un type de base que nous appellerons T : des piles d'objets de type T, des files d'objets de type T, etc. Le type T est quelconque (ce pourrait être *ENTIER, TEXTE* etc. ou un type complexe, comme *PILE D'ENTIERS* etc.) ; nous le considèrerons cependant comme connu, c'est-à-dire que nous n'aborderons pas le problème de la définition de types complexes paramétrés.

Nous considèrerons d'abord deux variétés de structures que l'on peut appeler "structures séquentielles à lecture destructrice" : les *piles* ou "structures à une seule tête de lecture/écriture", et les *files*, à "une tête de lecture et une tête d'écriture" ; nous étudierons ensuite les *tableaux*, qui sont à "lecture non destructrice, accès direct, et une seule tête de lecture/écriture".

6.1. – Piles.

Une pile a la spécification fonctionnelle suivante :

fonctions

$$\begin{aligned} &\textit{créer-pile} : \rightarrow \textit{PILE} && \text{(création)}\\ &\textit{vide} : \textit{PILE} \rightarrow \textit{LOGIQUE} && \text{(accès)}\\ &\textit{dépiler} : \textit{PILE} \rightarrow T \times \textit{PILE} && \text{(modification)}\\ &\textit{empiler} : T \times \textit{PILE} \rightarrow \textit{PILE} && \text{(modification)} \end{aligned}$$

relations

$\forall t \in T,\ \forall p \in PILE :$

(P_1) $\quad \textit{vide (créer-pile)}$

(P_2) $\quad \sim \textit{vide (empiler (p , t))}$

(P_3) $\quad \textit{dépiler (empiler (t , p))} = [t , p]$

Ces trois axiomes permettent de retrouver toutes les propriétés habituelles des piles. La plus importante de ces propriétés peut s'énoncer de façon très intuitive, "on retrouve les éléments dans l'ordre inverse de celui où on les a mis", ou encore

$$\forall p \in PILE,\ \forall t_1, t_2, \ldots, t_n \in T :$$
$$0 \leqslant m < n \Rightarrow \textit{dépiler}_T(\textit{dépiler}^{(m)}_{PILE}(\textit{empiler}^{(n)}(t_n , t_{n-1} , \ldots , t_1 ; p)) = t_{n-m}$$

C'est-à-dire que si l'on dépile $m + 1$ éléments d'une pile sur laquelle on a auparavant empilé n éléments $t_1, t_2, \ldots, t_n$, $(0 \leqslant m < n)$, le dernier élément dépilé est t_{n-m}. Cette propriété se démontre immédiatement à partir de (P_3) par récurrence sur m.

Plus généralement, on démontre facilement à partir de (P_3) la propriété suivante : soit

$$X \equiv a_n(a_{n-1}(\ldots a_1(p)\ldots))$$

une expression où chaque fonction a_i est :

$$\begin{cases} \text{soit } \textit{dépiler}_{PILE} \\ \text{soit la fonction associant à toute pile } y \text{ la pile } \textit{empiler}(t_i, y)\ (t_i \in T) \end{cases}$$

et telle que pour i, $1 \leqslant i \leqslant n$, il y ait parmi $a_i, a_{i+1}, \ldots, a_n$ au moins autant d'opérations *empiler* que d'opérations *dépiler*.

Alors, on obtient une expression égale à X en rayant successivement, à partir de la droite, chaque opération *dépiler* et l'opération *empiler* non encore rayée qui se trouve (par construction) immédiatement à sa droite. Exemple :

$$\textit{empiler}\ (t_6, \textit{dépiler}_{PILE}(\textit{empiler}(t_4, \textit{empiler}(t_3, \textit{dépiler}_{PILE}(\textit{empiler}(t_1 , p)))))) =$$
$$= \textit{empiler}(t_6 , \textit{empiler}(t_3 , p))$$

