TOULOUSE

Informatique

iut Midi-Pyrénées

UNIVERSITÉ TOULOUSE III

Introduction aux Types Abstraits de Données

Christian Percebois
Département Informatique
IUT A Paul Sabatier

Janvier 2010

TABLE DES MATIERES

Chapitre 1 : Les Types Abstraits de Données (TAD)

La conception d'un algorithme nécessite à chaque étape d'affinage de découvrir des structures de contrôle enchaînant les actions complexes mises en évidence par la décomposition. Le choix de ces actions n'est pas indépendant des données gérées par l'algorithme.

Considérons par exemple le dessin à l'écran d'un polygone à partir d'une liste de points désignant ses sommets. On écrira au premier niveau d'affinage de l'algorithme :

```
-- dessiner un polygone à partir de sa liste de sommets
se positionner sur les deux premiers sommets de la liste ;
tantque il reste un segment à tracer faire
        tracer le segment du sommet origine (premier sommet)
        au sommet extrémité (deuxième sommet) ;
        passer au sommet suivant de la liste ;
fin tantque ;
```

Cet exemple montre que le choix de la répétition comme structure de contrôle est lié à la *liste* des *sommets* du polygone. Il existe aussi un lien logique entre deux *sommets* et le *segment* correspondant. Ceci nous amène à considérer une donnée d'un algorithme comme un ensemble d'informations élémentaires et un ensemble de relations entre ces informations. C'est ce qu'on appelle une *structure de données*.

Une structure de données dans un langage de programmation se matérialise le plus souvent par la définition d'un type ; nous distinguons, du plus simple au plus complexe :

- les types élémentaires (voir cours d'algorithmique),
- les types composés (voir cours d'algorithmique),
- les types abstraits de données.

1. NOTION DE TYPE ABSTRAIT DE DONNEES

Un type abstrait permet de définir une structure de données. Son utilisation s'inscrit dans la démarche descendante d'écriture des algorithmes qui nécessite de considérer les données d'un algorithme de manière abstraite. On parle alors de Type Abstrait de Données (TAD). Par exemple, pour dessiner un polygone, nous introduisons les TAD *Liste*, *Sommet* et *Segment*.

Définition

Un type abstrait est défini par un ensemble de valeurs et un ensemble d'opérations applicables à ces valeurs. Les valeurs sont abstraites car la vue qu'en a l'utilisateur ne dépend pas de la représentation interne choisie pour les coder.

Remarque

Le terme *abstrait* pour qualifier de telles structures de données provient du remplacement d'une situation complexe et détaillée du monde réel par un modèle compréhensible grâce auquel nous pouvons résoudre un problème. Cela signifie que nous faisons abstraction des détails dont l'impact sur les solutions à un problème est minimal ou inexistant, créant par là même un modèle qui nous permet de nous consacrer à l'essence du problème.

Différents points de vue

Dès lors, on distingue trois points de vue :

- celui du *concepteur* du type abstrait qui définit précisément, complètement et de façon non ambiguë les opérations du type abstrait,
- celui du *programmeur* qui choisit une représentation physique du type et qui programme, dans un langage de programmation pour une machine donnée, les opérations définies par le concepteur,
- celui de *l'utilisateur* (ou *client*) qui exploite le type abstrait pour son application et qui ne connaît que les opérations du type définies par le concepteur.

Pour un TAD donné, le niveau du concepteur est appelé *spécification* (*quoi ?*) et celui du programmeur *implémentation* (*comment ?*).

Cette séparation entre spécification et implémentation est motivée par le souhait de ne montrer à un utilisateur du type abstrait que les propriétés externes de la donnée et de masquer les détails d'une implémentation particulière. Elle permet en particulier de proposer plusieurs implémentations d'un même type abstrait de données. Notons que les langages de programmation modernes fournissent un support linguistique permettant d'isoler la spécification de l'implémentation.

Propriétés des types abstraits de données

Des trois points de vue exprimés par le concepteur, le programmeur et l'utilisateur d'un TAD découlent trois propriétés qui caractérisent cette approche : *l'encapsulation*, *l'indépendance temporelle* et *l'indépendance spatiale* :

- l'encapsulation exprime que la représentation physique d'un type abstrait de données est cachée à un utilisateur qui n'en connaît que les opérations et les propriétés,
- l'indépendance temporelle traduit qu'on peut définir un type de données indépendamment du contexte d'une application qui l'utilise,
- l'indépendance spatiale (ou localisation) exprime que tous les aspects relatifs à une même structure de données sont physiquement rattachés au même module.

Indépendances temporelle et spatiale conduisent à des programmes dotés chacun d'une grande unité (*forte cohésion*[1]) et d'un petit nombre de relations directes et explicites avec les autres programmes (*faible couplage*[2]).

En fait, cette notion de type abstrait n'est pas nouvelle. Si l'on considère par exemple les réels des langages de programmation, le programmeur est capable de les manipuler sans en connaître leur représentation en machine ; il en connaît également les propriétés algébriques.

Avec les types abstraits de données, on définit des types en programmation ; c'est donc un enrichissement du langage algorithmique. Les types *Sommet, Segment, Polygone*, *Chaîne de caractères*, *Nombre complexe*, *Fichier de clients*, *Point d'écran*, *Compte bancaire*, ... sont autant d'exemples de types abstraits de données qu'on peut utiliser dans une application. Comme pour les sous-programmes, l'utilisation d'un type abstrait s'envisage dès que sa spécification est connue.

Exemple

Pour le dessin du polygone, on peut imaginer :

TAD	***Opérations du TAD***
Liste	Connaître son nombre d'éléments Accéder à un élément de rang donné Modifier un élément de rang donné[3] Insérer un élément à un rang donné Supprimer un élément de rang donné …
Sommet	Connaître son abscisse Connaître son ordonnée Modifier sa taille Translater un sommet …
Segment	Connaître son sommet origine Connaître son sommet extrémité Tracer un segment Calculer sa longueur Déterminer son point milieu Colorier un segment …
Polygone	Connaître sa liste de sommets Dessiner un polygone Calculer son périmètre Calculer son aire Colorier un polygone …

[1] mesure la force de solidarité qui lie les opérations

[2] mesure l'amplitude des contraintes qui assujettissent un segment de programme aux autres

[3] Cette opération n'est pas nécessaire pour exprimer l'algorithme ; elle est mentionnée ici car elle s'applique à une liste et illustre les principes d'indépendances temporelle et spatiale des TAD.

2. SPECIFICATION FONCTIONNELLE D'UN TAD

Définition

La spécification fonctionnelle d'un type abstrait de données précise l'ensemble des opérations autorisées sur les représentants (valeurs et variables) du type ainsi que les propriétés de ces opérations.

2.1. Opérations

Une opération exprime une fonctionnalité sur un représentant du type abstrait étudié. Elle précise notamment à quels types de données elle s'applique et quel type de résultat elle produit.

Pour dessiner le polygone à l'écran, on manipule ses sommets qu'on choisit de matérialiser dans ce qui suit par des points d'écran. Plus généralement, il s'agit de gérer des points en coordonnées cartésiennes à l'écran, ce qui nous conduit à spécifier le TAD *Point* d'écran.

Dans un premier temps, nous avons à identifier un certain nombre d'opérations relatives à un point. On peut imaginer :

- calculer l'ordonnée d'un point,
- translater un point,
- modifier la couleur d'un point,
- ...

Le problème consiste ici à définir la notion de point de façon à nommer, puis manipuler des valeurs de type *Point* comme on manipule des valeurs de type prédéfini *Entier* ou *Réel*. Le type *Point* est donc défini grâce à des propriétés abstraites et non pas par des informations sur les adresses de rangement ou sur le nombre de bits de leur représentation.

Plus formellement, si on introduit le type *Point*, on écrira :

ordonnée : Point → Réel
translater : Point x Réel x Réel → Point
modifierCouleur : Point x Couleur → Point
...

Une telle description est appelée signature du type abstrait : elle spécifie la syntaxe des opérations du type, sans en préciser la sémantique. Celle-ci sera détaillée ultérieurement lors de l'écriture des propriétés. Plus précisément, une signature est composée d'un ensemble de symboles de sortes, qui représentent des types de données, et d'un ensemble de symboles d'opérateurs, qui représentent les opérations sur ces données. A chaque symbole d'opérateur est associé un profil exprimant le type des données d'entrée et de sortie.

Indiquons tout d'abord la syntaxe utilisée pour écrire la signature d'un type abstrait. La forme générale d'une opération *f* (on dit aussi fonction) sur une structure de données *T* qu'on confondra avec le type *T* s'écrira $f : X_1 \times X_2 \times ... \times X_N \rightarrow Y_1 \times Y_2 \times ... \times Y_M$. Dans cette notation, X_i et Y_i désignent des types, dont un au moins est le

type *T* étudié. Le produit cartésien X_1 x X_2 x ... x X_N correspond au domaine d'entrée de la fonction *f* (on dit aussi domaine) et le produit Y_1 x Y_2 x ... x Y_M à son domaine de sortie (on dit aussi co-domaine). Les domaines d'entrée et de sortie spécifient ainsi le profil de l'opération *f*.

Finalement, les opérations du TAD *Point* peuvent se résumer ainsi :

pointOrigine : → Point
unPoint : Réel x Réel x Couleur x Réel → Point
abscisse : Point → Réel
ordonnée : Point → Réel
couleur : Point → Couleur
taille : Point → Réel
modifierCouleur : Point x Couleur → Point
modifierTaille : Point x Réel → Point
translater : Point x Réel x Réel → Point
mettreAEchelle : Point x Réel → Point

Remarques

1) Une opération ne s'applique pas nécessairement à tous les représentants de son domaine d'entrée ; on dit alors qu'elle est partielle. Par exemple, il semble raisonnable d'imposer que la mise à l'échelle d'un point nécessite un facteur d'échelle *h* compris entre 0 et 1 pour un rétrécissement et plus grand que 1 pour un agrandissement, soit en définitive une valeur strictement positive. De même pour la taille *t* d'un point que l'on n'imagine pas négative ou nulle. De telles restrictions sur les opérations s'expriment par des préconditions. Pour le TAD *Point*, on peut définir :

Pour *p* de type *Point*, *x*, *y*, *t, h* de type *Réel* et *c* de type *Couleur* :

unPoint (x, y, c, t) **est défini ssi** t > 0
modifierTaille (p, t) **est défini ssi** t > 0
mettreAEchelle (p, h) **est défini ssi** h > 0

2) Le symbole fonctionnel d'un profil d'opération agit comme nom de fonction. Par exemple, pour *unPoint* de profil *Réel x Réel x Couleur x Réel → Point*, la notation *unPoint (1.0, 2.0, rouge, 5.0)* désigne la fonction *unPoint* appliquée aux arguments *1.0*, *2.0*, *rouge* et *5.0* de sortes respectives *Réel*, *Réel*, *Couleur* et *Réel*. C'est cette vision qui confère à la spécification du TAD un point de vue fonctionnel.

3) Le plus souvent, plusieurs types cibles ne sont pas souhaitables pour un co-domaine d'opération, car il faut tôt ou tard les dissocier en définissant des fonctions de projection supplémentaires.

2.2. Propriétés

Une propriété permet d'interpréter formellement une opération. Elle prend la forme d'une équation devant être satisfaite.

Ainsi, malgré notre effort d'abstraction, le TAD *Point* d'écran n'est pas clairement défini car chaque opération peut s'interpréter de différentes façons. En effet, si l'on considère par exemple l'opération *taille*, aucune information n'est précisée par rapport à son calcul lorsque le point est mis à l'échelle. De même, pour la constante *pointOrigine*, les attributs *abscisse*, *ordonnée*, *couleur* et *taille* ne sont pas précisés.

Il manque donc la sémantique de chacune des opérations. Une des façons de procéder consiste à associer des propriétés aux opérations, notées par la suite *P*, en les composant entres-elles. Par exemple, que deviennent la taille, l'abscisse et l'ordonnée d'un point mis à une certaine échelle ? Ces propriétés sont des axiomes complétant la description syntaxique des opérations ; on parle alors de Types Abstraits Algébriques (TAA). Chaque axiome se suffit à lui-même ; cependant, pour plus de clarté, nous l'accompagnons d'une courte traduction en italiques si nécessaire.

Pour écrire les axiomes spécifiant le TAD, on commence par répartir en deux groupes les opérations : le premier est celui des *générateurs* qui construisent un représentant du TAD étudié et le second est celui des *observateurs* retournant une valeur observable d'un autre type. Parmi les générateurs, on identifie les *générateurs de base* permettant a priori d'engendrer tous les représentants de la sorte étudiée ; les autres sont appelés *générateurs secondaires*. Les axiomes s'obtiennent en appliquant les observateurs et générateurs secondaires aux générateurs de base. On peut aussi définir un générateur secondaire à partir d'un générateur de base.

Pour le type abstrait *Point*, on peut répartir comme suit ses opérations :

type d'opération	*opérations*
générateurs de base[4]	unPoint
générateurs secondaires	pointOrigine, modifierCouleur, modifierTaille, translater, mettreAEchelle
observateurs	abscisse, ordonnée, couleur, taille

Considérant que l'opération *unPoint* permet de définir tous les points, on définit :

Pour *c*, *c1*, *c2* de type *Couleur*, *x*, *y*, *h*, *t*, *t1*, *t2*, *tx* et *ty* de type *Réel :*

(P1) pointOrigine = unPoint (0.0, 0.0, noir, 1.0)
informations caractérisant le point à l'origine

(P2) abscisse (unPoint (x, y, c, t)) = x
(P3) ordonnée (unPoint (x, y, c, t)) = y
(P4) couleur (unPoint (x, y, c, t)) = c
(P5) taille (unPoint (x, y, c, t)) = t

[4] Un TAD peut posséder plusieurs générateurs de base.

(P6)	modifierCouleur (unPoint (x, y, c1, t), c2) = unPoint (x, y, c2, t)
(P7)	modifierTaille (unPoint (x, y, c, t1), t2) = unPoint (x, y, c, t2)
(P8)	translater (unPoint (x, y, c, t), tx, ty) = unPoint (x + tx, y + ty, c, t) *c'est la définition de translater !*
(P9)	mettreAEchelle (unPoint (x, y, c, t), h) = unPoint (x * h, y * h, c, t * h) *c'est la définition de mettre à l'échelle !*

L'annexe 1 fournit la spécification complète du TAD *Point* (opérations, préconditions et propriétés).

Remarques

1) L'opération *pointOrigine* est définie sémantiquement par équivalence au générateur de base *unPoint* via la propriété P1. Il en est de même pour les propriétés P6 à P9.

2) C'est le concepteur du type abstrait de données qui fournit l'ensemble des propriétés du type en termes d'axiomes. On aurait ainsi pu définir la propriété P7 :

(P7)	modifierTaille (unPoint (x, y, c, t1), t2) = unPoint (x, y, c, t1 + t2)

3) L'écriture des propriétés suppose que les préconditions des opérations sont respectées.

