Structures de données : des types abstraits de données à leur implantation en langage C

Mathias Paulin*
Licence d'informatique - Université Paul Sabatier - Toulouse

*IRIT-STORM, Mathias.Paulin@irit.fr

1 Préambule

La programmation d'applications en informatique peut être abordée de différentes manières. Une approche fonctionnelle, dans laquelle la fonction est l'unité de base d'une application et qui repose sur la spécification, la vérification et l'écriture dans un langage de programmation de ces fonctions ou une approche fondée sur une abstraction des données, qui repose sur la spécification, la vérification et l'écriture dans un langage de programmation de structures de données et des opérations s'y afférant. La nuance entre ces deux approches est subtile et nous verrons que de très nombreux points communs se retrouvent dans ces deux manières d'aborder la programmation.

Parmi ces points communs, on retrouvera la notion de **spécification**, se concentrant sur le *quoi*, plutôt que sur le *comment*, la notion de **vérification**, permettant de s'assurer de la bonne traduction dans un langage de programmation des spécifications et reposant sur une axiomatique logique permettant de vérifier des propriétés précises sur les algorithmes ou les structures de données. Il s'agit alors de construire une **abstraction** du programme et de ses données et de raisonner sur cette abstraction, indépendante de tout langage de programmation.

On retrouvera aussi, la notion de *performance* de l'implantation, reposant sur une analyse de **complexité**, tant en temps qu'en espace, des traitements proposés mais aussi les notions de *réutilisabilité* d'un programme, ou plus précisément d'un ensemble de sous-programmes, entre différentes applications. Nous sommes alors confrontés à une approche **concrète** de la programmation reposant sur des choix de représentation des données et d'écriture des algorithmes en fonction du langage de programmation utilisé.

L'objectif de cette unité d'enseignement est de familiariser les étudiants avec une approche rigoureuse de la programmation, reposant sur un raisonnement formel logique, mais surtout, de les amener à maitriser la programmation d'applications traitant des données complexes. Pour cela, le contenu de cette UE est articulé autour de concepts généraux sur les types abstraits de données, sans en approfondir la théorie qui sera abordée dans l'UE *Types abstraits* de la troisième année de licence d'informatique, mais en mettant l'accent sur la nécessité de l'abstraction et la manière de l'utiliser, mais aussi, et surtout, sur une approche de la programmation en langage C permettant de vérifier à la fois des contraintes de performances, de réutilisabilité et de sûreté de fonctionnement des applications produites.

Calendrier prévisionnel

Séance 1 à 2 : Préambule (section 1) et Introduction aux Types Abstraits de Données (section 2)
Séances 2 à 4 : Implantation des TAD en C (section **??**)
Séances 5 à 9 : Types à structure linéaire (section **??**)
Séances 10 : Parcours d'une collection et itérateurs (section **??**)
Séances 11 à 14 Types à structure arborescente (section **??**)
Séances 15 Tables de hachage (section **??**)

2 Introduction aux Types Abstraits de Données

2.1 Concepts clés et compétences transmises

Concepts présentés dans cette partie

- Définition des types abstraits de données
- Signature d'un type abstrait de données
- Axiomatique d'un type abstrait de données
- Notion de complétude et de consistance

Compétences acquises dans cette partie

- Savoir identifier les données complexes manipulées par un programme
- Savoir spécifier, sans preuve formelle de complétude ou de consistance, un type abstrait de données
- Savoir utiliser un type abstrait de données pour résoudre un problème

2.2 Méthodologie de programmation

Dans l'UE algorithmique et programmation, suivie au S3, la méthodologie de programmation présentée s'appliquait à des traitements de données simples ou simplement structurées : entiers et tableaux. Lorsqu'un programme doit gérer des données complexes, ne pouvant être structurées en tableaux ou présentes en très grand nombre, la difficulté majeure de programmation repose alors sur l'identification, la modélisation et le traitement de ces données.

