TYPES ABSTRAITS

1 Signature

Il est toujours intéressant de pouvoir travailler sur l'énoncé et la résolution d'un problème indépendamment d'une implantation particulière. En particulier, la représentation des données n'est pas fixée.

A ce premier niveau les données sont considérées de manière abstraite: on se donne une notation pour les décrire ainsi que l'ensemble des opérations qu'on peut leur appliquer et les propriétés de ces opérations. On parle alors de ***type abstrait de données***. La conception de l'algorithme se fait en utilisant les opérations du type abstrait. Les différentes représentations du type abstrait permettent d'obtenir différentes versions de l'algorithme si le type abstrait n'est pas un type du langage de programmation que l'on veut utiliser.

Précisons maintenant à l'aide d'un exemple élémentaire ce qu'est un type abstrait. Considérons le type réel des langages de programmation: on a une convention pour écrire les constantes réelles; on sait calculer sur les réels en utilisant '+', '-', '*', … dont les propriétés ne sont pas rappelées car elles sont bien connues. On peut manipuler les réels sans avoir à connaître leur représentation interne (mantisse, exposant). ….

Il existe plusieurs manières de définir un type de données. Toutes ont en commun le concept de *signature*. La signature d'un type de données décrit la syntaxe du type (nom des opérations, type de leurs arguments) mais elle ne définit pas les *propriétés* des opérations du type. C'est par ce dernier aspect que diffèrent les méthodes de définition des types de données.

Voici un exemple de signature:

Type Vecteur
Opérations

ième	:	Vecteur X Entier → Element
changer-ième	:	Vecteur X Entier X Element → Vecteur
bornesup	:	Vecteur → Entier
borneinf	:	Vecteur → 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:

Type R
Opérations

o	:	R X T → S
p	:	R X T X S → R
q	:	R → T
v	:	R → T

Il apparaît alors clairement qu'une partie de la définition manque, celle qui donne une sémantique, c'est à dire une signification, aux noms R, S, T, o, p, q, v.

2 Hiérarchie dans les types abstraits

On introduit une hiérarchie « d'utilisation » entre les types. Par exemple le type Vecteur est au-dessus des types Entier et Element dans cette hiérarchie. Cette hiérarchie est fondamentale pour structurer les définitions de type abstrait.

Elle permet d'introduire une classification importante parmi les types de données et ses opérations:

- ***Opération interne***
  opération qui rend un résultat du type défini. C'est la cas pour Vecteur de *changer-ième*. Toute valeur d'un type défini est le résultat d'une opération interne.
- ***Observateur***
  opération dont le résultat est d'un autre type prédéfini ou déjà défini comme type abstrait. C'est la cas pour Vecteur de *ième*, *bornesup* et *borneinf*.

3 Des types aux classes

Les classes concernent les langages à objets.

- A chaque type on associe une classe. En programmation type et classes deviennent synonymes.
- Les opérations du type se traduisent (se réalisent ou encore s'implantent) par les méthodes de la classe.
- Classe = Type
  De manière informelle, une classe est un élément logiciel qui décrit un type de données et son implantation totale ou partielle.
- Traduction des opérations
  Le premier argument d'une opération, qui est du type défini, devient le receveur, c'est à dire l'objet sur lequel on applique la méthode. Les autres arguments sont les paramètres de la méthode. Les méthodes qui renvoient un résultat du type pourront générer un nouvel objet (fonctions) ou modifier l'objet courant (procédures).
  On sait qu'on a besoin également d'opérations de construction d'objets, vides ou constants: les constructeurs.

4 Description des propriétés d'un type de données

Le problème est de donner une signification (une sémantique) aux opérations de la signature.

Si on veut définir un type abstraitement, c'est-à-dire indépendamment de ses implémentations possibles, la méthode la plus courante consiste à énoncé les propriétés des opérations sous forme d'***axiomes***. Par exemple:

$$borneinf(v) \leq i \leq bornesup(v) \Rightarrow ieme(changer\text{-}ieme(v,i,e),i) = e$$

où v, i et e sont des variables respectivement de type Vecteur, Entier et Element.

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 $i^{ème}$ élément la valeur *e*, et que l'on accède ensuite au $i^{ème}$ 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:

$$borneinf(v) \leq i \leq bornesup(v) \ \ \& \ \ borneinf(v) \leq j \leq bornesup(v) \ \ \& \ \ i \neq j$$

$$==> ieme(changer\text{-}ieme(v,i,e),j) = ieme(v, j)$$

Cet axiome combiné avec le précédent exprime que seul le $i^{ème}$ élément a changé dans le nouveau vecteur.

Ces deux propriétés doivent être satisfaites par toute implantation du type abstrait Vecteur.

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***.

Questions:

- N'y-a-t-il pas des axiomes contradictoires: problème de ***consistance***?
- Y-a-t-il un nombre suffisant d'axiomes: problème de ***complétude***?

Un exemple d'inconsistance serait, dans le cas de vecteurs d'entiers, de trouver une expression *v* de type Vecteur et une expression entière i telles que l'on puisse démontrer, en utilisant les axiomes, les deux propriétés:

$$ieme(v, i) = 0 \qquad \text{et} \qquad ieme(v, i) = 1$$

La notion de complétude est plus délicate et demande à être examinée avec soin quand on travaille sur des types abstraits. En mathématiques, une théorie *T*, et par 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 *non P*.

