Types abstraits algébriques

0.1 Abstraction et signatures

◊ Il est nécessaire de définir de manière non ambigüe[1] le comportement d'une routine.

◊ Il faut distinguer le modèle, l'interface et la représentation.

◊ Les types abstraits algébriques conviennent particulièrement bien à la description et à la modélisation des modules et des classes.

Un premier exemple très (trop ?) classique, la Pile

```
sorte
  Pile;
utilise
  Booléen, Élément;
opérations
  (* manipulateurs : *)
  (* - constructeur : *)
  pile_vide : --> Pile;
  (* transformateurs : *)
  empiler : Pile x Élément --> Pile;
  dépiler : Pile --> Pile;
  (* observateurs : *)
  sommet : Pile --> Élément;
  est_vide : Pile --> Booléen;
soit
  p : Pile; e : Élément;
antécédents
  dépiler(p) dssi non est_vide(p);
  sommet(p) dssi non est_vide(p);
axiomes
  est_vide(pile_vide) = vrai;
  est_vide(empiler(p,e)) = faux;
  est_vide(dépiler(empiler(p,e))) = est_vide(p);
  sommet(empiler(p,e)) = e;
  non est_vide(p) ==>
        sommet(dépiler(empiler(p,e)) = sommet(p);
fin Pile.
```

Nomenclature et règles d'écriture

Sorte définie :
: une sorte apparaissant à la rubrique «sorte».

Sorte pré-définie :
: une sorte apparaissant à la rubrique «utilise».

Profil d'une opération :
: la liste ordonnée des sortes déterminant les paramètres d'une opération et son résultat, notée

$$par_1 \text{ x } ... \text{ x } par_n \text{ --> res}$$

Observateur par rapport à une sorte *s* :
: une opération dont le profil comporte au moins un paramètre de sorte s mais dont le résultat n'est pas de sorte *s*.

---

[1] L'Académie française écrit, contre toute logique, ambiguë; nous préférons suivre l'exemple de Voltaire et écrire ambigüe; pour la petite histoire, il faut se rappeler que l'Académie avait fait droit aux représentations de Voltaire en 1972 et modifié l'orthographe du mot, elle est depuis revenue sur sa décision sous les pressions des fabricants de dictionnaires (nous ne saurions utiliser le mot éditeur car ce serait déprécier cette belle profession).

Manipulateur (ou opération interne) par rapport à une sorte *s* :

une opération dont le résultat est de sorte *s* (le profil peut aussi comporter, ou non, un paramètre de sorte *s*).

Constructeur par rapport à une sorte *s* :

un manipulateur par rapport à *s* dont aucun des paramètres du profil n'est de sorte *s*; un constructeur est dit constant lorsqu'il ne comporte aucun paramètre.

Transformateur par rapport à une sorte *s* :

un manipulateur par rapport à *s* dont au moins un des paramètres du profil est de sorte *s*; corollaire : tout manipulateur est soit un constructeur soit un transformateur.

R1 :

toute opération définie au sein d'un TA doit être soit un observateur en regard d'une sorte définie soit un manipulateur en regard d'une sorte définie.

R2 :

chaque sorte définie au sein d'un TA, doit être dotée d'au moins un constructeur.

0.2 Complétude suffisante partielle

Consistance :

une théorie (un TA) est dite consistante **ssi** pour toute formule P sans variable pouvant être démontrée vrai, il n'est pas possible de la démontrer fausse (la réciproque s'ensuivant).

Complétude :

une théorie (un TA) est dite complète ssi elle est consistante et si pour toute formule P sans variable on sait démontrer soit P, soit non P.

Complétude suffisante :

un TA est dit suffisamment complet ssi il est consistant et s'il est possible de déduire la valeur de toute application d'un observateur à un manipulateur ne comportant pas de variable.

Complétude suffisante partielle :

un TA est dit partiellement suffisamment complet ssi il est consistant et s'il est possible de déduire la valeur de toute application, dans les limites des domaines de définition des opérations en cause, d'un observateur à un manipulateur ne comportant pas de variable.