Les axiomes (P_1), (P_2), (P_3) n'interdisent pas expressément de *dépiler* une pile vide, mais empêchent de connaître quelque propriété que ce soit sur l'élément "dépilé" d'une pile vide. Une pile vide dépilée reste cependant une pile légale, sur laquelle on pourra ensuite *empiler* des éléments de T en leur appliquant l'axiome (P_3). Si l'on veut préciser au niveau de la spécification fonctionnelle que le dépilage d'une pile vide est une erreur et rend la pile inutilisable, on associera à chaque type X un élément spécial "indéfini" noté ω_X ; on ajoutera alors l'axiome suivant :

(P_4) $\quad \textit{vide (p)} \Rightarrow \textit{dépiler (p)} = [\omega_T , \omega_{PILE}]$

et l'on spécifiera que les fonctions caractéristiques de la structure sont "strictement monotones", ou encore [11, p. 359] des "extensions naturelles" au sens de la théorie du point fixe, c'est-à-dire qu'elles valent ω si l'un au moins de leurs arguments est ω :

(P_5) $vide\,(\omega_{PILE}) = \omega_{LOGIQUE}$

(P_6) $dépiler\,(\omega_{PILE}) = [\omega_T, \omega_{PILE}]$

(P_7) $empiler\,(t, \omega_{PILE}) = \omega_{PILE}$

(P_8) $empiler\,(\omega_T, p) = \omega_{PILE}$

La plupart des descriptions de piles publiées font intervenir la notion de "taille" d'une pile. Il s'agit là à notre sens d'un concept qui ne doit pas intervenir au niveau d'une spécification fonctionnelle : il est important dans une implantation contigue (tableau et pointeur), mais beaucoup moins dans le cas d'une implantation chaînée. De fait, si toute implantation oblige à prendre en compte une taille maximale, on peut considérer qu'elle ne fait pas partie du type **abstrait** *PILE*. Ceci dit, la notion de taille s'introduit toutefois facilement dans le formalisme ci-dessus : une pile de taille n est caractérisée par l'axiome supplémentaire :

(P_9) $\forall\, m > n,\ empiler^{(m)}_{PILE}\,(p\,;\, t_1, t_2, \ldots, t_m) = \omega_{PILE}$

ou simplement, si l'on suppose que les fonctions de base sont des "extensions naturelles" (axiomes (P_5) à (P_8)) :

(P'_9) $empiler^{(n+1)}\,(t_{n+1}, t_n, \ldots, t_2, t_1\,;\, créer\text{-}pile) = \omega_{PILE}$

6.2. – **Files.**

La spécification fonctionnelle d'une *file* (anglais *queue*) est donnée par

fonctions

créer-file :	$\rightarrow$ *FILE*	(création)
vide :	*FILE* $\rightarrow$ *LOGIQUE*	(accès)
défiler :	*FILE* $\rightarrow T \times$ *FILE*	(modification)
enfiler :	$T \times$ *FILE* $\rightarrow$ *FILE*	(modification)

relations

$\forall\, t \in T, \forall\, f \in F$:

(F_1) $vide\,(créer\text{-}file)$

(F_2) $\sim vide\,(enfiler\,(t, f))$

(F_3) $\sim vide\,(f) \Rightarrow défiler\,(enfiler\,(t, f)) = [défiler_T\,(f), enfiler\,(t, défiler_{FILE}\,(f))]$

(F_4) $vide\,(f) \Rightarrow défiler\,(enfiler\,(t, f)) = [t, f]$

L'axiome (F_3) affirme la commutativité des opérations *défiler* et *enfiler* sur une file non vide (seul cas où cette propriété ait un sens) ; il est évident intuitivement puisque l'on "enfile à un bout" et que l'on défile à l'autre.

Il est intéressant de noter que nos définitions de piles et de files ne diffèrent que par le 3e axiome (P_3 , F_3), l'axiome correspondant à (F_4) étant vrai sur les piles. Les axiomes (F_1 – F_4) permettent de retrouver toutes les propriétés habituelles des files. La plus importante s'exprime intuitivement par le fait que "l'on retrouve les éléments dans l'ordre où on les a mis". Par exemple, on montre immédiatement par récurrence que pour $0 \leqslant m < n$:

$$vide\,(f) \Rightarrow défiler_T(défiler^{(m)}_{FILE}\,(enfiler^{(n)}\,(t_n, t_{n-1}, \ldots, t_1\,;f)) = t_{m+1}$$