4) Les types élémentaires (Entier, Réel, Caractère…) sont des types abstraits de données particuliers qui existent déjà dans la plupart des langages de programmation. Il n'est donc pas nécessaire de les représenter. Par exemple, pour le type élémentaire *Entier*, on définit les deux opérations suivantes qui désignent respectivement la constante entière de valeur 0 (zéro) et l'addition de deux entiers (plus) :

zéro : → Entier plus : Entier x Entier → Entier

Les opérations sur les entiers ne se résument pas aux seules opérations présentées ici, mais elles traduisent assez fidèlement l'analogie qu'on établira entre types abstraits déjà définis et types abstraits de l'utilisateur.

5) De même, un type composé peut se spécifier par un TAD. Par exemple, *Tableau[T]*, le type abstrait de données définissant un tableau[5] d'éléments de type *T*.

[5] *Tableau* est un TAD générique qui définit une famille de tableaux. Il possède en paramètre un type non spécifié, appelé ici *T* et se note *Tableau[T]*. Le type *T* désigne le type des éléments du tableau et correspond à un paramètre formel de généricité. De la définition générale de *Tableau[T],* on pourra ainsi décliner les types *Tableau[Entier], Tableau[Chaîne]…* en remplaçant simplement *T* par le type souhaité, ici *Entier*, *Chaîne*… Dans l'écriture *Tableau[Entier]*, on dit que le type *Entier* est le paramètre effectif de généricité qui se substitue à *T*. Cette opération de substitution est appelée instanciation du type générique.

On peut ainsi imaginer les opérations suivantes sur les représentants de ce type *Tableau[T]* :

unTableau : Entier x Entier → Tableau[T] borneInf : Tableau[T] → Entier borneSup : Tableau[T] → Entier ième : Tableau[T] x Entier → T changerIème : Tableau[T] x Entier x T → Tableau[T]

L'opération *ième (tab, i)* correspond à un accès en consultation à l'élément noté traditionnellement *tab[i]* ; de même *changerIème (tab, i, e)* traduit un accès en modification à ce même élément, soit *tab[i]* <- *e*. Noter aussi que les opérations *borneInf (tab)* et *borneSup (tab)* expriment respectivement les bornes inférieure et supérieure du domaine d'indice du tableau *tab*.

Nous écrivons ci-dessous les préconditions que doivent vérifier les représentants du type *Tableau[T]* :

Pour *t* de type *Tableau[T]*, *i*, *inf*, *sup* de type *Entier* et *e* de type *T* : unTableau (inf, sup) **est défini ssi** inf ≤ sup ième (t, i) **est défini ssi** borneInf (t) ≤ i ≤ borneSup (t) changerIème (t, i, e) **est défini ssi** borneInf (t) ≤ i ≤ borneSup (t)

Dans les axiomes qui suivent et qui définissent les opérations sur un tableau, la forme ***si … alors … sinon*** dénote une équation conditionnelle :

Pour *t* de type *Tableau[T]*, *inf*, *sup*, *i*, *j* de type *Entier*, et *e* de type *T* : (P1) borneInf (unTableau (inf, sup)) = inf (P2) borneInf (changerIème (t, i, e)) = borneInf (t) (P3) borneSup (unTableau (inf, sup)) = sup (P4) borneSup (changerIème (t, i, e)) = borneSup (t) (P5) ième (changerIème (t, i, e), j) = **si** i = j **alors** e **sinon** ième (t, j)

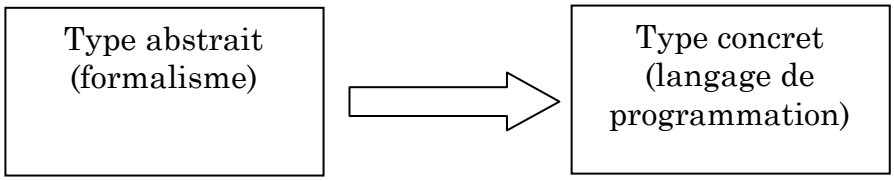
3. DU TYPE ABSTRAIT AU TYPE CONCRET

Définition

Un type concret est une incarnation des opérations d'un type abstrait dans un langage de programmation ; il doit en respecter la spécification fonctionnelle. A une spécification fonctionnelle d'un type abstrait peut correspondre un ou plusieurs types

concrets, dont les mises en œuvre ou implémentations seront plus ou moins efficaces, plus ou moins lisibles, plus ou moins maintenables... L'implémentation peut aussi dépendre de l'expressivité du langage support.

Coder un type abstrait nécessite de changer d'espace technologique comme le suggère la figure ci-dessous :



- opérations
- propriétés

- types (au sens programmation)
- sous-programmes

3.1. Etapes de transformation

En programmation, le type abstrait devient donc concret. Soient T_a un type abstrait de données et T_c le type concret correspondant à T_a dans le langage de programmation. La transformation de T_a en T_c s'effectue en trois étapes et nécessite de définir dans le langage de programmation :

1. un en-tête de sous-programme pour chaque opération de T_a,
2. une représentation physique pour coder en mémoire un représentant de T_a,
3. un corps de sous-programme pour chaque en-tête.

L'étape 1 fait l'objet de cette fin de chapitre. Les étapes 2 et 3 seront détaillées respectivement dans les deux chapitres suivants. Le premier est plus spécifiquement consacré au choix d'une représentation en mémoire des valeurs caractérisant le type abstrait (étape 2) ; le second concerne l'écriture des opérations compte tenu du choix de représentation physique (étape 3).

3.2. En-têtes des sous-programmes

Les opérations d'une spécification fonctionnelle d'un TAD sont en réalité des fonctions mathématiques. Une traduction littérale des opérations imposerait l'usage exclusif de fonctions en programmation, interdisant donc les procédures.

Exemple

L'opération *translater : Point x Réel x Réel → Point* devrait se programmer par une fonction munie de trois paramètres, l'un de type *Point* et deux de type *Réel*, et qui construit un nouveau *Point* :

```
-- construit un nouveau point par translation du point p
-- de tx (abscisse) et ty (ordonnée)
fonction translater
      (entrée p <Point>, entrée tx <Réel>, entrée ty <Réel>)
retourne <Point> ;
```

En conséquence, *translater* ainsi défini produit un nouveau point, tout en laissant le point transmis par l'appelant inchangé...

En pratique, ce point de vue est trop restrictif et il est préférable, tout en respectant la spécification fonctionnelle, d'accepter une modification des paramètres effectifs grâce au mode mise à jour. On parle alors de mutation ou d'effet de bord ce qui est impossible avec des fonctions puisque les paramètres sont toujours transmis en mode entrée. En fait, il faut opérer un passage du fonctionnel à l'impératif.

Dans ce qui suit, on donne quelques indications pour établir un choix entre procédure ou fonction. Il se base essentiellement sur une classification des opérations (générateurs de base, générateurs secondaires et observateurs) d'un TAD ; on distingue :

- les opérations de construction,
- les opérations de consultation,
- les opérations de modification,
- les opérations d'évaluation.

3.2.1. Les opérations de construction

Définition

On les appelle aussi *constructeurs*. Leur rôle est de créer des représentants du type abstrait T_a, à partir de représentants d'autres types (éventuellement).

Caractéristique

Pour une telle opération, le type T_a n'apparaît que dans le domaine de sortie de l'opération. Ce sont donc en principe des générateurs de base. Lorsqu'aucun type n'apparaît dans le domaine d'entrée, on considère qu'il s'agit d'une constante du type abstrait.

Règle

En général, une opération de construction dans T_a se traduit par une fonction du langage algorithmique dans T_c.

Exemples

1) Le constructeur *unPoint* de *Point* permet de construire un nouveau point à partir d'une abscisse, d'une ordonnée, d'une couleur et d'une taille. Il a pour profil dans T_a :

unPoint : Réel x Réel x Couleur x Réel → Point

et dans T_c :

```
-- construit un point d'abscisse x, d'ordonnée y, de couleur c et de taille t
-- nécessite t > 0.0
fonction unPoint
        (entrée x <Réel>, entrée y <Réel>,
        entrée c <Couleur>, entrée t <Réel>)
retourne <Point>
déclenche tailleInvalide ;
```

2) Avec le type *Point*, on peut imaginer la constante *pointOrigine*, dont l'en-tête dans le type concret T_c sera :

```
-- constante point à l'origine
fonction pointOrigine retourne <Point> ;
```

Remarques

1) Le non respect d'une précondition d'une opération du type abstrait se traduit le plus souvent par un déclenchement d'exception dans le type concret. Ainsi, pour le constructeur *unPoint*, la taille *t* doit être strictement positive. Si ce n'est pas le cas, une exception *tailleInvalide* sera délenchée.

2) L'opération *pointOrigine* est qualifiée de constante car on retrouve la notion habituelle de constante dans son usage. Un client doit ainsi pouvoir ainsi écrire, si *p* est un point de l'application : *p* <- *pointOrigine*.

3) Il n'est pas indispensable de spécifier dans le TAD *Point* la constante *pointOrigine* car un utilisateur du TAD peut facilement la déduire de l'opération *unPoint*.

4) Selon le statut d'exportation donné à l'affectation (<-), il est parfois nécessaire d'incarner un constructeur par une procédure. Cet effet peut se conjuguer avec la nécessité de communiquer l'identité d'une variable en tant que paramètre effectif d'une transmission en mode mise à jour.

3.2.2. Les opérations de consultation

Définition

On les appelle aussi opérations d'accès ou *accesseurs*. Leur rôle consiste à fournir des valeurs caractérisant une variable de type T_a, sans modification de celle-ci.

Caractéristique

Le type T_a apparaît uniquement dans le domaine d'entrée de l'opération. Ce sont donc des observateurs.

Règle

Une opération de consultation de T_a s'exprime toujours par une fonction du langage algorithmique dans T_c.

Exemple

L'opération *ordonnée* de T_a de permet de consulter l'ordonnée d'un point :

```
ordonnée : Point → Réel
```

Elle se traduit dans T_c par :

```
-- ordonnée du point p
fonction ordonnée (entrée p <Point>) retourne <Réel> ;
```

3.2.3. Les opérations de modification

Définition

Leur rôle est de modifier des éléments de type T_a à partir d'éléments de type T_a déjà existants et d'éléments d'autres types (éventuellement).

Caractéristique

Le type T_a apparaît à la fois dans le domaine d'entrée et le domaine de sortie de l'opération. Ce sont donc des générateurs de base ou secondaires.

Règle

Une telle opération de T_a se traduit par une procédure dans T_c. La modification attendue pour le représentant du type est mise en œuvre par le mode mise à jour.

Exemple

Soit dans T_a :

```
translater : Point x Réel x Réel → Point
```

Cette opération se traduit par une procédure dans T_c, en considérant que le même point est à la fois en entrée et en sortie de l'opération, ce qui revient à un mode mise à jour. On déclarera :

```
-- translate le point p de tx (abscisse) et ty (ordonnée)
procédure translater
      (màj p <Point>, entrée tx <Réel>, entrée ty <Réel>) ;
```

3.2.4. Les opérations d'évaluation

Définition

Leur rôle est de créer de nouveaux éléments de type T_a à partir d'éléments de type T_a déjà existants et d'éléments d'autres types (éventuellement).

Caractéristique

Le type Ta apparaît à la fois dans le domaine d'entrée et le domaine de sortie l'opération. Ce sont donc des générateurs de base ou secondaires.

Règle

Une telle opération de T_a se traduit par une fonction dans T_c. Deux représentants du type abstrait sont manipulés : celui en entrée de la fonction et celui en tant que résultat produit.

Exemple

L'opération *translater*, définie précédemment comme une opération de modification, peut être considérée comme une fonction d'évaluation ; elle fournira alors un nouveau point et son profil :

```
translater : Point x Réel x Réel → Point
```

se traduira maintenant par une fonction du langage algorithmique dans T_c :

```
-- construit un nouveau point par translation du point p
-- de tx (abscisse) et ty (ordonnée)
fonction nouveauPointTranslaté
      (entrée p <Point>, entrée tx <Réel>, entrée ty <Réel>)
retourne <Point> ;
```

Remarques :

1) Noter le changement de nom de l'opération afin de tenir compte du respect des conventions de nommage des procédures et des fonctions.

2) Le choix dans T_c entre opération de modification et d'évaluation résulte pour l'essentiel du style d'utilisation pour un client : impératif dans le cas d'une opération de modification et fonctionnel dans le cas d'une fonction.

3.3. Du type abstrait *Point* à son incarnation en type concret

Une spécification possible du TAD *Point* en programmation est résumée par le tableau suivant. Ce tableau précise le type d'opération (construction, consultation, modification ou évaluation) et le type de sous-programme (procédure ou fonction) de

chaque opération dans le type concret. Il illustre donc le passage du type abstrait au type concret.

opération	*type d'opération (type abstrait)*	*type d'opération (type concret)*	*type de sous-programme (type concret)*
pointOrigine	générateur secondaire	construction	fonction
unPoint	générateur de base	construction	fonction
abscisse	observateur	consultation	fonction
ordonné	observateur	consultation	fonction
couleur	observateur	consultation	fonction
taille	observateur	consultation	fonction
modifierCouleur	générateur secondaire	modification	procédure
modifierTaille	générateur secondaire	modification	procédure
translater	générateur secondaire	modification	procédure
mettreAEchelle	générateur secondaire	modification	procédure

D'après les règles qui précèdent, on peut aussi spécifier les types des paramètres des sous-programmes car ils découlent directement du profil des opérations du TAD. Voici les interfaces de quelques opérations du type abstrait *Point* :

opération	*nature du sous-programme*	*type du résultat (si fonction)*	*type et mode de transmission des paramètres*		
			Entrée	*Sortie*	*Mise à jour*
pointOrigine	fonction	Point	3 Entier (s)	_	_
unPoint	fonction	Point	_	_	_
abscisse	fonction	Réel	Point	_	_
couleur	fonction	Couleur	Point	_	_
modifierTaille	procédure	-	Réel	_	Point
mettreAEchelle	procédure	-	Réel	_	Point

On en déduit dès lors la spécification du type concret impératif *Point* en langage algorithmique. Cette version du TAD est dite impérative car elle propose pour *modifierTaille* et *mettreAEchelle* une mise à jour du point transmis en paramètre ; dans le cas où ces opérations seraient définies par des fonctions, le TAD serait dit fonctionnel.

4. SPECIFICATION D'UN TYPE CONCRET

La spécification d'un type concret, encore appelée spécification algorithmique du type abstrait[6], regroupe l'ensemble des en-têtes des opérations du type concret et leurs propriétés, seule partie connue du client. Le contrôle se fait alors à travers une interface bien délimitée et clairement définie. La spécification du type concret

[6] En programmation, on parle aussi de type abstrait, en référence au formalisme… y compris pour un type concret.

supporte directement le principe d'encapsulation des données ; c'est l'interface visible à l'utilisateur, son mode d'emploi.