Si l'on prend l'exemple d'un moteur de recherche de pages web (google par exemple), on s'aperçoit que les données traitées par ce type de programme vérifient tous les qualificatifs ci-dessus :

- Données complexes : une page html, bien que pouvant être traitée comme un simple tableau de caractères, présente toujours une structuration syntaxique ou sémantique forte. Si l'on cherche à représenter une page html en mémoire, un simple tableau de caractères ne permettra pas de représenter toute cette complexité.
- Données non structurées (en tableau) : les relations entre les pages html, exprimées à travers les liens représentés par des urls ne peuvent pas être évaluées par des opérations arithmétiques sur des indices d'un tableau. Ces relations sont pourtant au cœur du principe de fonctionnement d'un moteur de recherche.
- Données en très grand nombre : le nombre de pages html est considérable. Si l'on met en œuvre un algorithme de complexité linéaire (dans un tableau par exemple) ou logarithmique (dans un tableau trié) pour effectuer une recherche, on peut démontrer qu'il faudrait un temps extrêmement long pour donner un résultat certain. Il est donc nécessaire de diminuer cette complexité, tant en temps qu'en espace, pour pouvoir exploiter au mieux l'ensemble des connaissances représentées par le web.

Une analyse similaire peut être faite sur des applications différentes : logiciel de production cinématographique, jeu vidéo, système d'information de santé, système bancaire, ...

Lors du développement d'une application à grande échelle (gérant des données complexes), il est nécessaire d'identifier et de spécifier les données et les traitements associés très tôt dans le processus de développement.

La première étape importante de ce processus est la spécification des données et des traitements que l'on peut leur appliquer. Cette **spécification** doit répondre uniquement à la question **quoi faire ?** Le but de cette spécification est de définir l'interface d'utilisation (**représentation**

externe) de cette donnée et de lui donner une sémantique abstraite. Le résultat de cette spécification est appelé un **type abstrait de données**.

Cette sémantique est indépendante de la réponse à la question **comment le faire ?**, qui amènera, dans un second temps, à la définition de la **représentation interne** des données et à l'**implantation** dans le langage de programmation choisi des traitements associés.

Les langages de programmation impératifs typés offrent plusieurs types abstraits de données prédéfinis. Ce sont les types de base de tout langage de programmation que sont les entiers, les nombres réels, ... Ceci conduit à une utilisation simple et naturelle des variables définies à partir de ces types. Ainsi, le programmeur, l'utilisateur des types abstraits de données, n'a besoin de connaître que la représentation externe d'un type pour l'utiliser (Figure 1).

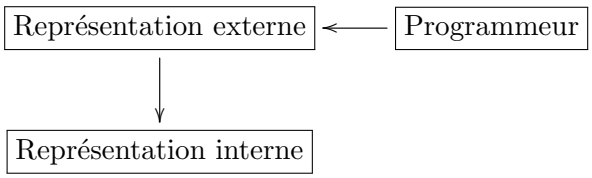


FIGURE 1 – Points de vue d'un type abstrait de données

Les types de base des langages de programmation sont la plupart du temps des types scalaires définis de façon abstraite par leurs propriétés arithmétiques et représentés en interne de façon dépendante de la machine ou du langage de programmation. Cette représentation interne peut rajouter des contraintes sur l'utilisation de ces types et celles-ci apparaissent alors dans leur représentation externe sous forme d'axiomes, postulats considérés comme évidents.

Exemple (incomplet) de type abstrait de données : les entiers

— Représentation externe des entiers vue par le programmeur (en C) :

```
int                 /* le nom du type */
5, -6, 21           /* des constantes */
+, -, *, /          /* les opérations permises */
```

Ceci représente l'interface des entiers. Cette représentation externe donne la possibilité d'une utilisation naturelle des entiers.

— Représentation interne des entiers : il s'agit d'une représentation binaire ; par exemple, l'entier 3 est codé par 00000011 sur un octet.

La représentation externe est incomparablement plus facile à utiliser qu'une représentation interne. Si un programmeur veut effectuer la multiplication de deux entiers, il est plus facile d'écrire $3 * 4$ que $00000011 * 00000100$ (Figure 2).