Cette notion est trop forte pour les *types abstraits algébriques* : elle entraine 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 implémenteurs. Prenons l'exemple des vecteurs. Supposons que l'on applique *changer-ieme* 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 implémentations: par exemple une liste chainée des couples < indice, element >, où *changer-ieme* 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 *ieme*, rendent toujours le même résultats. 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 pour toute application d'un observateur à un objet d'un type défini.

Comme on obtient les objets par les opérations internes, il faut écrire des axiomes qui définissent le résultat de la composition des observateurs avec toutes les opérations internes. De tels axiomes ont été donnés pour *ieme* et *changer-ieme*, dans l'exemple des vecteurs.

Cependant la règle qui vient d'être énoncée doit être modulée: il existe des types de données où certaines opérations sont des fonctions partielles, non définies partout. C'est par exemple le cas du sommet d'une pile, qui n'est pas défini pour une pile vide ou encore de l'accès à une place non initialisée dans le cas de tableaux. On reformule donc la règle ci-dessus en disant que l'on doit pouvoir déduire *une valeur pour tous les observateurs sur tout objet d'un type défini appartenant au* ***domaine de définition*** *de cet observateur*. Le domaine de définition d'une opération partielle est défini par une ***précondition***.

Revenons à l'exemple des vecteurs. Les axiomes donnés précédemment définissent l'opération *ieme* suffisamment par rapport à l'opération *changer-ieme*. Cependant on n'a pas la possibilité avec la signature actuelle d'écrire une expression de type Vecteur sans variable: la seule opération interne est changer-ieme qui prend en argument un vecteur. Il faut donc ajouter une opération interne qui correspond au contenu d'un vecteur non initialisé mais dont on connait les bornes:

$$vect \quad : \quad \text{Entier X Entier} \quad \rightarrow \quad \text{Vecteur}$$

avec les axiomes

$$borneinf(vect(i, j)) = i$$
$$bornesup(vect(i, j)) = j$$

Il n'y aura pas d'axiomes définissant $ieme(vect(i, j), k)$ puisque *vect* retourne un vecteur où aucun élément n'est défini. En revanche, il faut écrire la précondition sur l'opération *ieme*. 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:

$$init : \quad \text{Vecteur X Entier} \quad \rightarrow \quad \text{Booleen}$$

avec les axiomes:

$$init(vect(i, j), k) = \text{faux}$$
$$(borneinf(v) \leq i \leq bornesup(v)) \Rightarrow (init(changer\text{-}ieme(v, i, e), i) = \text{vrai})$$
$$(borneinf(v) \leq i \leq bornesup(v)) \quad \& \quad i \neq j) \Rightarrow$$
$$(init(changer\text{-}ieme(v, i, e), j) = \text{init}(v, j))$$

L'opération ieme est définie si, et seulement si:

$$(borneinf(v) \leq i \leq bornesup(v)) \quad \& \quad init(v, i) = \text{vrai}$$

La formule ci-dessus est appelée une précondition sur l'opération *ieme*.

Il manque les définitions des observateurs *bornesup* et *borneinf* sur le résultat de l'opération interne *changer-ieme*:

$$borneinf(changer\text{-}ieme(v, i, e)) = borneinf(v)$$
$$bornesup(changer\text{-}ieme(v, i, e)) = bornesup(v)$$

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-ieme*. Les axiomes permettent de déduire le résultat de *init*, *bornesup* et *borneinf* dans tous les cas. Pour ce qui est de *ieme*, on peut établir son résultat en utilisant les axiomes quand la précondition est satisfaite, c'est-à-dire quand une des opérations *changer-ieme* a pour argument l'indice en argument de *ieme*.

En conclusion un critère pour savoir si on a écrit suffisamment d'axiomes est: peut-on déduire de ces axiomes le résultat de chaque observateur sur son domaine de définition?

Définition finale du type Vecteur

type Vecteur

Opérations

vect	:	Entier X Entier	→	Vecteur	
changer-ieme	:	Vecteur X Entier X Element	→	Vecteur	
ieme	:	Vecteur X Entier	→	Element	
init	:	Vecteur X Entier	→	Booleen	
borneinf	:	Vecteur	→	Entier	
bornesup	:	Vecteur	→	Entier	

Préconditions

ieme(*v*, i) **est-défini-ssi**

borneinf(*v*) ≤ *i* ≤ *bornesup*(*v*) & *init*(*v*, *i*) = vrai

Axiomes

borneinf(*v*) ≤ *i* ≤ *bornesup*(*v*) ⇒
ieme(changer-ieme(v, i, e), i) = e

borneinf(*v*) ≤ *i* ≤ *bornesup*(*v*) & *borneinf*(*v*) ≤ *j* ≤ *bornesup*(*v*) & *i* ≠ *j* ⇒
ieme(*changer-ieme*(*v*, *i*, *e*), *i*) = *ieme*(*v*, *j*)

init(vect(i, j), k) = faux

borneinf(*v*) ≤ *i* ≤ *bornesup*(*v*) ⇒
init(*changer-ieme*(*v*, *i*, *e*), *i*) = vrai

borneinf(*v*) ≤ *i* ≤ *bornesup*(*v*) & *i* ≠ *j* ⇒
init(*changer-ieme*(*v*, *i*, *e*), *j*) = *init*(*v*, *j*)

borneinf(*vect*(*i*, *j*) = *i*
borneinf(*changer-ieme*(*v*, *i*, *e*)) = *borneinf*(*v*)

bornesup(*vect*(*i*, *j*) = *j*
bornesup(*changer-ieme*(*v*, *i*, *e*)) = *bornesup*(*v*)