Exemples

1) Pour le TAD *Point*, on spécifie :

```
importer Couleur ;

-- constante point à l'origine
fonction pointOrigine retourne <Point> ;

-- construit un point d'abscisse x, d'ordonnée y, de couleur c et de taille t
-- nécessite t > 0.0
fonction unPoint
        (entrée x <Réel>, entrée y <Réel>,
        entrée c <Couleur>, entrée t <Réel>)
retourne <Point>
déclenche tailleInvalide ;

-- abscisse du point p
fonction abscisse (entrée p <Point>) retourne <Réel> ;

-- ordonnée du point p
fonction ordonnée (entrée p <Point>) retourne <Réel> ;

-- couleur du point p
fonction couleur (entrée p <Point>) retourne <Couleur> ;

-- taille du point p
fonction taille (entrée p <Point>) retourne <Réel> ;

-- affecte la couleur c au point p
procédure modifierCouleur (màj p <Point>, entrée c <Couleur>) ;

-- affecte la taille t au point p
-- nécessite t > 0.0
procédure modifierTaille (màj p <Point>, entrée t <Réel>)
déclenche tailleInvalide ;

-- translate le point p de tx (abscisse) et ty (ordonnée)
procédure translater
        (màj p <Point>, entrée tx <Réel>, entrée ty <Réel>) ;

-- met au facteur d'échelle h le point p
-- nécessite h > 0.0
procédure mettreAEchelle (màj p <Point>, entrée h <Réel>) ;
déclenche échelleInvalide ;
```

```
--    pointOrigine = unPoint (0.0, 0.0, noir, 1.0)
--    abscisse (unPoint (x, y, c, t)) = x
--    ordonnée (unPoint (x, y, c, t)) = y
--    couleur (unPoint (x, y, c, t)) = c
--    taille (unPoint (x, y, c, t)) = t
--    modifierCouleur (unPoint (x, y, c1, t), c2) = unPoint (x, y, c2, t)
--    modifierTaille (unPoint (x, y, c, t1), t2) = unPoint (x, y, c, t2)
--    translater (unPoint (x, y, c, t), tx, ty) = unPoint (x + tx, y + ty, c, t)
--    mettreAEchelle (unPoint (x, y, c, t), h) = unPoint (x * h, y * h, c, t * h)
```

2) La spécification des opérations du type concret générique *Tableau[T]* s'écrit :

```
-- définit les bornes inférieure inf et supérieure sup d'un tableau
-- nécessite inf <= sup
fonction unTableau (entrée inf <Entier>, entrée sup <Entier>)
retourne <Tableau[T]>
déclenche tailleInvalide ;

-- fournit la borne inférieure du tableau t
fonction borneInf (entrée t <Tableau[T]>) retourne <Entier> ;

-- fournit la borne supérieure du tableau t
fonction borneSup (entrée t <Tableau[T]>) retourne <Entier> ;

-- fournit l'élément d'indice i du tableau t
-- nécessite borneInf (t) <= i <= borneSup (t)
fonction ième (entrée t <Tableau[T]>, entrée i <Entier>)
retourne <T>
déclenche indiceInvalide ;

-- affecte e à l'élément d'indice i du tableau t
-- nécessite borneInf (t) <= i <= borneSup (t)
procédure changerIème
      (màj t <Tableau[T]>, entrée i <Entier>, entrée e <T>)
déclenche indiceInvalide ;


--    borneInf (unTableau (inf, sup)) = inf
--    borneInf (changerIème (t, i, e)) = borneInf (t)
--    borneSup (unTableau (inf, sup)) = sup
--    borneSup (changerIème (t, i, e)) = borneSup (t)
--    ième (changerIème (t, i, e), j) =
--          si i = j alors e sinon ième (t, j)
```

Remarques

1) Par la clause d'importation (mot-clé ***importer***), la spécification du type *Point* a accès à celle de *Couleur*.

2) Selon le statut accordé aux opérations d'affectation (<-) et de comparaison (= et /=) mettant en jeu deux représentants du type abstrait, la spécification algorithmique d'un type pourra être reconsidérée. Ce point sera évoqué au chapitre 3 relatif à l'implémentation d'un type abstrait.

5. UTILISATION D'UN TAD

Pour écrire une application donnée, un programmeur peut faire référence à un ou plusieurs types abstraits (par abus de langage par rapport à type concret), comme il fait référence à une bibliothèque de sous-programmes (voir cours d'algorithmique). Il est alors client des TAD qu'il exploite. Pour utiliser un type abstrait de données, il suffit d'en connaître sa spécification, c'est-à-dire son mode d'emploi.

Un client du TAD *T* peut :

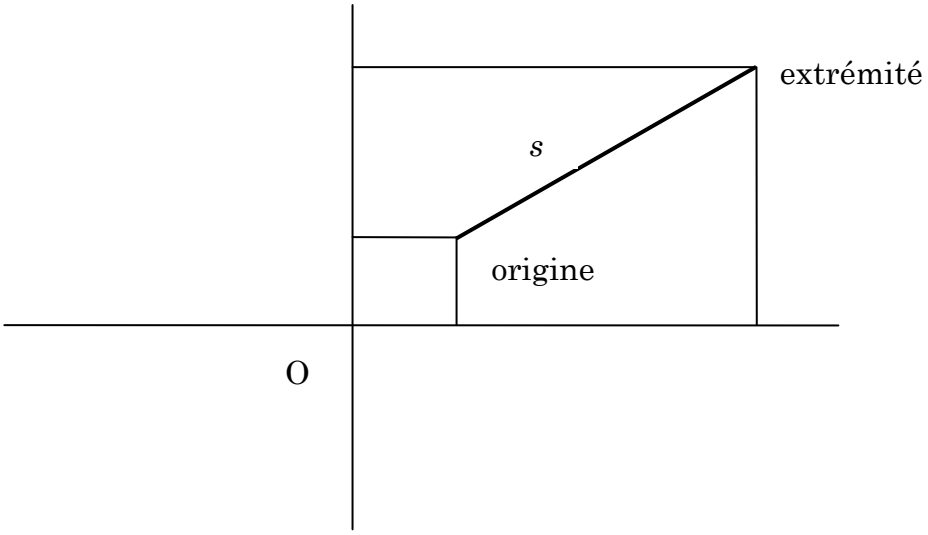
- utiliser le type *T* pour déclarer des variables et des paramètres,
- utiliser le type *T* pour définir de nouveaux types,
- appeler les sous-programmes du type *T*.

Exemple

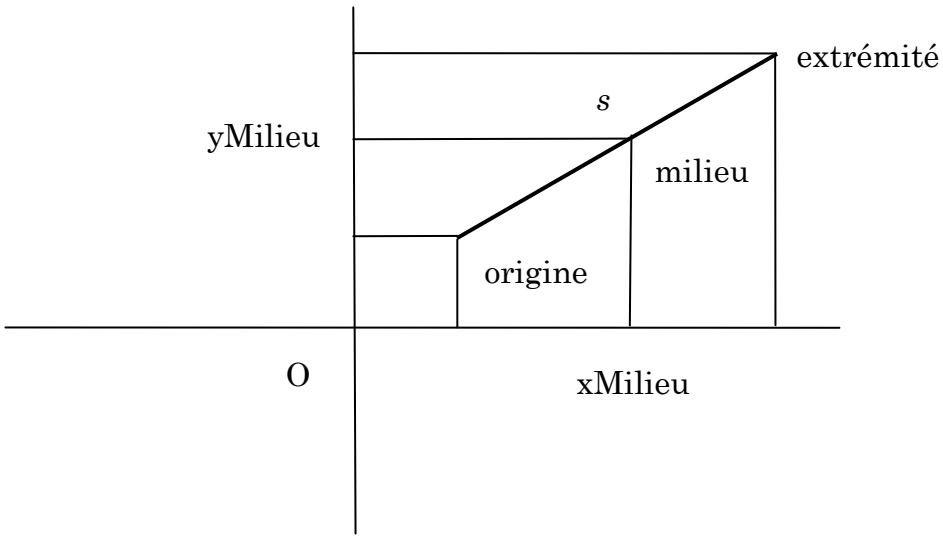
Supposons une application graphique devant manipuler des figures géométriques. On décide de modéliser un segment dans le plan par deux points, ce qui se traduira par :

```
-- définition du type Segment
type Segment : enregistrement
      origine <Point>,          -- origine du segment
      extrémité <Point> ;       -- extrémité du segment
```

On peut représenter graphiquement un segment *s* de la façon suivante :



Soit maintenant à écrire un sous-programme de l'application permettant de construire un nouveau point *milieu*, milieu du segment *s*, de couleur noire et de taille 1.0 :



On définira :

```
-- point milieu du segment s
fonction pointMilieu (entrée s <Segment>) retourne <Point>

glossaire
    xMilieu <Réel > ;        -- abscisse du point milieu
    yMilieu <Réel > ;        -- ordonnée du point milieu

début
    xMilieu <- (abscisse (s.origine) + abscisse (s.extrémité)) / 2 ;
    yMilieu <- (ordonnée (s.origine) + ordonnée (s.extrémité)) / 2 ;
    retourner (unPoint (xMilieu, yMilieu, noir, 1.0)) ;
fin
```

Remarques

1) Grâce à la notion de type abstrait, nous sommes capables de manipuler des points sans en connaître leur représentation physique.

2) Le type *Segment* pourrait être un type abstrait de données. Il conviendrait alors de fournir ses opérations et ses propriétés, pourquoi pas le point milieu d'un segment, ... De même, nous pourrions spécifier le type abstrait *Couleur* utilisé par *Point*.

3) L'exploitation des opérations d'un TAD par un client nécessite d'importer sa spécification. Il mentionne la dépendance de son application vis-à-vis du TAD par une clause ***importer*** (voir cours d'algorithmique). Pour le calcul du point milieu d'un segment, on obtient ainsi l'architecture :

```
importer Point, Couleur ;

type Segment : … ;

fonction pointMilieu …

programme exemple
début
    …              -- appel de la fonction pointMilieu
fin
```

6. PROCESSUS D'ELABORATION D'UN TAD

Le processus d'élaboration d'un TAD obéit aux étapes suivantes :

1. énumérer les opérations du TAD,
2. pour chaque opération, spécifier son profil,
3. énoncer les propriétés des opérations du TAD,
4. associer un en-tête de sous-programme à chaque opération,
5. choisir une représentation physique pour coder un représentant du TAD,
6. coder les corps des sous-programmes avec la représentation choisie.

Les étapes 1 à 3 sont relatives au type abstrait alors que les étapes 4 à 6 concernent le type concret.

Les propriétés permettent notamment :

1. de définir la sémantique du type,
2. d'aider au codage des opérations du type concret,
3. de construire les jeux de tests fonctionnels des opérations.

Chapitre 2 : Représentation physique d'un TAD

L'implémentation d'un TAD réalise les fonctionnalités définies par sa spécification. Pour cela, elle comprend l'ensemble des déclarations et des sous-programmes nécessaires à la mise en œuvre de la spécification. Cette partie est cachée à un utilisateur du TAD qui n'a pas à connaître les détails d'implémentation, comme par exemple la représentation d'un point en mémoire. Une implémentation de TAD peut être remplacée par une implémentation équivalente réalisant les mêmes fonctionnalités et donnant accès aux mêmes ressources.

En préambule du codage des opérations d'un type, se pose la question du choix d'une représentation en mémoire d'un représentant du type. Deux grandes solutions, éventuellement combinées, existent pour représenter physiquement une donnée en mémoire : l'une est statique, l'autre dynamique.

1. REPRESENTATION STATIQUE

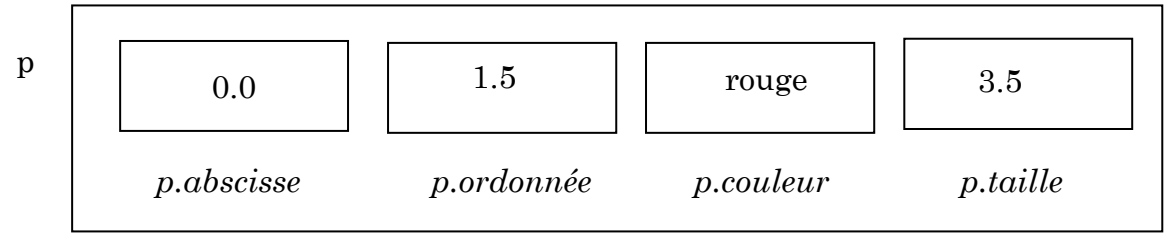
Avec une représentation statique, c'est le compilateur du langage de programmation qui se charge de réserver l'espace nécessaire pour mémoriser la donnée, à partir de la définition du type fournie par le programmeur. Par exemple, la déclaration d'un représentant du type, c'est-à-dire d'une variable, entraîne automatiquement une réservation de place ; le type correspond alors au modèle de la donnée. Les principaux types statiques sont les tableaux et les enregistrements.

Exemple

On peut représenter statiquement le type *Point* par un enregistrement en écrivant :

```
-- définition du type Point
type Point : enregistrement
      abscisse <Réel>,          -- abscisse du point
      ordonnée <Réel>,          -- ordonnée du point
      couleur <Couleur>,        -- couleur du point
      taille <Réel> ;           -- taille du point
```

Un point *p* d'abscisse 0.0, d'ordonnée 1.5, de couleur *rouge* et de taille 3.5 se représentera en mémoire par :



Avec cette spécification, les principales opérations du TAD *Point* s'écrivent :

a) *pointOrigine*

```
-- constante point à l'origine
fonction pointOrigine retourne <Point>

glossaire
        p <Point> ;             -- point retourné par la fonction

début
        p.abscisse <- 0.0 ;
        p.ordonnée <- 0.0 ;
        p.couleur <- noir ;
        p.taille <- 1.0 ;
        retourner (p) ;
fin
```

Dans cette écriture du corps de la fonction *pointOrigine*, on suppose que la constante *noir* appartient à l'ensemble des valeurs du type *Couleur*.

b) *ordonnée*

```
-- ordonnée du point p
fonction ordonnée (entrée p <Point>) retourne <Réel>

début
        retourner (p.ordonnée) ;
fin
```

c) *translater*

On traduit simplement la propriété P8 du Type Abstrait :

```
-- translate le point p de tx (abscisse) et ty (ordonnée)
procédure translater
        (màj p <Point>, entrée tx <Réel>, entrée ty <Réel>)

début
        p.abscisse <- p.abscisse + tx ;
        p.ordonnée <- p.ordonnée + ty ;
fin
```

Les autres opérations de la spécification fonctionnelle du type abstrait de données *Point* se déduisent facilement de ces trois écritures représentatives. Les annexes 2 et 3 précisent en détail la programmation des opérations du TAD *Point* en représentation statique (hormis l'affectation (<-) et l'égalité (=)).

2. REPRESENTATION DYNAMIQUE

Avec une représentation dynamique, c'est le programmeur qui gère explicitement la représentation de la donnée en mémoire à l'exécution. Un mécanisme d'allocation dynamique de place mémoire à la demande lui permet ainsi de réserver l'espace associé à la donnée. Cette allocation de place est réalisée dans une zone mémoire de la machine, appelée tas, accédée par des pointeurs ou variables accès. Les pointeurs permettent donc d'adapter la taille d'une structure de données au volume réel d'information à traiter.

2.1. Notion de pointeur

Définition

Un pointeur (ou variable accès) est une variable élémentaire (statique) pouvant prendre pour valeur une adresse d'une donnée en mémoire (dans le tas). Auparavant, il faudra déclarer la variable pointeur ; pour cela, en bon programmeur, on utilise la définition de type ***pointeur sur*** qui explicite le type de zone pointée.