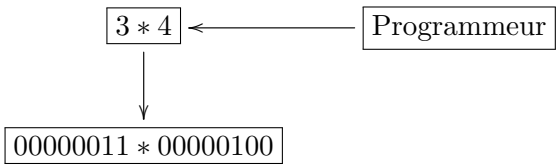


FIGURE 2 – Utilisation du type abstrait de données *int*

Lors du développement d'une application à grande échelle, plusieurs rôles sont à considérer et ces rôles peuvent être attribués à plusieurs personnes différentes, éventuellement géographi-

quement très distantes les unes des autres. Il est donc primordial de pouvoir présenter de façon claire, précise et compréhensible les différentes tâches de développement à réaliser pour produire l'application. Dans la suite de ce cours, nous nommerons les différents rôles vis-à-vis de leur articulation autour des types abstraits de données. D'un côté, nous avons le *spécifieur*, qui répond à la question **quoi faire ?** en fournissant une **spécification** des types abstraits de données. De l'autre côté, nous avons l'*utilisateur*, qui va développer l'application en utilisant l'interface externe du type abstrait de données. Entre les deux, nous avons le *programmeur*, qui va fournir la représentation interne du type abstrait de données et l'implantation des opérations que l'on peut effectuer sur ce type. C'est ce rôle là qui sera principalement attribué aux étudiants tout au long du semestre ainsi que celui d'*utilisateur*.

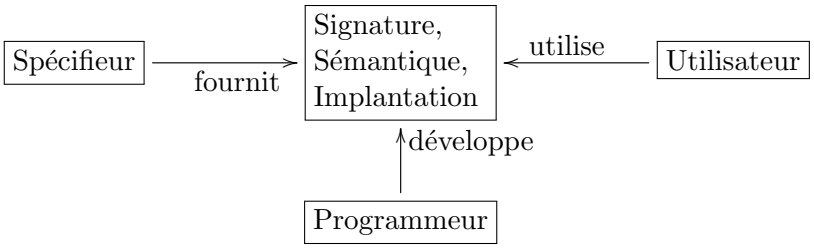


FIGURE 3 – Rôles dans la programmation de types abstraits de données

Dans ce cours, nous nous intéressons aux types abstraits de données fondamentaux permettant de gérer de façon efficace des collections, au sens général du terme, de données.

Nous proposons la classification suivante qui sera utilisée tout au long de cette UE :

— Les types à structure linéaire : Pile (*Stack*), File (*Queue*), Liste(*List*)
— Les types à structure arborescente : Arbres Binaires (*BinaryTree*), Arbres Rouge-Noir (*RBTree*)
— Les types à structure associative : Tables de hachage (*HashTable*)

2.3 Spécification d'un type abstrait de données

La spécification d'un type abstrait de données se fait en fournissant d'une part sa signature, définissant la syntaxe d'un TAD, et d'autre part un ensemble d'axiomes définissant la sémantique du TAD.

Signature d'un TAD

La signature d'un TAD définit sa syntaxe et doit faire apparaître le profil fonctionnel des opérations que l'on peut réaliser sur ce TAD. D'une manière générale, la signature comprend les informations suivantes :

Sorte : Nom du type défini. On veillera à choisir un nom significatif par rapport au type défini. Bien que la sémantique ne soit pas associée au nom du type, pour une meilleure compréhension, à la fois du TAD et de son implantation future, ce nom devra être représentatif du type défini.

Utilise : Liste des types de données utilisés par le type défini. Cette liste de types de données permettra de définir les dépendances de type mais aussi de définir correctement les opérateurs sur le type.

Opérateurs : Profil fonctionnel des opérateurs. Dans la spécification d'un TAD, toutes les opérations sont considérées comme des fonctions. Ces fonctions seront donc définies par leur nom, leur domaine et leur co-domaine. Outre ces profils, il peut être important de

classer ces opérations selon qu'elles sont nécessaires pour créer tous les termes de la sorte définie (*i.e.* créer toutes les valeurs possibles d'une variable de ce type) ou pas. Dans le premier cas, on parlera de constructeur, dans le second on gardera le terme générique d'opérateurs.

Par exemple, si l'on souhaite donner la signature d'un TAD Vecteur, permettant de représenter une collection d'Elements indicés par un Entier (en fait, un tableau), on écrirait la signature suivante :

Sorte : Vecteur
Utilise : Entier, Element
Opérateurs_Constructeurs :
vect : $\rightarrow$ Vecteur
changer-ième : Vecteur $\times$ Entier $\times$ Element $\rightarrow$ Vecteur
Opérateurs :
ième : Vecteur $\times$ Entier $\rightarrow$ Element
borne-sup : Vecteur $\rightarrow$ Entier
borne-inf : Vecteur $\rightarrow$ Entier