R3 :

il suffit de définir tous les observateurs en regard de chacun des manipulateurs pour obtenir un TA partiellement suffisamment complet.

Un deuxième exemple, la File

```
sorte
  File;
utilise
  Élément, Booléen;
opérations
  (* manipulateurs : *)
  (* constructeur : *)
  file_vide : --> File;
  (* transformateurs : *)
  ajouter : File x Élément --> File;
  retirer : File --> File;
  (* observateurs : *)
  est_vide : File --> Booléen;
  premier : File --> Élément;
```

```
soit
  f : File; e : Élément;
antécédents
  premier(f) dssi non est_vide(f);
  retirer(f) dssi non est_vide(f);
axiomes
  est_vide(file_vide) = vrai;
  est_vide(ajouter(f,e)) = faux;
  est_vide(retirer(ajouter(f,e)) = est_vide(f);
  premier(ajouter(file_vide,e)) = e;
  non est_vide(f) ==>
        premier(ajouter(f,e)) = premier(f);
  non est_vide(f) ==>
        premier(retirer(ajouter(f,e)) = premier(ajouter(retirer(f),e);
fin File.
```

0.3 Noyau et principe de factorisation

Noyau d'une sorte :

un noyau est un sous-ensemble des manipulateurs d'une sorte permettant d'en engendrer toutes les valeurs. En général on ne considère que les noyaux minimaux, un noyau est minimal ssi on ne peut en retirer aucun élément sans qu'il cesse d'être un noyau. En général, il peut exister plusieurs noyaux pour une même sorte.

R4 :

il suffit de définir toutes les opérations n'appartenant pas au noyau en regard de celles du noyau pour obtenir un TA partiellement suffisamment complet.

Factorisation appliquée à la Pile.

```
noyau
  { pile_vide, empiler };
axiomes
  dépiler(empiler(p,e)) = p
  est_vide(pile_vide) = vrai
  est_vide(empiler(p,e)) = faux
  sommet(empiler(p,e)) = e
```

Factorisation appliquée à la File.

```
noyau
  { file_vide, ajouter };
axiomes
  retirer(ajouter(file_vide,e)) = file_vide
  non est_vide(f) ==>
        retirer(ajouter(f,e)) = ajouter(retirer(f),e)
  est_vide(file_vide) = vrai
  est_vide(ajouter(f,e)) = faux
  premier(ajouter(file_vide,e)) = e
  non est_vide(f) ==>
        premier(ajouter(f,e)) = premier(f)
```

Un troisième exemple, le Vecteur

```
sorte
  Vecteur;
utilise
  Booléen, Entier, Élément;
opérations
  (* manipulateurs : *)
  (* constructeur : *)
  vect : Entier x Entier--> Vecteur;
  (* transformateur : *)
  changer : Vecteur x Entier x Élément --> Vecteur;
  (* observateurs : *)
  ième : Vecteur x Entier --> Élément;
```

```
  init : Vecteur x Entier--> Booléen;
  bInf : Vecteur --> Entier;
  bSup : Vecteur --> Entier;
noyau
  { vect, changer };
soit
  v : Vecteur; e : Élément; i, j, k : Entier;
antécédents
  changer(v,i,e) dssi (bInf(v) <= i <= bSup(v));
  ième(v,i) dssi (bInf(v) <= i <= bSup(v)) et init(v,i);
axiomes
  vrai ==>
        non init(vect(i,j),k);
  (bInf(v) <= i <= bSup(v)) ==>
        init(changer(v,i,e),i);
  (bInf(v) <= i <= bSup(v)) et (i<>j) ==>
        init(changer(v,i,e),j) = init(v,j);
  (* init(vect(i,j)) ne doit pas être défini, voir l'antécédent *)
  (bInf(v) <= i <= bSup(v)) ==>
        ième(changer(v,i,e),i) = e;
  (bInf(v) <= i,j <= bSup(v)) et (i<>j) et init(v,j) ==>
        ième(changer(v,i,e),j) = ième(v,j);
  vrai ==>
        bInf(vect(i,j)) = i;
  (bInf(v) <= i <= bSup(v)) ==>
        bInf(changer(v,i,e)) = bInf(v);
  vrai ==>
        bSup(vect(i,j)) = j;
  (bInf(v) <= i <= bSup(v)) ==>
        bSup(changer(v,i,e)) = bSup(v);
fin Vecteur.
```