Exemples

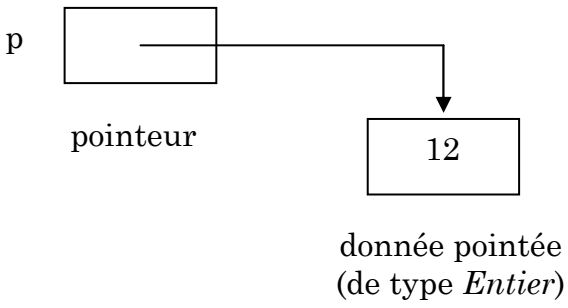
1) Soit le type *PtrEntier* définissant un type pointeur sur un entier :

```
-- définition du type PtrEntier
type PtrEntier : pointeur sur <Entier> ;
```

et une variable *p* de ce type :

```
glossaire
    -- déclaration du pointeur p sur un entier
    p <PtrEntier> ;
```

En cours d'exécution, on pourra obtenir la configuration suivante :



2) Soit le type *Point* désignant l'adresse d'un enregistrement *EnrPoint* dans le tas. On définira donc les type *Point* et *EnrPoint* :

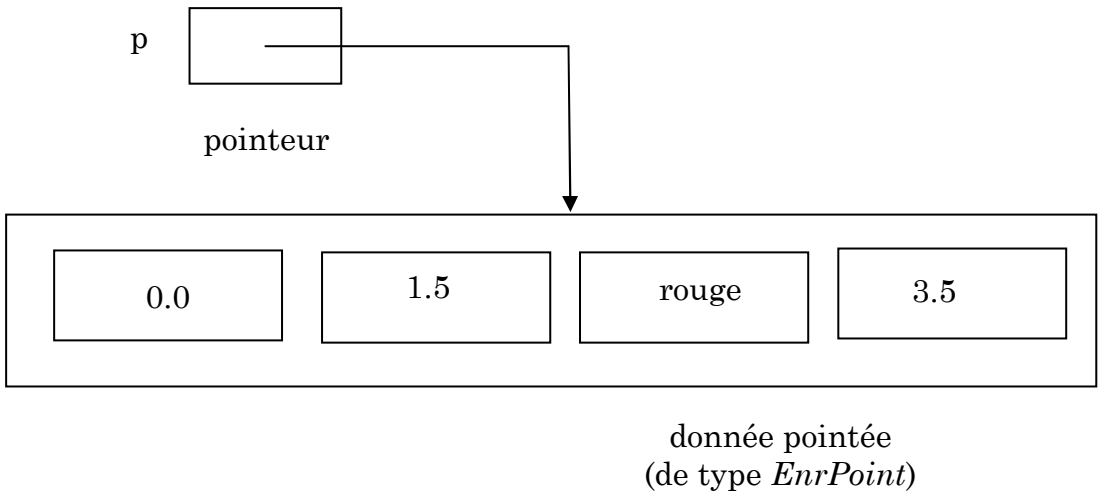
```
-- définition du type Point
type Point : pointeur sur <EnrPoint> ;

-- définition du type EnrPoint
type EnrPoint : enregistrement
        abscisse <Réel>,          -- abscisse du point
        ordonnée <Réel>,          -- ordonnée du point
        couleur <Couleur>,        -- couleur du point
        taille <Réel> ;           -- taille du point
```

Bien que type *EnrPoint* ne soit pas encore défini, on ne s'inquiétera pas en algorithmique de sa référence en avant utilisée pour définir *Point*.

En cours d'exécution, on pourra obtenir la configuration suivante pour un point *p*, déclaré comme suit :

```
glossaire
        -- déclaration de la variable p de type Point
        p <Point> ;
```

et d'abscisse 0.0, d'ordonnée 1.5, de couleur *rouge* et de taille 3.5 :



Remarque

Le contenu d'un pointeur se matérialise par une flèche vers la donnée pointée ; le programmeur manipule symboliquement cette adresse, sans réellement en connaître sa valeur.

2.2. Opérations sur les pointeurs

Il existe six catégories d'opérations applicables aux pointeurs résumées dans le tableau suivant où *p*, *p1* et *p2* sont des pointeurs :

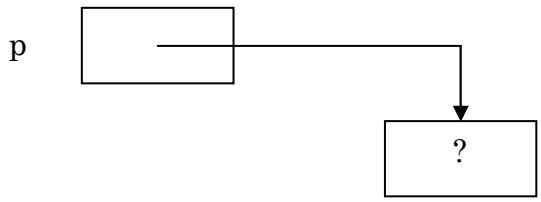
Opération	Syntaxe	Sémantique
Allocation d'espace mémoire	allouer (p)	Alloue de l'espace mémoire conformément au type référencé par *p* et affecte le pointeur *p* par l'adresse de l'espace alloué
Accès à la donnée pointée	p↑	Accède à la donnée pointée par *p*
Affectation	p1 <- p2	Affecte le contenu du pointeur *p1* par la valeur du pointeur *p2*
Egalité	p1 = p2	VRAI si le contenu du pointeur *p1* est égal au contenu du pointeur *p2*, et FAUX sinon
Différence	p1 /= p2	VRAI si le contenu du pointeur *p1* est différent du contenu du pointeur *p2*, et FAUX sinon
Libération d'espace mémoire	libérer (p)	Restitue à la mémoire la zone pointée par le pointeur *p*

2.2.1. L'allocation d'espace mémoire (*allouer*)

Cette opération alloue la mémoire correspondant au type d'objet pointé, puis retourne l'adresse de la zone allouée dans la variable pointeur. En langage algorithmique, elle s'écrit *allouer (p)* où *p* est nécessairement une variable de type ***pointeur sur***.

Exemples

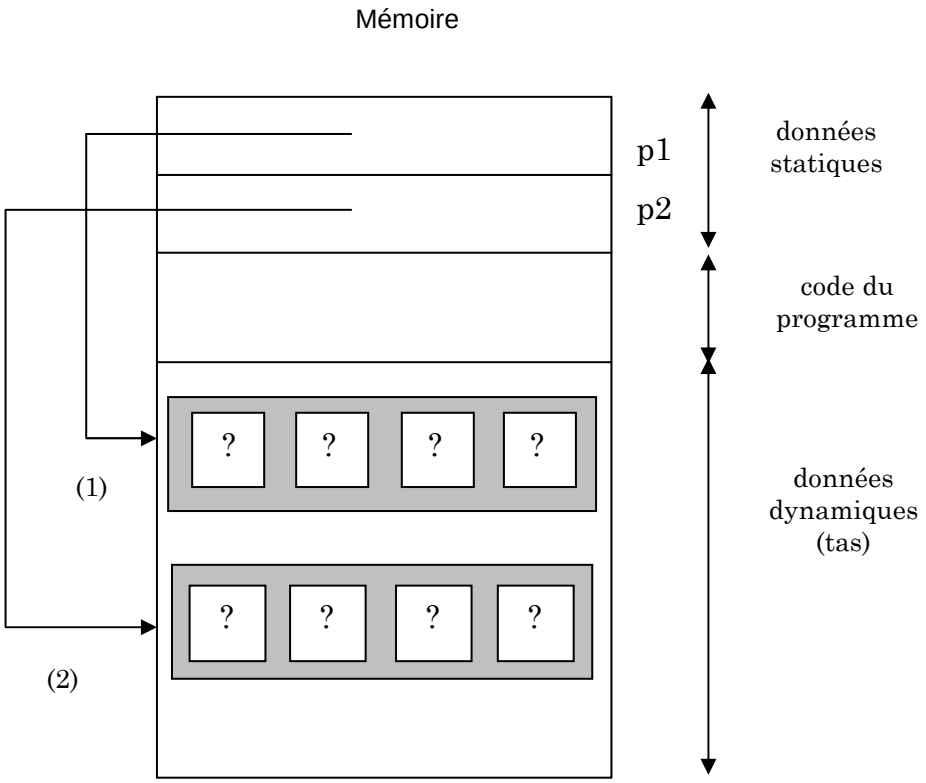
1) Soit *p* <*PtrEntier*>. Cette variable *p* désigne donc l'adresse d'un entier en mémoire. L'opération *allouer (p)* a pour effet :



La zone associée à l'entier pointé par *p* n'est pas initialisée par l'opération *allouer (p)* ; son contenu est donc indéterminé ('?').

2) Si *p1* et *p2* sont des variables de type *Point*, elles désigneront des adresses de données dynamiques de type *EnrPoint* dans le tas ; le schéma qui suit illustre les effets des deux opérations *allouer (p1)* et *allouer (p2)* sur la mémoire :

```
programme p

glossaire
      p1 <Point> ;
      p2 <Point> ;

début

      allouer (p1) ;
      -- 1


      allouer (p2) ;
      -- 2

      …

fin
```

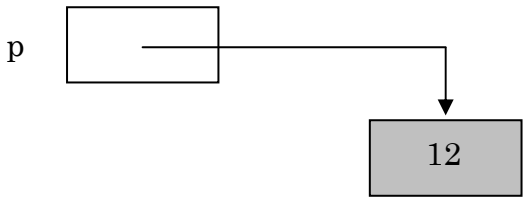


2.2.2. L'accès à la donnée pointée (↑)

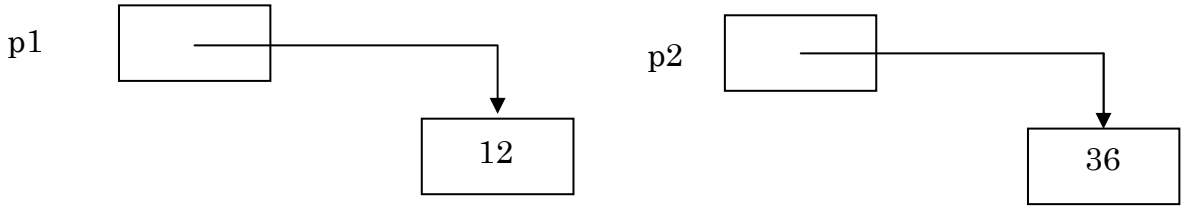
Un pointeur *p* traduit une indirection vers une donnée mémorisée dans le tas. L'accès à cette donnée se note $p\uparrow$. Ne pas confondre la variable statique *p* (pointeur) et la variable dynamique $p\uparrow$ (donnée allouée).

Exemples

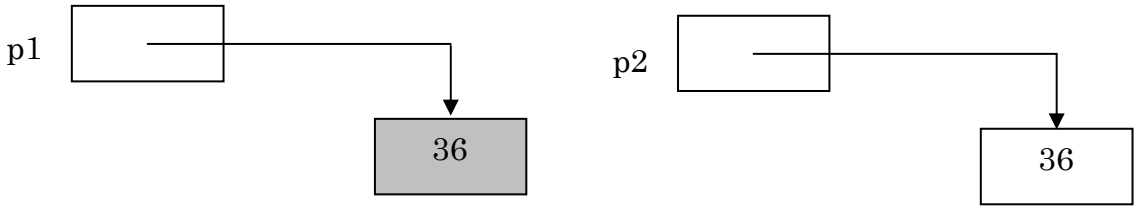
1) En considérant *p* de type *PtrEntier*, l'écriture $p\uparrow$ <- *12* affecte la valeur 12 à l'entier pointé par *p* et se traduit schématiquement par :



2) Soient *p1* et *p2* de type *PtrEntier*. En supposant la configuration suivante pour la mémoire :



l'affectation *p1↑ <- p2↑* permet d'obtenir :



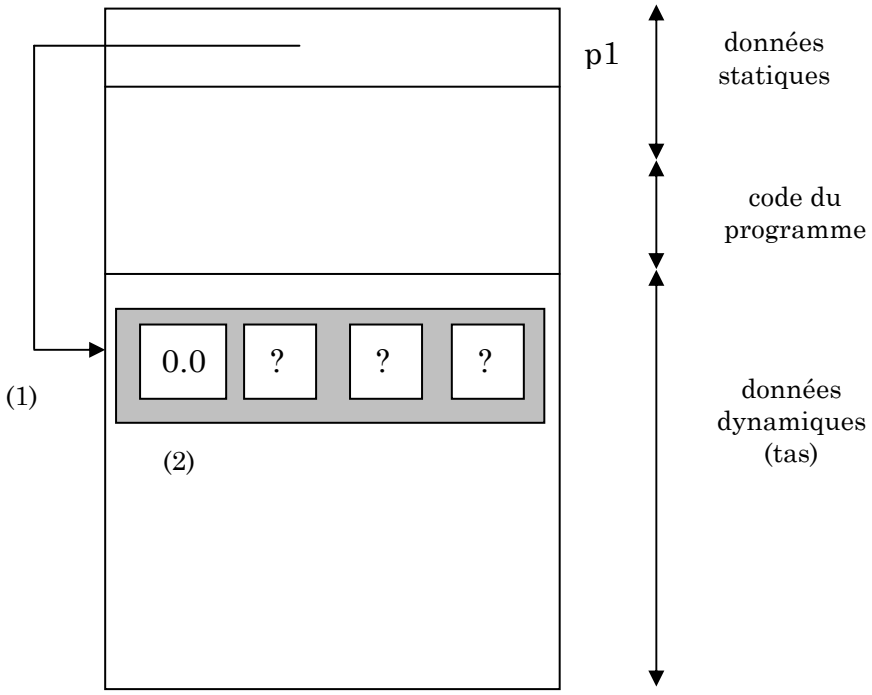
On remarque ainsi que :

- l'entier 12 n'existe plus car il a été remplacé par l'entier 36,
- l'entier 36 est référencé 2 fois : l'un par *p1* et l'autre par *p2*.

3) Si *p* désigne l'adresse d'un point mémorisé dans le tas, on accède à l'un de ses champs en le préfixant par *p↑*. En effet, par convention, *p↑* désigne l'objet pointé par *p*, à savoir ici un enregistrement de type *EnrPoint*, avec ses champs *abscisse*, *ordonnée*, *couleur* et *taille*.