Remarquons que cette signature (description) du type donne sa syntaxe mais ne suffit pas à définir celui-ci. La relative compréhension du lecteur est due au choix des noms et reste basée sur son intuition, ce qui manque pour le moins de rigueur. Pour s'en persuader il suffit de considérer la signature ci-dessous, qui est identique, au choix des noms près :

Sorte : R
Utilise : X, P
Opérateurs_Constructeurs :
a : $\rightarrow$ R
b : R $\times$ X $\times$ P $\rightarrow$ R
Opérateurs :
c : R $\times$ X $\rightarrow$ P
d : R $\rightarrow$ X
e : R $\rightarrow$ X

Il apparaît alors clairement qu'une partie de la définition manque, celle qui donne la sémantique du TAD.

Sémantique d'un TAD

Si on veut définir un type abstrait indépendamment de ses implémentations possibles, la méthode la plus courante consiste à énoncer les propriétés des opérations sous forme d'axiomes. Ces axiomes, énoncés sous forme de propositions logiques fournissent la spécification des opérateurs de manière assimilable au triplet de Hoare vu au semestre précédent.

Ces axiomes peuvent être organisés selon deux catégories, les **pré-conditions** et les **propriétés**. Les axiomes représentant des pré-conditions doivent spécifier si les opérateurs sont des fonctions totales (pouvant s'appliquer sur tous les termes des sortes définies et utilisées) ou partielles (ne pouvant s'appliquer que sur un sous-ensemble de ces termes). Les axiomes représentant les propriétés définissent, de manière fonctionnelle le comportement et la logique des opérateurs.

Par exemple, sur notre TAD Vecteur, le comportement de l'opérateur *ième* peut se définir de la façon suivante :

$$\mid \quad \textit{borne-inf(v)} \leq i < \textit{borne-sup(v)} \Rightarrow \textit{ième(changer-ième(v, i, e), i)} = e$$

Cet axiome exprime que, dans la mesure où i est compris entre les bornes d'un vecteur v, quand on construit un nouveau vecteur en donnant au ième élément la valeur e, et que l'on accède ensuite au ième élément de ce nouveau vecteur, on obtient e. Cette propriété est satisfaite quelles que soient les valeurs, correctement typées, données aux variables.

Un autre axiome serait :

$$\begin{array}{l} \textit{borne-inf(v)} \leq i < \textit{borne-sup(v))} \wedge \textit{borne-inf(v)} \leq j < \textit{borne-sup(v))} \wedge (i \neq j) \\ \qquad \Rightarrow \textit{ième(changer-ième(v, i, e), j)} = \textit{ième(v, j)} \end{array}$$

Cet axiome combiné avec le précédent exprime que seul le ième élément a changé dans le nouveau vecteur.

Lorsqu'on implantera (programmera) les types abstraits de données, toutes les propriétés définies par les axiomes devront être vérifiées.

La définition d'un type abstrait de données est donc composée d'une signature et d'un ensemble d'axiomes.

Les axiomes sont accompagnés de la déclaration d'un certain nombre de variables. Ce type de définition s'appelle une définition algébrique ou axiomatique d'un type abstrait. Pour abréger on parle souvent de *types abstraits algébriques*.

Lors de la définition d'un *type abstrait algébrique*, et principalement lors de la construction de l'axiomatique de ce TAD, deux questions peuvent être posées :

— N'y-a-t'il pas des axiomes contradictoires ? Il s'agit d'une question de **consistance**.

— Y-a-t'il un nombre suffisant d'axiomes ? Il s'agit d'une question de **complétude**.

La consistance et la complétude (avec, pour cette dernière, certaines simplifications) sont deux propriétés clés pour la définition d'un TAD et son implantation. Elles assurent une certaine liberté dans les choix d'implantation qui seront faits, comme on le verra dans la suite du cours, tout en assurant une utilisation indépendante de l'implantation.

Un exemple d'inconsistance serait, dans le cas de notre TAD Vecteur, de trouver une expression v (un terme de la sorte Vecteur) et une expression entière i telles que l'on puisse démontrer, en utilisant les axiomes, les deux propriétés :

$$\begin{array}{l} \textit{ième(v, i)} = 0 \\ \textit{ième(v, i)} = 1 \end{array}$$

La notion de complétude est plus complexe et demande à être examinée avec soin lorsqu'on travaille sur les types abstraits de données. En mathématiques, une théorie T, et par la suite, le système d'axiomes qui la définit, est dite complète si elle est consistante et si, pour toute formule P sans variable, on sait démontrer soit P soit $\neg P$.