0.4 Types abstraits à plusieurs noyaux

R5 :

> s'il existe plusieurs noyaux, lors de l'application R4, il est suffisant de définir chacune des opérations n'appartenant pas à un noyau en regard de celles d'un seul noyau... dans la mesure où les axiomes définissant l'équivalence entre les noyaux sont fournis.

Un quatrième exemple, la Queue

```
sorte
  Queue;
utilise
  Élément, Booléen;
opérations
  (* manipulateurs : *)
  (* constructeur : *)
  queue_vide : --> Queue;
  (* transformateurs : *)
  ajouterD : Queue x Élément --> Queue;
  ajouterF : Queue x Élément --> Queue;
  retirerD : Queue x Élément --> Queue;
  retirerF : Queue x Élément --> Queue;
  (* observateurs : *)
  est_vide : Queue --> Booléen;
  premier : Queue --> Élément;
  dernier : Queue --> Élément;
  longueur : Queue --> Entier;
noyaux
  {queue_vide, ajouter_D}, {queue_vide, ajouter_F};
soit
  q : Queue; e, e1, e2 : Élément;
```

```
antécédents
  premier(q) dssi non est_vide(q);
  dernier(q) dssi non est_vide(q);
  retirerD(q) dssi non est_vide(q);
  retirerF(q) dssi non est_vide(q);
axiomes
  (* définition de ajouterD et ajouterF l'une p/r à l'autre *)
  (* d'où il s'ensuit l'équivalence des deux noyaux *)
  ajouterD(queue_vide,e) = ajouterF(queue_vide,e);
  ajouterD(ajouterF(q,e1),e2) = ajouterF(ajouterD(q,e2),e1);
  (* définition des observateurs *)
  est_vide(queue_vide) = vrai;
  est_vide(ajouterF(q,e)) = faux;
  premier(ajouterD(q,e)) = e;
  dernier(ajouterF(q,e)) = e;
  longueur(queue_vide) = 0;
  longueur(ajouterD(q,e)) = longueur(q) + 1;
  (* définition des autres opérations  *)
  retirerD(ajouterD(q,e)) = q;
  retirerF(ajouterF(q,e)) = q;
fin Queue.
```

0.5 Incidences sur la testabilité

...

0.6 Incidences sur la dérivation

La dérivation d'une classe C par rapport à une classe D ne peut en affaiblir les invariants.

La redéfinition d'une méthode ne peut entraîner le renforcement d'un antécédent ni l'affaiblissement d'un conséquent.

0.7 Incidences sur le codage

Une méthode ne peut se terminer normalement que ssi elle remplit son mandat (maintenir les invariants, garantir le conséquent) ; corollaire sur la bonne utilisation des exceptions.

Coder consiste à élaborer une suite de transformations qui mènent de façon démontrable de l'antécédent au conséquent à un coût acceptable en terme de critères pré-établis.

Programmer regroupe les activités de conception, de codage et d'essais.

Selon les auteurs, on parle de (langages de) programmation de haut niveau :

- ◊ lorsqu'il n'y a pas de codage ;
- ◊ lorsque le codage peut être exprimé dans la même notation que la conception ;
- ◊ lorsque le codage fait appel au même niveau d'abstraction que la conception ;
- ◊ lorsque le codage fait abstraction de la représentation ;
- ◊ n'importe quand.