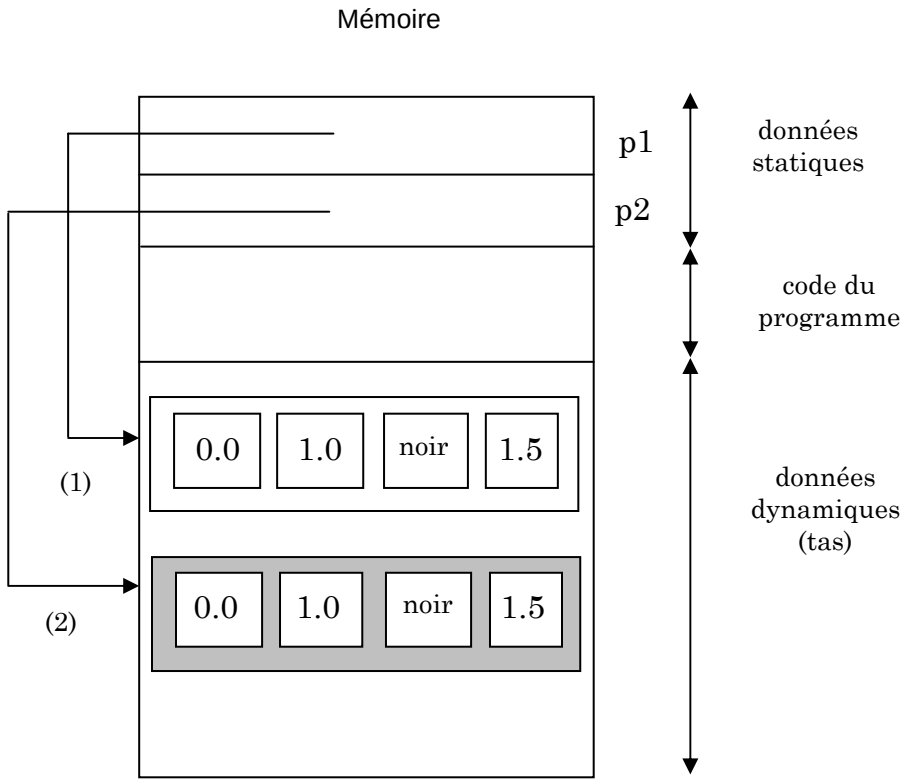
Ainsi par exemple, l'affectation *p↑.abscisse <- 0.0* affecte le réel 0.0 au point référencé par *p*. On notera la combinaison des fonctions d'accès « ↑ » des pointeurs et « . » des enregistrements. Le schéma qui suit illustre cette affectation.

```
programme p

glossaire
        p1 <Point> ;

début

        allouer (p1) ;
        -- 1

        p1↑.abscisse <- 0.0 ;
        -- 2


        …

fin
```



4) On peut également recopier un point *p1* dans un autre point *p2* en écrivant simplement *p2↑ <- p1↑*. Cette affectation se traduit en effet par une copie d'enregistrements.

```
programme p

glossaire
        p1 <Point> ;
        p2 <Point> ;

début

        allouer (p1) ;
        p1↑.abscisse <- 0.0 ;
        p1↑.ordonnée <- 1.0 ;
        …
        -- 1

        allouer (p2) ;
        p2↑ <- p1↑ ;
        -- 2

        …

fin
```



Remarques

1) La donnée pointée n'a pas de nom ! Elle a seulement un type. On ne peut y accéder que par indirection à partir d'un pointeur.

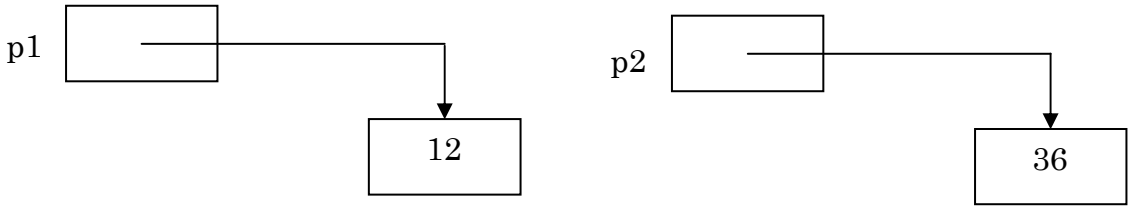
2) Les opérations applicables à la donnée pointée dépendent du type de la donnée pointée. Ce type est mentionné lors de la définition du type ***pointeur sur*** correspondant.

2.2.3. L'affectation (<-) et les comparaisons (= et /=)

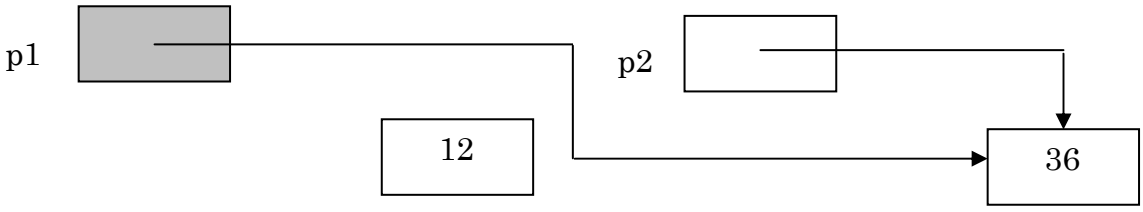
Les actions d'allocation (*allouer*) et d'accès à la donnée référencée (↑) ne sont pas les seules opérations autorisées sur des variables de type pointeur. On peut aussi affecter le contenu d'un pointeur par un autre pointeur et comparer le contenu de deux pointeurs. Attention, dans ce cas il s'agit d'affectation et de comparaison d'adresses.

Exemples

1) Soient deux variables *p1* et *p2* de type *PtrEntier* permettant respectivement l'accès aux deux entiers 12 et 36. On a donc :



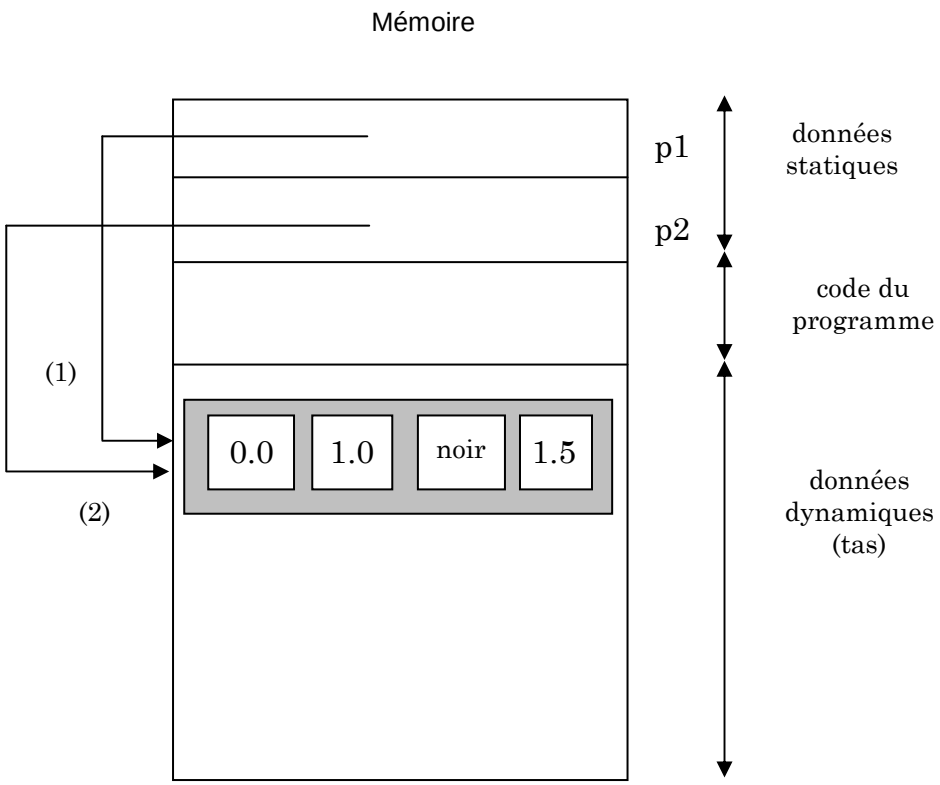
L'affectation *p1* <- *p2* a l'effet suivant sur la mémoire :



On remarque que :
- l'entier 12 n'est plus accessible car *p1* ne mémorise plus son adresse,
- on peut atteindre l'entier 36 par l'intermédiaire de *p1* ou de *p2*,
- *p1* et *p2* sont tels que la condition *p1* = *p2* est vérifiée (valeur *VRAI*).

2) En ce qui concerne l'exemple des points, l'affectation *p2* <- *p1* produit les mêmes effets et se traduit par :

```
programme p

glossaire
        p1 <Point> ;
        p2 <Point> ;

début

        allouer (p1) ;
        p1↑.abscisse <- 0.0 ;
        p1↑.ordonnée <- 1.0 ;
        …
        -- 1



        p2 <- p1 ;
        -- 2

        …

fin
```

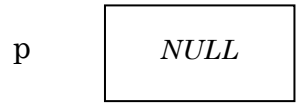


Les opérations d'affectation et de comparaison travaillent sur des adresses de données en mémoire.

Remarques

1) Un pointeur *p* peut référencer une zone mémoire du tas qu'il n'a pas préalablement réservé par *allouer (p)*.

2) La constante *NULL* (de type ***pointeur sur*** ...) est une valeur particulière à tous les pointeurs qui ne désigne aucune donnée dynamique du tas. On pourra affecter et comparer la valeur d'un pointeur à cette adresse symbolique *NULL*. Un pointeur *p* qui ne désigne aucune donnée dynamique dans le tas peut ainsi se matérialiser comme suit :



3) Si l'allocation échoue par manque de place libre en zone dynamique, l'opération *allouer (p)* affecte au pointeur *p* la valeur symbolique *NULL*[7].

4) La notation $p\uparrow$ qui permet l'accès à la donnée référencée par *p* n'a de sens que si la valeur de *p* est différente de la constante *NULL*.

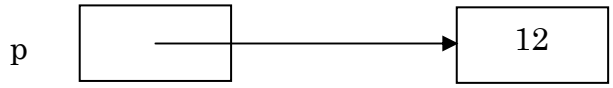
[7] Dans certains langages, l'opération lève une exception prédéfinie.

2.2.4. La libération d'espace mémoire (*libérer*)

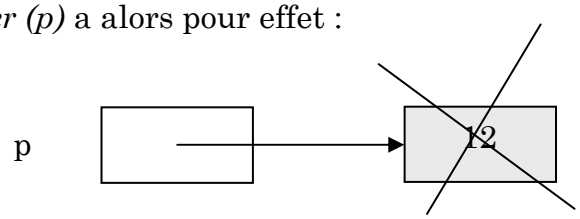
Le programmeur peut libérer l'emplacement occupé par une donnée dynamique grâce à l'action *libérer* en écrivant *libérer (p)* où *p* désigne toujours l'adresse d'une zone allouée dans le tas.

Exemples

1) Soit *p* <*PtrEntier*> et supposons que *p* désigne l'entier 12 dans le tas :



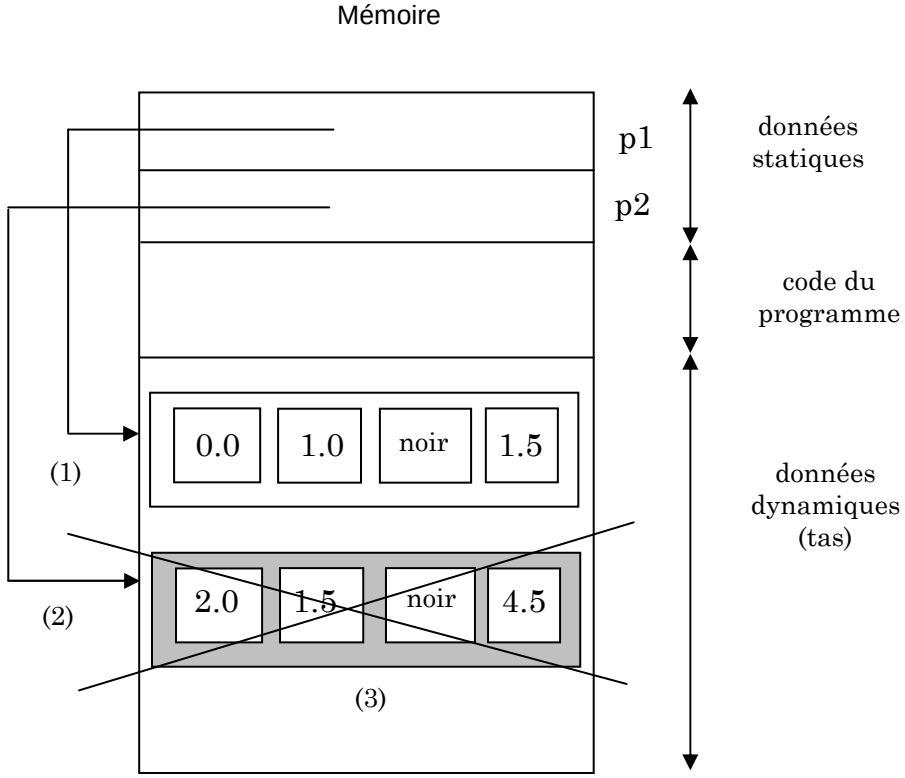
L'action *libérer (p)* a alors pour effet :



qui traduit qu'il est désormais impossible d'accéder à l'entier 12 du tas à partir de *p*.

2) Soient *p1* et *p2* de type *Point*. On peut libérer l'espace mémoire occupé par le point référencé par le pointeur *p2* en écrivant *libérer (p2)* comme dans l'exemple :

```
programme p

glossaire
        p1 <Point> ;
        p2 <Point> ;

début
        allouer (p1) ;
        p1↑.abscisse <- 0.0 ;
        p1↑.ordonnée <- 1.0 ;
        …
        -- 1

        allouer (p2) ;
        p2↑.abscisse <- 2.0 ;
        p2↑.ordonnée <- 1.5 ;
        …
        -- 2

        libérer (p2) ;
        -- 3

fin
```



Après l'action *libérer (p2),* l'emplacement occupé par le deuxième point alloué dans le tas n'est plus accessible et la zone mémoire nécessaire à sa représentation est récupérée dynamiquement.

Remarques

1) L'opération *libérer (p)* n'affecte pas le contenu du pointeur *p* qui continue de mémoriser l'adresse d'une zone… restituée à la mémoire ! Après cette opération, il convient donc impérativement d'éviter tout accès $p\uparrow$ à la donnée pointée.

2) De même, si d'autres pointeurs référencent par ailleurs la zone mémoire libérée par *p*, leur contenu, après exécution de l'instruction *libérer (p)*, n'est plus significatif.

2.3. Représentation dynamique du TAD *Point*

Par rapport à notre exemple, on représentera dynamiquement le type abstrait de données *Point* par un pointeur sur un enregistrement *EnrPoint* composé de quatre champs :

```
-- définition du type Point
type Point : pointeur sur <EnrPoint> ;

-- définition du type EnrPoint
type EnrPoint : enregistrement
        abscisse <Réel>,          -- abscisse du point
        ordonnée <Réel>,          -- ordonnée du point
        couleur <Couleur>,        -- couleur du point
        taille <Réel> ;           -- taille du point
```

Avec cette spécification pour le type *Point*, les principales opérations de la spécification fonctionnelle s'écrivent :

a) *pointOrigine*

```
-- constante point à l'origine
fonction pointOrigine retourne <Point>

glossaire
        p <Point> ;       -- point retourné par la fonction

début
        -- allouer l'espace pour mémoriser le point
        allouer (p) ;
        -- affecter les données du point
        p↑.abscisse <- 0.0 ;
        p↑.ordonnée <- 0.0 ;
        p↑.couleur <- noir ;
        p↑.taille <- 1.0 ;
```

```
        -- retourner le point
        retourner (p) ;
fin
```

On remarquera que cette fonction commence par allouer de la place en mémoire, puis affecte la zone allouée. Finalement, elle retourne l'adresse de cet objet alloué.

b) *ordonnée*

```
-- ordonnée du point p
fonction ordonnée (entrée p <Point>) retourne <Réel>

début
        retourner (p↑.ordonnée) ;
fin
```

On suppose ici que le point existe, c'est-à-dire que le pointeur *p* a une valeur différente du pointeur *NULL*.

c) *translater*

```
-- translate le point p de tx (abscisse) et ty (ordonnée)
procédure translater
        (màj p <Point>, entrée tx <Réel>, entrée ty <Réel>)

début
        p↑.abscisse <- p↑.abscisse + tx ;
        p↑.ordonnée <- p↑.ordonnée + ty ;
fin
```

Remarquer que cette nouvelle représentation physique d'un point ne modifie pas sa spécification fonctionnelle. Ceci montre qu'il existe plusieurs représentations possibles d'une même donnée et qu'un utilisateur du type *Point* n'a à en connaître que sa spécification fonctionnelle, c'est-à-dire son mode d'emploi.