Cette notion est trop forte pour les types abstraits algébriques : elle entraîne que toute égalité de deux expressions sans variable doit être soit vraie, soit fausse. Or souvent on veut (et on doit !) laisser une latitude aux programmeurs qui implanteront le TAD. Prenons l'exemple des Vecteur. Supposons que l'on applique *changer-ième* à un vecteur v avec pour arguments 5 et a, puis 10 et b. Considérons le vecteur obtenu si on change l'ordre des deux opérations. A priori on a envie de dire que les deux résultats sont égaux. Mais cela peut ne pas être vrai pour certaines implantations : par exemple une liste chaînée des couples $<$ *indice, élément* $>$, où *changer-ième* fait simplement une adjonction d'un nouveau couple en tête de la liste. En fait

ce qui est important c'est que deux vecteurs, quand on leur applique *ième*, rendent toujours le même résultat. Le fait que leurs représentations soient différentes n'est pas essentiel.

Le critère utilisé pour les types abstraits algébriques est la **complétude suffisante** : les axiomes doivent permettre de déduire une valeur d'un type prédéfini (sortes utilisées par le TAD) pour toute application d'un opérateur à un objet d'un type défini (sortes définies par le TAD).

Comme on obtient les objets de type défini par les constructeurs, il faut alors écrire des axiomes qui définissent le résultat de la composition des opérateurs avec tous les constructeurs. Cependant, cette règle doit être modulée par le fait que certains opérateurs sont des fonctions partielles (ne s'appliquant pas sur certains termes) dont le domaine de définition a été précisé par les pré-conditions. Pour obtenir une spécification de TAD suffisamment complète, on écrira donc le système d'axiomes permettant de déduire une valeur pour tous les opérateurs, appliqués sur tout objet de type défini appartenant au domaine de définition de l'opérateur.

En ce qui concerne les axiomes, revenons à l'exemple des vecteurs. Les axiomes donnés précédemment suffisent à définir l'opération *ième* par rapport à l'opération *changer-ième*. D'autre part, il n'y aura pas d'axiome définissant *ième(vect(), i)* puisque *vect* retourne un vecteur où aucun élément n'est défini (il faudra prendre cela en compte dans une pré-condition).

Passons maintenant à la définition des opérations *borne-inf* et *borne-sup*. Nous allons ici avoir un problème puisque nous n'avons pas la possibilité avec la signature actuelle d'écrire une expression de type VECTEUR dont on connaîtrait les paramètres (bornes inférieure et supérieure) : les seules opérations disponibles sont *vect* qui produit un VECTEUR sans élément défini mais sans paramétrage possible et *changer-ième* qui prend en argument un vecteur déjà constitué (paramètres demeurant inchangés). Il va donc falloir ajouter par exemple une opération qui correspond au contenu d'un vecteur non initialisé mais dont on connaît les bornes. En fait, le plus simple est de modifier l'opération *vect* :

| *vect :* ENTIER × ENTIER → VECTEUR

On aura alors les axiomes :

| *borne-inf(vect(i, j)) = i*
| *borne-sup(vect(i, j)) = j*

Et il n'y aura toujours pas d'axiome définissant *ième(vect(i, j), k)* (pour la même raison que celle donnée précédemment). Par contre, il faut désormais écrire la pré-condition sur l'opération *ième*. Pour cela on a besoin d'une opération auxiliaire sur les vecteurs, qui permet de savoir si un élément a été associé à un certain indice :

| *init :* VECTEUR × ENTIER → BOOLEAN

avec les axiomes

| *init(vect(i,j), k)* = FAUX
| $borne\text{-}inf(v) \leq i < borne\text{-}sup(v) \Rightarrow init(changer\text{-}ième(v, i, e), i) =$ VRAI
| $borne\text{-}inf(v) \leq i < borne\text{-}sup(v) \wedge i \neq j \Rightarrow init(changer\text{-}ième(v, i, e), j) = init(v, j)$