Plus généralement, si f est une file vide, et si l'on considère l'expression

$$X = a_n(a_{n-1}(\ldots(a_1(f)\ldots))$$

ou chaque a_i est

{ soit $défiler_{FILE}$
{ soit la fonction qui associe à toute file y la file $enfiler\,(t_i, y)$ $(t_i \in T)$

et telle que pour tout i, $1 \leqslant i \leqslant n$, il y ait parmi a_i, a_{i+1}, . . . , a_n au moins autant d'*enfiler* que de *défiler*, alors on ne change pas la valeur de X en rayant toutes les opérations $défiler_{FILE}$, et autant d'opérations *enfiler* **à partir de la droite.** Exemple :

si *vide (f)* alors
enfiler (t₆ , défiler$_{FILE}$ (enfiler (t₄ , enfiler (t₃, défiler$_{FILE}$ (enfiler (t₁ , f)))))) =

= enfiler (t₆ , enfiler (t₄ , f))

Si f n'est pas vide, on peut d'après (F_2) et (F_3) ramener toutes les opérations $défiler_{FILE}$ à gauche dans l'expression X. Ainsi, l'expression de l'exemple ci-dessus est égale pour tout f à

défiler$_{FILE}$ (défiler$_{FILE}$ (enfiler (t₆ , enfiler (t₄ , enfiler (t₃ , enfiler (t₁ , f))))))

Comme pour une pile, on peut spécifier qu'il est illégal de *défiler* une file vide par des axiomes semblables à (P_4) et (P_5 – P_8) ("extension naturelle"). On peut de même préciser qu'une file est de "taille" n par un axiome transposé de (P_9) ou (P'_9).

6.3. – Tableaux.

Par opposition aux structures précédentes, un tableau est une structure à *accès direct* indexé.

La spécification fonctionnelle d'un tableau à une dimension de bornes m et n est donnée ci-dessous (on note $[m : n]$ l'ensemble $\{i \mid i \in \mathbf{N},\ m \leqslant i \leqslant n\}$, et TABLEAU $[m : n]$ l'ensemble des tableaux de bornes m et n) :

fonctions

créer-tableau : → *TABLEAU* $[m : n]$	(création)
accéder-élément : $[m : n]$ x *TABLEAU* $[m : n]$ → T	(accès)
modifier-élément : T x $[m : n]$ x *TABLEAU* $[m : n]$ → *TABLEAU* $[m : n]$	(modification)

relations

$\forall t, t' \in T, \forall tab \in TABLEAU\,[m : n], \forall i, j \in [m : n]$:

(T_1) *accéder-élément (i, modifier-élément (t, i, tab)) = t*

(T_2) $i \neq j \Rightarrow$ *modifier-élément (t, i, modifier-élément (t', j, tab))*
= modifier-élément (t', j, modifier-élément (t, i, tab))

(T_3) *modifier-élément (t, i, modifier-élément (t', i, tab)) = modifier-élément (t, i, tab)*

Notons que le langage LISP 1.5 [10] introduit précisément les tableaux à l'aide de deux fonctions distinctes, accès et modification.

On pourrait aussi considérer que les fonctions *accéder-élément* et *modifier-élément* font intervenir le domaine N plutôt que $[m : n]$, avec la convention que le résultat est indéfini (ω_T *ou* $\omega_{tableau}$) si leur argument entier n'appartient pas à $[m : n]$.

6.4. – Ensembles.

Même au niveau de la spécification fonctionnelle, on peut définir les caractéristiques d'une structure de données de façon plus ou moins précise. Ainsi, piles, listes et tableaux (de même qu'un grand nombre d'autres structures, comme les *arbres binaires de recherche* dont la spécification fonctionnelle est facile à donner en supposant T muni d'une relation d'ordre) peuvent servir de réalisations particulières à une structure générale, celle d'*ensemble*, dont la spécification fonctionnelle est :

fonctions

créer-ensemble : → *ENSEMBLE*	(création)
membre : T x *ENSEMBLE* → *LOGIQUE*	(accès)
ajout : T x *ENSEMBLE* → *ENSEMBLE*	(modification)
retrait : T x *ENSEMBLE* → *ENSEMBLE*	(modification)

relations

$\forall e \in ENSEMBLE, \forall t, t' \in T$:

(E_1) *membre (t, créer-ensemble)* = **faux**

(E_2) *membre (t, ajout (t , e))* = **vrai**

(E_3) *membre (t, retrait (t , e))* = **faux**

(E_4) *ajout (t, ajout (t',e))* = *ajout (t', ajout (t , e))*

(E_5) $t \neq t' \Rightarrow$ *membre (t, retrait (t' , e))* = *membre (t , e)*

7. – CONCLUSION

Nous avons jusqu'ici négligé un problème lié aux structures de données : celui de la complexité des algorithmes qui les manipulent. On peut remarquer par exemple que les *files* peuvent servir à une description logique des *tableaux*, à l'aide de l'algorithme ci-dessous :

sous-programme *accéder (i : ENTIER ; tab : TABLEAU$_T$ [m : n]) : T*

{*tab est représenté par une file*}
variable *x* : *T* ;
si $i < m$ **ou** $i > n$ **alors**
ω_T
sinon | **pour** *j* **variant de** *m* **à** *i* **répéter**
[*x* , *tab*] ← *défiler (tab)* ;
enfiler (x , tab) ;
x {*résultat renvoyé par le sous-programme*}

et d'un algorithme similaire pour *modifier (t , i , tab)*. L'inconvénient est évidemment que cette décomposition conduit à une implantation physique de complexité *0(i)*, alors qu'on attend d'un tableau qu'il soit à "accès direct". Un des rôles de la spécification fonctionnelle est donc précisément de mettre en lumière les opérations fondamentales sur la structure, que l'on désire ensuite réaliser de la façon la plus efficace possible.

Pour ce problème, comme pour tous ceux qui se posent à propos des structures de données, la distinction préalable entre les trois niveaux de description dégagés dans cet article nous paraît une méthode de décomposition indispensable à la clarification des concepts fondamentaux.

BIBLIOGRAPHIE

[1] ABRIAL J.R. – *Data Semantics.* Université Scientifique et Médicale de Grenoble, Laboratoire d'Informatique, 1974.

[2] AHO A.V., HOPCROFT J.D. et ULLMANN. – *The Design and Analysis of Computer Algorithms.* Addison-Wesley, 1974.

[3] BURGE W. – *Recursive Programming Techniques.* Addison-Wesley, 1976.

[4] CODD E.F. – A Relational Model for Large Shared Data Banks ; *Communications of the ACM,* Vol. 13, n° 6, Juin 1970, pp. 377-387.

[5] HOARE C.A.R. – Notes on Data Structuring ; *in* DAHL, DIJKSTRA, HOARE. – *Structured Programming.* Academic Press, 1972.

[6] HOARE C.A.R. – *Recursive Data Structures.* Stanford University, rapport STAN-CS-73-400, Octobre 1973.

[7] d'IMPERIO M.E. – Data Structures and their Representation in Storage ; *Annual Review in Automatic Programming,* 1969, pages 1-75.

[8] LISKOV B. et ZILLES S. – Programming with Abstract Data Types ; *Sigplan Notices,* 9, 5, Mai 1974.

[9] LISKOV B. et ZILLES S. – Specification Techniques for Data Abstractions ; *IEEE Transactions on Software Engineering,* vol. 1 n° 1, pages 7-19.

[10] McCARTHY J., *et al. : LISP 1.5 Programmer's Manual.* The MIT Press, 1962.

[11] MANNA Z. – *The Mathematical Theory of Computation.* McGraw-Hill, 1975.

[12] PARNAS D.L. – A Technique for Software Module Specification with Examples ; *Communications of the ACM,* vol. 6 n° 1, mai 1972, pages 330-336.

[13] REYNOLDS J.C. – GEDANKEN, a Simple Typeless Language Based on the Principle of Completeness and the Reference Concept ; *Communications of the ACM,* vol. 13 n° 5, mai 1970, pages 308-318.