Les annexes 2 et 4 précisent en détail la programmation des opérations du TAD *Point* en représentation dynamique (hormis l'affectation (<-) et l'égalité (=)).

Chapitre 3 : Implémentation d'un TAD

L'implémentation d'un TAD réalise les fonctionnalités définies dans sa partie spécification. C'est une mise en œuvre de la spécification du type concret.

1. IMPLEMENTATION D'UN TYPE CONCRET

Définition

Une implémentation de TAD regroupe l'ensemble des définitions utiles à la mise en œuvre de la spécification.

Remarque

Une implémentation de TAD peut inclure des définitions de constantes (mot-clé ***constante***) et de types (mot-clé ***type***). Elle comprend impérativement une définition de type (mot-clé ***type***) fournissant la représentation physique du TAD. Par convention, cette définition de type porte le même nom que le TAD spécifié.

Lorsque la spécification du TAD définit un en-tête de sous-programme (mot-clé ***procédure*** ou ***fonction***), l'implémentation doit proposer son corps correspondant. Bien évidemment, un corps de sous-programme peut être défini au sein d'une implémentation, sans que son en-tête ne figure dans la spécification. Un tel sous-programme sera interne au TAD et ne pourra être appelé que par l'implémentation.

Exemple

Nous obtenons finalement l'implémentation suivante pour notre TAD *Point* représenté statiquement par un enregistrement, conforme à sa spécification (voir aussi annexe 3) :

```
importer Couleur ;

-- définition du type Point
type Point : enregistrement
      abscisse <Réel>,          -- abscisse du point
      ordonnée <Réel>,          -- ordonnée du point
      couleur <Couleur>,        -- couleur du point
      taille <Réel> ;           -- taille du point


-- constante point à l'origine
fonction pointOrigine retourne <Point>

glossaire
      p <Point> ;          -- point retourné par la fonction
```

```
début
        p.abscisse <- 0.0 ;
        p.ordonnée <- 0.0 ;
        p.couleur <- noir ;
        p.taille <- 1.0 ;
        retourner (p) ;
fin


-- construit un point d'abscisse x, d'ordonnée y, de couleur c et de taille t
-- nécessite t > 0.0
fonction unPoint
        (entrée x <Réel>, entrée y <Réel>,
        entrée c <Couleur>, entrée t <Réel>)
retourne <Point>
déclenche tailleInvalide

glossaire
        p <Point> ;             -- point retourné par la fonction

début
        si t <= 0.0 alors
                déclencher (tailleInvalide) ;
        fin si ;
        p.abscisse <- x ;
        p.ordonnée <- y ;
        p.couleur <- c ;
        p.taille <- t ;
        retourner (p) ;
fin


-- abscisse du point p
fonction abscisse (entrée p <Point>) retourne <Réel>

début
        retourner (p.abscisse) ;
fin


-- ordonnée du point p
fonction ordonnée (entrée p <Point>) retourne <Réel>

début
        retourner (p.ordonnée) ;
fin
```

```
-- couleur du point p
fonction couleur (entrée p <Point>) retourne <Couleur>

début
    retourner (p.couleur) ;
fin


-- taille du point p
fonction taille (entrée p <Point>) retourne <Réel>

début
    retourner (p.taille) ;
fin


-- affecte la couleur c au point p
procédure modifierCouleur (màj p <Point>, entrée c <Couleur>)

début
    p.couleur <- c ;
fin


-- affecte la taille t au point p
-- nécessite t > 0.0
procédure modifierTaille (màj p <Point>, entrée t <Réel>)
déclenche tailleInvalide

début
    si t <= 0.0 alors
        déclencher (tailleInvalide) ;
    fin si ;
    p.taille <- t ;
fin


-- translate le point p de tx (abscisse) et ty (ordonnée)
procédure translater
    (màj p <Point>, entrée tx <Réel>, entrée ty <Réel>)

début
    p.abscisse <- p.abscisse + tx ;
    p.ordonnée <- p.ordonnée + ty ;
fin
```

```
-- met au facteur d'échelle h le point p
-- nécessite h > 0.0
procédure mettreAEchelle (màj p <Point>, entrée h <Réel>)
déclenche échelleInvalide

début
    si h <= 0.0 alors
        déclencher (échelleInvalide) ;
    fin si ;
    p.abscisse <- p.abscisse * h ;
    p.ordonnée <- p.ordonnée * h ;
    p.taille <- p.taille * h ;
fin
```

2. SEMANTIQUE DE VALEUR ET SEMANTIQUE DE REFERENCE

Le langage algorithmique et la plupart des langages de programmation supportent l'égalité (=) et l'affectation (<-) mettant en jeu deux variables d'un type donné. Si l'égalité de deux nombres entiers ou de deux tableaux par exemple est bien définie, il peut en être tout autrement de l'égalité de deux listes, de deux polygones ou de deux points.

La définition d'un opérateur = dans le cas de représentants de types abstraits (donc de variables du type) soulève la question de son comportement pour le client ; il est en est de même pour l'opérateur d'affectation <-. Il importe donc de bien définir la sémantique de ces opérateurs proposés par le TAD.

2.1. Sémantique de valeur

Définition

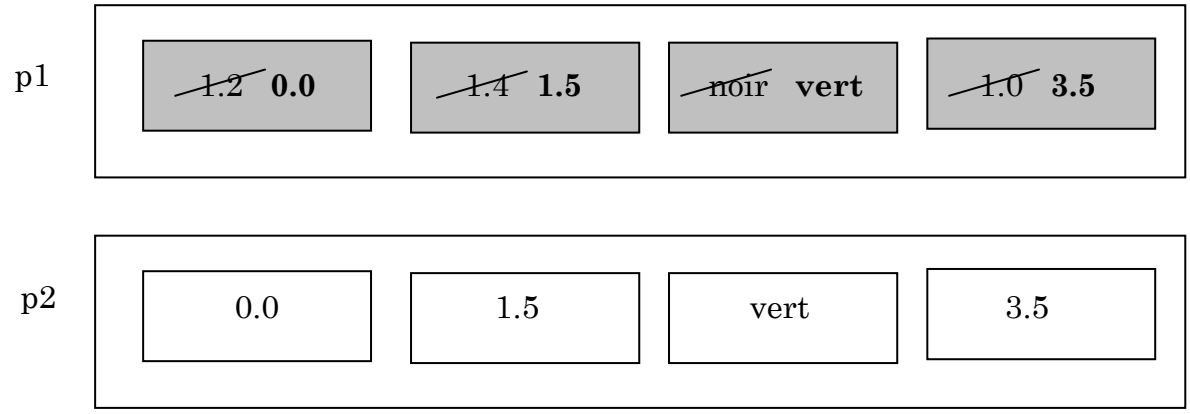
Une sémantique de valeur est telle que l'affectation *x* <- *y* revient à transférer le contenu de *y* dans *x*.

Remarque

Avec une *sémantique de valeur*, la signification de l'opération *x* <- *y*, où *x* et *y* sont deux représentants du TAD étudié, consiste à transférer le contenu de *y* (sa valeur) dans *x*. Une modification apportée par une partie du programme à *y* n'affecte pas les autres duplications (comme *x*). De même, la comparaison par l'égalité (=) ou l'inégalité (/=) n'implique que l'égalité ou l'inégalité des valeurs, que ces valeurs référencent ou non un même représentant du TAD.

Exemple

Ainsi, avec la représentation statique choisie pour *Point*, l'affectation *p1* <- *p2*, où *p1* et *p2* sont deux points, a une sémantique de valeur et produit *p1* par recopie des champs de l'enregistrement désigné par *p2* dans les champs correspondants de *p1* :



2.2. Sémantique de référence

Définition

Une sémantique de référence est telle que l'affectation *x* <- *y* revient à faire en sorte que *x* et *y* désignent le même représentant.

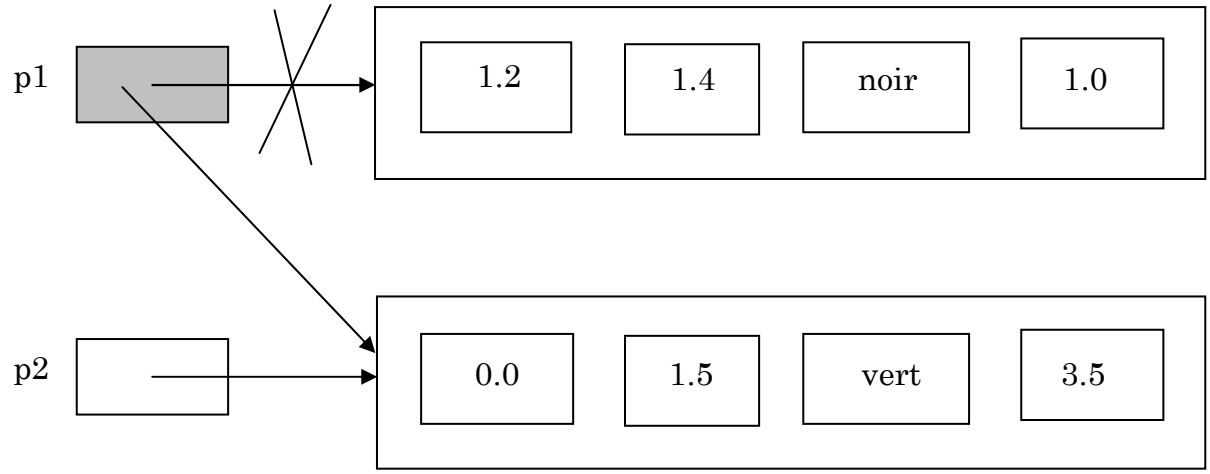
Remarque

Nous parlerons de *sémantique de référence* chaque fois que l'affectation *x* <- *y* est telle que *x* et *y* désignent le même représentant du TAD à l'issue de l'affectation. La variable *x* ou *y* est un identifiant dont les différentes duplications se réfèrent en fait au même contenant ; ces deux variables *x* et *y* sont dès lors des alias pour accéder à un même contenu. Toute modification apportée par un élément de programme au contenu du représentant du TAD désigné par *y* sera répercutée à tous ceux qui utilisent des copies de la référence au représentant, comme *x* par exemple.

De même, la comparaison d'égalité (=) entre deux variables sera telle que les deux variables désignent le même représentant, ce qui impliquera naturellement l'égalité des contenus. En revanche, deux représentants peuvent être trouvés différents même si les contenus sont identiques.

Exemple

Ainsi, si l'on considère la représentation dynamique du TAD *Point*, l'affectation *p1* <- *p2* se traduit par une affectation de pointeurs et induit une sémantique de référence pour le client symbolisée par le schéma ci-dessous :



Ainsi, toute modification sur le point référencé par *p1* se répercute sur le point référencé par *p2* et réciproquement ! On dit aussi que *p1* et *p2* sont des alias car ils désignent le même point. En pratique, l'effet n'est pas ici celui désiré. On pourra dès lors soit interdire à un client les opérations d'affectation (<-) et de comparaison (= et /=), soit surcharger par des sous-programmes ces opérations pour effectivement leur donner l'effet désiré.

Remarques

1) Les types numériques offrent typiquement une sémantique de valeur alors que les pointeurs sont basés sur une sémantique de référence. Attention toutefois à ne pas limiter la différence d'approche à cette simple alternative : un indice de tableau ou une chaîne de caractères désignant un nom de fichier ont une sémantique de référence !

2) Même avec une sémantique de référence, le programmeur du TAD peut offrir le transfert du contenu d'un représentant du TAD dans un autre, comme dans une sémantique de valeur. Il faut alors disposer dans le type concret d'une opération de recopie (*copier*) ; de même, on peut imaginer une comparaison de valeurs pour l'égalité (*égal*). On peut ainsi créer un type à sémantique de valeur avec une représentation à sémantique de référence. Ainsi, dans le cas d'une représentation d'un point par un pointeur, le programmeur doit définir :

```
-- p1 <- p2
procédure copier (sortie p1 <Point>, entrée p2 <Point>)

début
      p1↑ <- p2↑ ;                    -- copie d'enregistrements
fin
```

```
-- p1 = p2
fonction égal (entrée p1 <Point>, entrée p2 <Point>)
retourne <Booléen>

début
    retourner (p1↑ = p2↑) ;        -- comparaison d'enregistrements
fin
```

3. EXPORTATION DES OPERATEURS <-, = ET /=

Nous avons vu précédemment comment le concept d'implémentation cache ses entités internes à un utilisateur du TAD, notamment ce qui concerne les détails de construction du type. Nous pouvons aussi autoriser ou interdire indépendamment les opérateurs d'affectation (<-) et de comparaisons (= et /=). On retrouve ainsi les opérations *copier* et *égal* définies précédemment.

3.1. Implémentation sans exportation

Une implémentation sans exportation des opérateurs d'affectation (<-) et de comparaisons (= et /=) limite les seules opérations sur les variables d'un type donné à celles déclarées dans la spécification du TAD. On interdit alors à un client l'usage de l'affectation (<-) et des comparaisons (= et /=).

Exemple

Par le biais de sa spécification algorithmique, le TAD *Point* exporte les opérations définies sur un point (*pointOrigine*, *unPoint*, *abscisse*, …, *mettreAEchelle*), mais n'autorise pas les opérations d'affectation (<-) et de comparaison (= et /=) à un client.

En conséquence et quelle que soit la représentation physique choisie, le TAD limite les seules opérations sur les variables du type à celles déclarées dans sa spécification. Il en résulte dès lors que les écritures suivantes par un client sont illégales :

```
glossaire
    p1 <Point> ;            -- un premier point de l'application
    p2 <Point> ;            -- un deuxième point
    …
début
    …
    p1 <- p2 ;              -- opération <- non définie sur le type Point
    si p1 = p2 alors        -- opération = non définie sur le type Point
        …
    fin si ;
    …
fin
```

Remarque

Avec une implémentation sans exportation des opérations d'affectation (<-) et de comparaison (= et /=), on peut protéger de façon complète des variables dont on veut explicitement qu'elles ne soient pas manipulées en dehors des sous-programmes du TAD. Ce mode d'implémentation du TAD permet à son auteur d'avoir un contrôle total des variables du type, pour gérer par exemple la copie des ressources.

3.2. Implémentation avec exportation

Il apparaît cependant souhaitable dans certains cas que le TAD puisse exporter pour ses clients des opérateurs de recopie (<-) et de comparaison (= et /=). Ceux-ci peuvent être supportés implicitement par le langage algorithmique[8] ou définis par le programmeur. L'opérateur d'inégalité (/=) prenant toujours sa signification à partir de l'opérateur d'égalité (=) n'aura pas à être défini explicitement.