Avec cet opérateur, on peut donc définir la pré-condition sur l'opérateur *ième* :

| *ième(v, i)* ***défini ssi*** $(borne\text{-}inf(v) \leq i < borne\text{-}sup(v)) \wedge init(v, i) =$ VRAI

Afin d'être suffisamment complète, la spécification du TAD VECTEUR doit aussi être com-

plétée par les axiomes suivants :

> *borne-inf(changer-ième(v, i, e)) = borne-inf(v)*
> *borne-sup(changer-ième(v, i, e)) = borne-sup(v)*

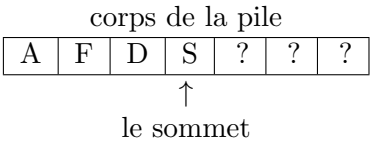
La définition finale du type VECTEUR est donnée ci-après. Elle est suffisamment complète. Tout vecteur est le résultat d'une opération *vect* et d'une suite d'opérations *changer-ième*. Les axiomes permettent de déduire le résultat de *init*, *borne-sup* et *borne-inf* dans tous les cas. Pour ce qui est de *ième*, on peut établir son résultat en utilisant les axiomes quand la pré-condition est satisfaite, c'est-à-dire quand une des opérations *changer-ième* a pour argument l'indice en argument de *ième*. En conclusion, un critère pour savoir si on a écrit suffisamment d'axiomes est : peut-on déduire de ces axiomes le résultat de chaque opérateur sur son domaine de définition ?

> **Sorte :** VECTEUR
> **Utilise :** ENTIER, ELEMENT, BOOLEEN
>
> **Opérateurs_Constructeurs :**
> *vect :* ENTIER × ENTIER → VECTEUR
> *changer-ième :* VECTEUR × ENTIER × ELEMENT → VECTEUR
> **Opérateurs :**
> *init :* VECTEUR × ENTIER → BOOLEAN
> *ième :* VECTEUR × ENTIER → ELEMENT
> *borne-sup :* VECTEUR → ENTIER
> *borne-inf :* VECTEUR → ENTIER
>
> **Pré-conditions :**
> *ième(v, i)* ***défini ssi*** *(borne-inf(v) ≤ i < borne-sup(v)) ∧ init(v, i) =* VRAI
>
> **Axiomes :**
> *init(vect(i,j), k) =* FAUX
> *borne-inf(v) ≤ i < borne-sup(v) ⇒ init(changer-ième(v, i, e), i) =* VRAI
> *borne-inf(v) ≤ i < borne-sup(v) ∧ i ≠ j ⇒ init(changer-ième(v, i, e), j) = init(v, j)*
> *borne-inf(v) ≤ i < borne-sup(v) ⇒ ième(changer-ième(v, i, e), i) = e*
> *borne-inf(v) ≤ i < borne-sup(v) ∧ borne-inf(v) ≤ j < borne-sup(v) ∧ i ≠ j*
> *⇒ ième(changer-ième(v, i, e), j) = ième(v, j)*
> *borne-inf(vect(i,j)) = i*
> *borne-inf(changer-ième(v, i, e)) = borne-inf(v)*
> *borne-sup(vect(i,j)) = j*
> *borne-sup(changer-ième(v, i, e)) = borne-sup(v)*

2.4 Exercice : spécification du TAD STACK

Une des structures de données fondamentales en informatique est la pile (STACK). Une pile est une structure de stockage d'informations de même type permettant d'accéder uniquement à la dernière information qui a été ajoutée à la pile. Les piles sont donc gérées sur le principe du "dernier entré, premier sorti" (LIFO). Les opérations possibles sur une pile sont :

— Ajouter un élément à une pile
— Accès au sommet de pile (le dernier élément ajouté)
— Supprimer un élément d'une pile (le dernier ajouté)



Les idées que l'on veut exploiter pour définir ce type STACK sont :
— La pile doit être vide à la création
— Une pile vide est construite par `createStack`
— Une pile non vide est construite par `createStack` suivie par une suite de `push`
— On accède au sommet de pile par `top`
— On supprime le sommet de pile par `pop`

`createStack` et `push` seront les constructeurs ; ils sont indispensables pour représenter n'importe quel terme du type STACK (pile vide ou non).
Exemple : `push (push (push (push (createStack,'A'),'F'),'D'),'S')`

Écrire la spécification suffisamment complète du TAD STACK correspondant aux propriétés énoncées ci-dessus.