Au plan de la syntaxe, un TAD souhaitant exporter ces opérateurs devra le préciser dans la spécification algorithmique, sous la forme d'une procédure pour l'affectation (<-) et d'une fonction pour l'égalité (=) ayant pour identificateur le nom de l'opérateur. Les en-têtes des opérateurs "<-" et "=" pour un type *T* donné devront alors respecter la syntaxe suivante :

```
-- t1 <- t2
procédure "<-" (sortie t1 <T>, entrée t2 <T>) ;
```

```
-- t1 = t2
fonction "=" (entrée t1 <T>, entrée t2 <T>) retourne <Booléen> ;
```

Si la sémantique implicite du langage algorithmique de ces deux opérateurs correspond à ce qui est normalement attendu pour le TAD, seule la spécification algorithmique complétée par les profils suffit.

Dans le cas contraire, il conviendra de proposer, dans la partie implémentation du TAD, une définition des opérateurs que le TAD souhaite exporter, se substituant au traitement implicite effectué par le compilateur du langage algorithmique.

Exemples

1) La représentation statique du type *Point* confère une sémantique de valeur à ses opérateurs d'affectation (<-) et de comparaison (= et /=). Pour que le client puisse en bénéficier, il suffit de les rajouter dans la spécification algorithmique (voir annexe 2). Leur implémentation sera alors celle définie par le langage algorithmique, à savoir la copie et la comparaison d'enregistrements (voir annexe 3).

[8] Voir paragraphe sur la sémantique par valeur ou par référence

```
importer Couleur ;

-- constante point à l'origine
fonction pointOrigine retourne <Point> ;
…

-- p1 <- p2
procédure "<-" (sortie p1 <Point>, entrée p2 <Point>) ;

-- p1 = p2
fonction "=" (entrée p1 <Point>, entrée p2 <Point>)
retourne <Booléen> ;

…
```

Une invocation de ces opérateurs par le client pourra se faire en notation préfixée (comme pour un appel de sous-programme) ou en notation infixée (comme pour un opérateur) :

```
glossaire
    p1 <Point> ;          -- un premier point de l'application
    p2 <Point> ;          -- un deuxième point
    …
début
    …
    p1 <- p2 ;                -- ou "<-" (p1, p2) en notation préfixée
    si p1 = p2 alors          -- ou "=" (p1, p2) en notation préfixée
        …
    fin si ;
    …
fin
```

2) En représentation dynamique, un point est à sémantique de référence de par l'usage même des pointeurs. Pour bénéficier d'opérations prédéfinies à sémantique de valeur, le programmeur doit en proposer une définition dans la partie spécification du TAD (voir annexe 2) et une mise en œuvre dans la partie implémentation qui s'écrit alors (voir aussi annexe 4) :

```
importer Couleur ;

-- définition du type EnrPoint
type EnrPoint : enregistrement
    abscisse <Réel>,          -- abscisse du point
    ordonnée <Réel>,          -- ordonnée du point
    couleur <Couleur>,        -- couleur du point
    taille <Réel> ;           -- taille du point
```

```
-- définition du type Point
type Point : pointeur sur <EnrPoint> ;

-- constante point à l'origine
fonction pointOrigine retourne <Point>

glossaire
        p <Point> ;        -- point retourné par la fonction

début
        -- allouer l'espace pour mémoriser le point
        allouer (p) ;
        -- affecter les données du point
        p↑.abscisse <- 0.0 ;
        p↑.ordonnée <- 0.0 ;
        p↑.couleur <- noir ;
        p↑.taille <- 1.0 ;
        -- retourner le point
        retourner (p) ;
fin


-- construit un point d'abscisse x, d'ordonnée y, de couleur c et de taille t
-- nécessite t > 0.0
fonction unPoint
        (entrée x <Réel>, entrée y <Réel>,
        entrée c <Couleur>, entrée t <Réel>)
retourne <Point>
déclenche tailleInvalide

glossaire
        p <Point> ;              -- point retourné par la fonction

début
        si t <= 0.0 alors
                déclencher (tailleInvalide) ;
        fin si ;
        allouer (p) ;
        p↑.abscisse <- 0.0 ;
        p↑.ordonnée <- 0.0 ;
        p↑.couleur <- noir ;
        p↑.taille <- 1.0 ;
        retourner (p) ;
fin
```

```
-- abscisse du point p
fonction abscisse (entrée p <Point>) retourne <Réel>

début
    retourner (p↑.abscisse) ;
fin

-- ordonnée du point p
fonction ordonnée (entrée p <Point>) retourne <Réel>

début
    retourner (p↑.ordonnée) ;
fin


-- couleur du point p
fonction couleur (entrée p <Point>) retourne <Couleur>

début
    retourner (p↑.couleur) ;
fin


-- taille du point p
fonction taille (entrée p <Point>) retourne <Réel>

début
    retourner (p↑.taille) ;
fin


-- affecte la couleur c au point p
procédure modifierCouleur (màj p <Point>, entrée c <Couleur>)

début
    p↑.couleur <- c ;
fin


-- affecte la taille t au point p
-- nécessite t > 0.0
procédure modifierTaille (màj p <Point>, entrée t <Réel>)
déclenche tailleInvalide

début
    si t <= 0.0 alors
        lever (tailleInvalide) ;
    fin si ;
    p↑.taille <- t ;
fin
```

```
-- translate le point p de tx (abscisse) et ty (ordonnée)
procédure translater
        (màj p <Point>, entrée tx <Réel>, entrée ty <Réel>)

début
        p↑.abscisse <- p↑.abscisse + tx ;
        p↑.ordonnée <- p↑.ordonnée + ty ;
fin


-- met au facteur d'échelle h le point p
-- nécessite h > 0.0
procédure mettreAEchelle (màj p <Point>, entrée h <Réel>)
déclenche échelleInvalide

début
        si h <= 0.0 alors
                déclencher (échelleInvalide) ;
        fin si ;
        p↑.abscisse <- p↑.abscisse * h ;
        p↑.ordonnée <- p↑.ordonnée * h ;
        p↑.taille <- p↑.taille * h ;
fin


-- p1 <- p2
procédure "<-" (sortie p1 <Point>, entrée p2 <Point>)

début
        -- copier l'enregistrement pointé par p2
        -- dans l'enregistrement pointé par p1
        p1↑ <- p2↑ ;
fin


-- p1 = p2
fonction "=" (entrée p1 <Point>, entrée p2 <Point>) retourne <Booléen>

début
        -- comparer les deux enregistrements pointés par p1 et p2
        retourner (p1↑ = p2↑) ;
fin
```

Remarques

1) Attention à ne pas systématiquement priver le client des opérations d'affectation (<-), d'égalité (=) et de différence (/=), alors que parfois ces opérations sont offertes implicitement par le langage algorithmique. Il en est ainsi du TAD *Point* en représentation statique pour lequel l'affectation, l'égalité et la différence

correspondent à ce qui est normalement attendu pour un point. En conséquence, on aura tout intérêt à spécifier le profil de ces opérateurs dans l'interface du TAD.

2) Il est possible de séparer les traitements afférents à l'affectation (<-) et aux comparaisons (= et /=). Par exemple, on peut imaginer un TAD où l'affectation (<-) est prédéfinie et pour lequel les comparaisons (= et /=) sont inadaptées.

3) Le plus souvent, l'incarnation d'une fonction *égal* d'un type abstrait se traduira par l'opérateur = au niveau du type concret. De même, on sera souvent conduit à incarner une opération de copie par une définition de l'opérateur algorithmique <-.

4) Si un TAD générique utilise pour son implémentation l'opérateur d'affectation (<-), d'égalité (=) et de différence (/=) sur des représentants de son paramètre formel de généricité, le paramètre effectif de généricité devra exporter les opérateurs correspondants. Dans le cas contraire, l'instanciation générique n'est pas autorisée. Noter ainsi que le type *Couleur* importé par *Point* doit exporter l'affectation de deux couleurs, utilisée notamment dans les écritures *p.couleur* <- *c* (en représentation statique) et *p*↑*.couleur* <- *c* (en représentation dynamique).

3.3. Synthèse

Les tableaux ci-dessous résument les caractéristiques liées à l'exportation des opérateurs d'affectation (<-) et de comparaison (= et /=).

	Implémentation sans exportation des opérateurs <-, = et /=
Client du type	Opérateur interdit
Spécification du type	Pas d'en-tête pour l'opérateur
Implémentation du type	Pas de corps pour l'opérateur

	Implémentation avec exportation des opérateurs <-, = et /=
Client du type	Opérateur autorisé
Spécification du type	Présence d'un en-tête pour l'opérateur
Implémentation du type	Pas de corps pour l'opérateur : opérateur synthétisé par le langage algorithmique
	Présence d'un corps pour l'opérateur : opérateur défini par le sous-programme correspondant

4. TYPE FONCTIONNEL VS. TYPE IMPERATIF

Dans ce qui suit, nous établissons la distinction entre type fonctionnel et type impératif. Un type impératif se caractérise essentiellement par l'existence d'opérations de modification sur ses représentants (paramètres transmis en mode *mise à jour*) et fait usage de l'affectation (<-), alors qu'un type fonctionnel est tel qu'il exploite l'application de fonctions comme technique de calcul (paramètres transmis en mode *entrée* uniquement).

4.1. Type fonctionnel

Définition

Un type fonctionnel est composé uniquement de fonctions. Il ne propose pas en général d'opération d'affectation.

Exemple

Dans la vision fonctionnelle du TAD *Point*, les opérations de modification *modifierCouleur*, *modifierTaille*, *translater* et *mettreAEchelle* sont transformées en opérations d'évaluation et sont rebaptisées respectivement *nouveauPointColoré*, *nouveauPointAvecTaille*, *nouveauPointTranslaté* et *nouveauPointMisAEchelle* :

```
…
-- construit un nouveau point par changement de la couleur du point p
fonction nouveauPointColoré (entrée p <Point>, entrée c <Couleur>)
retourne <Point> ;

-- construit un nouveau point par changement de la taille du point p
-- nécessite t > 0.0
fonction nouveauPointAvecTaille (entrée p <Point>, entrée t <Réel>)
retourne <Point>
déclenche tailleInvalide ;

-- construit un nouveau point par translation du point p
-- de tx (abscisse) et ty (ordonnée)
fonction nouveauPointTranslaté
      (entrée p <Point>, entrée tx <Réel>, entrée ty <Réel>)
retourne <Point> ;

-- construit un nouveau point par mise à l'échelle du point p
-- nécessite h > 0.0
fonction nouveauPointMisAEchelle (entrée p <Point>, entrée h <Réel>)
retourne <Point>
déclenche échelleInvalide ;


--          pointOrigine = unPoint (0.0, 0.0, noir, 1.0)
--          …
```

4.2. Type impératif

Définition

Un type impératif exporte au moins une opération de modification, c'est-à-dire une procédure modifiant le représentant (paramètre transmis en mode *mise à jour*). Après l'appel d'une opération avec mutation, l'ancienne valeur du représentant est perdue. Le représentant du type abstrait est alors dit mutable car il possède un état pouvant évoluer dans le temps. On dit aussi qu'il s'agit d'un effet de bord. On notera

en particulier que l'affectation est la plus simple des instructions agissant par effet de bord sur le déroulement du programme.

Exemple

La vision impérative du TAD *Point* a déjà été proposée à la section 4 du chapitre 1 où les opérations *modifierCouleur*, *modifierTaille*, *translater* et *mettreAEchelle* ont été programmées par des procédures modifiant le point :

```
-- affecte la couleur c au point p
procédure modifierCouleur (màj p <Point>, entrée c <Couleur>) ;

-- affecte la taille t au point p
-- nécessite t > 0.0
procédure modifierTaille (màj p <Point>, entrée t <Réel>)
déclenche tailleInvalide ;

-- translate le point p de tx (abscisse) et ty (ordonnée)
procédure translater
      (màj p <Point>, entrée tx <Réel>, entrée ty <Réel>) ;

-- met au facteur d'échelle h le point p
-- nécessite h > 0.0
procédure mettreAEchelle (màj p <Point>, entrée h <Réel>)
déclenche échelleInvalide ;


--          pointOrigine = unPoint (0.0, 0.0, noir, 1.0)
--          …
…
```

Remarque

Pour un type impératif n'autorisant pas l'exportation de l'opérateur d'affectation, il conviendra parfois de remplacer toute fonction ayant en charge la construction d'un représentant du type par une procédure fournissant en sortie ce représentant. Ce changement s'impose notamment lorsque le type abstrait propose des opérations de modification sur un représentant… ce qui nécessite préalablement de mémoriser ce représentant autrement que par l'affectation classique.

Pour notre TAD *Point*, il s'agira de remplacer le constructeur *unPoint* :

```
-- construit un point d'abscisse x, d'ordonnée y, de couleur c et de taille t
-- nécessite t > 0.0
fonction unPoint
      (entrée x <Réel>, entrée y <Réel>,
      entrée c <Couleur>, entrée t <Réel>)
retourne <Point>
déclenche tailleInvalide ;
```

par un constructeur *créerUnPoint* incarné désormais par une procédure :

```
-- construit un point p d'abscisse x, d'ordonnée y, de couleur c et de taille t
-- nécessite t > 0.0
procédure créerUnPoint
        (entrée x <Réel>, entrée y <Réel>,
        entrée c <Couleur>, entrée t <Réel>, sortie p <Point>)
déclenche tailleInvalide ;
```

De même, la fonction *pointOrigine* :

```
-- constante point à l'origine
fonction pointOrigine retourne <Point> ;
```

donnera lieu à la procédure *créerPointOrigine* :

```
-- créer le point à l'origine
procédure créerPointOrigine (sortie p <Point>) ;
```

A noter cependant que la fonction *pointOrigine* pourra être maintenue en tant que constante si le TAD exporte l'opérateur d'égalité (=).

4.3. Style de programmation

La version impérative oblige le client à écrire des mutations de représentants par le biais d'appels à des procédures de modification. A l'inverse, dans la version fonctionnelle, c'est un enchaînement d'appels de fonction qui se charge de construire les caractéristiques que le représentant final doit posséder.

Le choix entre type concret impératif et fonctionnel a donc des incidences directes sur le style de programmation du client :

- avec des effets de bord et des appels de procédure pour un type concret impératif,
- avec uniquement des appels de fonction dans le cas d'un type concret fonctionnel.

Exemple

A partir du point à l'origine, cherchons à construire, un point aux coordonnées (10, 10), de couleur *rouge*, en exploitant la procédure d'affichage d'un point à l'écran suivante :

```
-- affiche le point p à l'écran
procédure afficher (entrée p <Point>) ;
```

Avec un style fonctionnel :

```
afficher
    (nouveauPointColoré
        (nouveauPointTranslaté (pointOrigine, 10.0, 10.0), rouge)) ;
```

Avec un style impératif avec affectation :

```
glossaire
    p <Point> ;      -- le point à obtenir

début
    p <- pointOrigine ;
    translater (p, 10.0, 10.0) ;
    modifierCouleur (p, rouge) ;
    afficher (p) ;
fin
```

Avec un style impératif sans affectation :

```
glossaire
    p <Point> ;      -- le point à obtenir

début
    créerPointOrigine (p) ;
    translater (p, 10.0, 10.0) ;
    modifierCouleur (p, rouge) ;
    afficher (p) ;
fin
```

Remarques

1) Il n'existe pas de règle générale pour choisir entre un type impératif et un type fonctionnel.

2) La vision du type abstrait peut être différente selon le style choisi ; on peut ainsi obtenir un jeu d'opérations différent selon que le type est incarné en impératif ou en fonctionnel.

Annexe

1) Spécification fonctionnelle du type abstrait *Point*

Opérations :

pointOrigine : → Point
unPoint : Réel x Réel x Couleur x Réel → Point
abscisse : Point → Réel
ordonnée : Point → Réel
couleur : Point → Couleur
taille : Point → Réel
modifierCouleur : Point x Couleur → Point
modifierTaille : Point x Réel → Point
translater : Point x Réel x Réel → Point
mettreAEchelle : Point x Réel → Point

Préconditions :

Pour *p* de type *Point*, *x*, *y*, *t, h* de type *Réel* et *c* de type *Couleur* :

unPoint (x, y, c, t) **est défini ssi** t > 0
modifierTaille (p, t) **est défini ssi** t > 0
mettreAEchelle (p, h) **est défini ssi** h > 0

Propriétés :

Pour *c*, *c1*, *c2* de type *Couleur*, *x*, *y*, *h*, *t*, *t1*, *t2*, *tx* et *ty* de type *Réel :*

(P1) pointOrigine = unPoint (0.0, 0.0, noir, 1.0)
(P2) abscisse (unPoint (x, y, c, t)) = x
(P3) ordonnée (unPoint (x, y, c, t)) = y
(P4) couleur (unPoint (x, y, c, t)) = c
(P5) taille (unPoint (x, y, c, t)) = t
(P6) modifierCouleur (unPoint (x, y, c1, t), c2) = unPoint (x, y, c2, t)
(P7) modifierTaille (unPoint (x, y, c, t1), t2) = unPoint (x, y, c, t2)
(P8) translater (unPoint (x, y, c, t), tx, ty) = unPoint (x + tx, y + ty, c, t)
(P9) mettreAEchelle (unPoint (x, y, c, t), h) = unPoint (x * h, y * h, c, t * h)

2) Spécification algorithmique du type abstrait *Point*

```
importer Couleur ;

-- constante point à l'origine
fonction pointOrigine retourne <Point> ;

-- construit un point d'abscisse x, d'ordonnée y, de couleur c et de taille t
-- nécessite t > 0.0
fonction unPoint
      (entrée x <Réel>, entrée y <Réel>,
      entrée c <Couleur>, entrée t <Réel>)
retourne <Point>
déclenche tailleInvalide ;

-- abscisse du point p
fonction abscisse (entrée p <Point>) retourne <Réel> ;

-- ordonnée du point p
fonction ordonnée (entrée p <Point>) retourne <Réel> ;

-- couleur du point p
fonction couleur (entrée p <Point>) retourne <Couleur> ;

-- taille du point p
fonction taille (entrée p <Point>) retourne <Réel> ;

-- affecte la couleur c au point p
procédure modifierCouleur (màj p <Point>, entrée c <Couleur>) ;

-- affecte la taille t au point p
-- nécessite t > 0.0
procédure modifierTaille (màj p <Point>, entrée t <Réel>)
déclenche tailleInvalide ;

-- translate le point p de tx (abscisse) et ty (ordonnée)
procédure translater
      (màj p <Point>, entrée tx <Réel>, entrée ty <Réel>) ;

-- met au facteur d'échelle h le point p
-- nécessite h > 0.0
procédure mettreAEchelle (màj p <Point>, entrée h <Réel>) ;
déclenche échelleInvalide ;
```

```
-- p1 <- p2
procédure "<-" (sortie p1 <Point>, entrée p2 <Point>) ;

-- p1 = p2
fonction "=" (entrée p1 <Point>, entrée p2 <Point>)
retourne <Booléen> ;


--    pointOrigine = unPoint (0.0, 0.0, noir, 1.0)
--    abscisse (unPoint (x, y, c, t)) = x
--    ordonnée (unPoint (x, y, c, t)) = y
--    couleur (unPoint (x, y, c, t)) = c
--    taille (unPoint (x, y, c, t)) = t
--    modifierCouleur (unPoint (x, y, c1, t), c2) = unPoint (x, y, c2, t)
--    modifierTaille (unPoint (x, y, c, t1), t2) = unPoint (x, y, c, t2)
--    translater (unPoint (x, y, c, t), tx, ty) = unPoint (x + tx, y + ty, c, t)
--    mettreAEchelle (unPoint (x, y, c, t), h) = unPoint (x * h, y * h, c, t * h)
```

3) Implémentation statique du type *Point*

```
importer Couleur ;

-- définition du type Point
type Point : enregistrement
      abscisse <Réel>,          -- abscisse du point
      ordonnée <Réel>,          -- ordonnée du point
      couleur <Couleur>,        -- couleur du point
      taille <Réel> ;           -- taille du point

-- constante point à l'origine
fonction pointOrigine retourne <Point>
glossaire
      p <Point> ;          -- point retourné par la fonction
début
      p.abscisse <- 0.0 ;
      p.ordonnée <- 0.0 ;
      p.couleur <- noir ;
      p.taille <- 1.0 ;
      retourner (p) ;
fin

-- construit un point d'abscisse x, d'ordonnée y, de couleur c et de taille t
-- nécessite t > 0.0
fonction unPoint
      (entrée x <Réel>, entrée y <Réel>,
      entrée c <Couleur>, entrée t <Réel>)
retourne <Point>
déclenche tailleInvalide
```

```
glossaire
        p <Point> ;              -- point retourné par la fonction
début
        si t <= 0.0 alors
                déclencher (tailleInvalide) ;
        fin si ;
        p.abscisse <- x ;
        p.ordonnée <- y ;
        p.couleur <- c ;
        p.taille <- t ;
        retourner (p) ;
fin

-- abscisse du point p
fonction abscisse (entrée p <Point>) retourne <Réel>
début
        retourner (p.abscisse) ;
fin

-- ordonnée du point p
fonction ordonnée (entrée p <Point>) retourne <Réel>
début
        retourner (p.ordonnée) ;
fin

-- couleur du point p
fonction couleur (entrée p <Point>) retourne <Couleur>
début
        retourner (p.couleur) ;
fin

-- taille du point p
fonction taille (entrée p <Point>) retourne <Réel>
début
        retourner (p.taille) ;
fin

-- affecte la couleur c au point p
procédure modifierCouleur (màj p <Point>, entrée c <Couleur>)

début
        p.couleur <- c ;
fin
```

```
-- affecte la taille t au point p
-- nécessite t > 0.0
procédure modifierTaille (màj p <Point>, entrée t <Réel>)
déclenche tailleInvalide
début
        si t <= 0.0 alors
                déclencher (tailleInvalide) ;
        fin si ;
        p.taille <- t ;
fin

-- translate le point p de tx (abscisse) et ty (ordonnée)
procédure translater
        (màj p <Point>, entrée tx <Réel>, entrée ty <Réel>)
début
        p.abscisse <- p.abscisse + tx ;
        p.ordonnée <- p.ordonnée + ty ;
fin

-- met au facteur d'échelle h le point p
-- nécessite h > 0.0
procédure mettreAEchelle (màj p <Point>, entrée h <Réel>)
déclenche échelleInvalide
début
        si h <= 0.0 alors
                déclencher (échelleInvalide) ;
        fin si ;
        p.abscisse <- p.abscisse * h ;
        p.ordonnée <- p.ordonnée * h ;
        p.taille <- p.taille * h ;
fin
```

4) Implémentation dynamique du type *Point*

```
importer Couleur ;

-- définition du type EnrPoint
type EnrPoint : enregistrement
        abscisse <Réel>,          -- abscisse du point
        ordonnée <Réel>,          -- ordonnée du point
        couleur <Couleur>,        -- couleur du point
        taille <Réel> ;           -- taille du point

-- définition du type Point
type Point : pointeur sur <EnrPoint> ;
```

```
-- constante point à l'origine
fonction pointOrigine retourne <Point>
glossaire
        p <Point> ;       -- point retourné par la fonction
début
        -- allouer l'espace pour mémoriser le point
        allouer (p) ;
        -- affecter les données du point
        p↑.abscisse <- 0.0 ;
        p↑.ordonnée <- 0.0 ;
        p↑.couleur <- noir ;
        p↑.taille <- 1.0 ;
        -- retourner le point
        retourner (p) ;
fin

-- construit un point d'abscisse x, d'ordonnée y, de couleur c et de taille t
-- nécessite t > 0.0
fonction unPoint
        (entrée x <Réel>, entrée y <Réel>,
        entrée c <Couleur>, entrée t <Réel>)
retourne <Point>
déclenche tailleInvalide
glossaire
        p <Point> ;              -- point retourné par la fonction
début
        si t <= 0.0 alors
                déclencher (tailleInvalide) ;
        fin si ;
        allouer (p) ;
        p↑.abscisse <- 0.0 ;
        p↑.ordonnée <- 0.0 ;
        p↑.couleur <- noir ;
        p↑.taille <- 1.0 ;
        retourner (p) ;
fin

-- abscisse du point p
fonction abscisse (entrée p <Point>) retourne <Réel>

début
        retourner (p↑.abscisse) ;
fin

-- ordonnée du point p
fonction ordonnée (entrée p <Point>) retourne <Réel>
début
        retourner (p↑.ordonnée) ;
fin
```

```
-- couleur du point p
fonction couleur (entrée p <Point>) retourne <Couleur>
début
      retourner (p↑.couleur) ;
fin

-- taille du point p
fonction taille (entrée p <Point>) retourne <Réel>
début
      retourner (p↑.taille) ;
fin

-- affecte la couleur c au point p
procédure modifierCouleur (màj p <Point>, entrée c <Couleur>)
début
      p↑.couleur <- c ;
fin

-- affecte la taille t au point p
-- nécessite t > 0.0
procédure modifierTaille (màj p <Point>, entrée t <Réel>)
déclenche tailleInvalide
début
      si t <= 0.0 alors
            lever (tailleInvalide) ;
      fin si ;
      p↑.taille <- t ;
fin

-- translate le point p de tx (abscisse) et ty (ordonnée)
procédure translater
      (màj p <Point>, entrée tx <Réel>, entrée ty <Réel>)
début
      p↑.abscisse <- p↑.abscisse + tx ;
      p↑.ordonnée <- p↑.ordonnée + ty ;
fin

-- met au facteur d'échelle h le point p
-- nécessite h > 0.0
procédure mettreAEchelle (màj p <Point>, entrée h <Réel>)
déclenche échelleInvalide
début
      si h <= 0.0 alors
            déclencher (échelleInvalide) ;
      fin si ;
      p↑.abscisse <- p↑.abscisse * h ;
      p↑.ordonnée <- p↑.ordonnée * h ;
      p↑.taille <- p↑.taille * h ;
fin
```

```
-- p1 <- p2
procédure "<-" (sortie p1 <Point>, entrée p2 <Point>)
début
    -- copier l'enregistrement pointé par p2
    -- dans l'enregistrement pointé par p1
    p1↑ <- p2↑ ;
fin

-- p1 = p2
fonction "=" (entrée p1 <Point>, entrée p2 <Point>) retourne <Booléen>
début
    -- comparer les deux enregistrements pointés par p1 et p2
    retourner (p1↑ = p2↑) ;
fin
```

Glossaire

Axiome : Egalité de deux termes définissant une *propriété* du type abstrait.

Client : Programmeur qui utilise une opération du type abstrait pour les besoins de son application.

Cohésion : Force de solidarité qui lie les opérations.

Couplage : Amplitude des contraintes qui assujettissent un segment de programme aux autres.

Encapsulation : Regroupement au sein d'une même entité de données et de sous-programmes.

Implémentation : Mise en œuvre de la spécification des opérations d'un type concret. Voir *type concret*.

Indépendance spatiale : Propriété selon laquelle tous les aspects relatifs à un type de données sont physiquement rattachés au même module. Voir aussi *encapsulation*.

Indépendance temporelle : Propriété selon laquelle un type de données peut se définir indépendamment du contexte d'une application qui l'utilise.

Opération : Symbole fonctionnel auquel est associé un profil précisant les *types* (*sortes*) des arguments et du résultat.

Opération de construction : Opération permettant de créer un représentant du type abstrait.

Opération de consultation : Opération permettant de fournir une valeur caractérisant un représentant du type abstrait.

Opération d'évaluation : Opération permettant de créer un nouveau représentant du type abstrait à partir d'un représentant existant.

Opération de modification : Opération permettant de modifier le contenu d'un représentant du type abstrait.

Profil : Domaine de valeurs des arguments et du résultat d'une opération.

Propriété : Description axiomatique du comportement d'une opération. Voir *axiome*.

Représentation physique : Structure pour mémoriser un représentant du *type concret*.

Sémantique de valeur : Propriété d'un type concret pour lequel l'affectation x <- y signifie que le contenu du représentant y est transféré dans le représentant x.

Sémantique de référence : Propriété d'un type concret pour lequel l'affectation *x* <- *y* signifie que *x* et *y* désignent le même représentant. Toute modification ultérieure de l'un modifie l'autre.

Signature : Syntaxe d'un type abstrait (nom des opérations et type de leurs arguments).

Sorte : Voir *Type*.

Spécification d'un type concret : Ensemble regroupant les en-têtes des sous-programmes du type concret et les propriétés du type abstrait. Voir *type concret*.

Spécification fonctionnelle : Ensemble des opérations autorisées sur les représentants du type et des propriétés de ces opérations. Voir *type abstrait*.

Type : Ensemble de valeurs et d'opérations applicables à ces valeurs.

Type abstrait : *Type* dont les valeurs sont abstraites car la vue du *client* est indépendante de la *représentation physique* adoptée.

Type concret : Incarnation d'un *type abstrait* dans un langage de programmation.

Type concret fonctionnel : *Type concret* composé uniquement de fonctions.

Type concret impératif : *Type concret* possédant au moins une procédure modifiant un représentant du type.

Type générique : Modèle de *type* prenant en paramètre un ou plusieurs types. On obtient un type effectif par instanciation du type générique.

Utilisateur : Voir *client*.

Bibliographie

J. Barnes
Programmer en Ada
InterEditions, 1988

M. Divay
Algorithmes et structures de données
Dunod, 1999

C. Froidevaux, M.-C. Gaudel, M. Soria
Types de Données et Algorithmes
Mc Graw-Hill, 1990

J. Guyot, C. Vial
Arbres, tables et algorithmes
Eyrolles, 1988

E. Horowitz, S. Sahni, S. Anderson-Freed
L'essentiel des structures de données en C
Dunod, 1993

G. Pierra
Les bases de la programmation et du génie logiciel
Dunod, 1991

J.-P. Rosen
Méthodes de Génie Logiciel avec Ada 95
InterEditions, 1995