0.8 Quelques types abstraits classiques

0.8.1 Type abstrait Liste récursive

Premier exemple de définition conjointe de deux sortes; on remarque que l'opération tête est un observateur par rapport à Liste et un constructeur par rapport à Place.

```
sorte
  Liste, Place;
utilise
  Élément, Booléen;
opérations
(* opérations propres aux listes *)
  (* manipulateurs : *)
  (* constructeur : *)
  liste_vide : --> Liste;
```

```
  (* transformateurs : *)
  cons : Liste x Élément --> Liste;
  corps : Liste --> Liste;
  (* observateur : *)
  est_vide : Liste --> Booléen;
(* opérations propres aux places *)
  (* manipulateurs : *)
  (* constructeur : *)
  nil : --> Place;
  (* transformateur : *)
  succ : Place --> Place;
  (* observateur : *)
  contenu : Place --> Élément;
(* opération mixte : *)
  tête : Liste --> Place;
noyaux
  Liste : {liste_vide, cons};
  Place : {tête, succ};
soit
  x : Liste; e : Élément; c : Place;
extension
  premier(x) == contenu(tête(x));
antécédents
  corps(x) dssi x <> liste_vide;
  succ(c) dssi c <> nil;
  contenu(c) dssi c <> nil;
axiomes
  vrai ==>
        est_vide(liste_vide) = vrai;
  vrai ==>
        est_vide(cons(x,e)) = faux;
  vrai ==>
        corps(cons(x,e)) = x;
  vrai ==>
        nil = tête(liste_vide);
  vrai ==>
        nil = succ(cons(liste_vide,e)));
  vrai ==>
        contenu(tête(cons(x,e))) = e;
  x <> liste_vide ==>
        contenu(succ(cons(x,e))) = contenu(tête(x));
fin Liste.
```

0.8.2 Type abstrait Liste itérative

Ce type est une généralisation du type abstrait Liste récursive incluant un accès discret aux places, on peut y voir une fusion du type Liste récursive et du type Vecteur.

```
sorte
  Liste, Place;
utilise
  Élément, Booléen, Naturel;
définition
  Rang == Naturel;
opérations
(* opérations propres aux listes *)
  (* manipulateurs : *)
  (* constructeur : *)
  liste_vide : --> Liste;
  (* transformateurs : *)
  insérer : Liste x Rang x Élément --> Liste;
  retirer : Liste x Rang --> Liste;
  (* observateur : *)
  longueur : Liste --> Entier;
(* opérations propres aux places *)
  (* manipulateurs : *)
```

```
  (* constructeur : *)
  nil : --> Place;
  (* transformateurs : *)
  succ : Place --> Place;
  pred : Place --> Place;
  (* observateurs : *)
  contenu : Place --> Élément;
(* opérations mixtes : *)
  accès : Liste x Rang --> Place;
  insérerCP : Liste x Place x Élément --> Liste;
  insérerCS : Liste x Place x Élément --> Liste;
  retirerC : Liste x Place --> Liste
noyaux
  Liste : { liste_vide, insérer };
  Place : { nil, accès };
soit
  x : Liste; e : Élément; i,j : Rang; c : Place;
extensions
  est_vide(x) == longueur(x) = 0;
  valeur(x,i) == contenu(accès(x,i));
antécédents
  insérer(x,i,e) dssi 0 <= i <= longueur(x);
  retirer(x,i) dssi 0 <= i < longueur(x);
  pred(c) dssi c <> nil;
  succ(c) dssi c <> nil;
  contenu(c) dssi c <> nil;
  accès(x,i) dssi 0 <= i < longueur(x);
  insérerCP(x,c,e) dssi c <> nil;
  insérerCS(x,c,e) dssi c <> nil;
  retirerC(x,c) dssi c <> nil;
axiomes
  vrai ==>
        longueur(liste_vide) = 0;
  vrai ==>
        longueur(insérer(x,i,e)) = longueur(x) + 1;
  i < j ==>
        retirer(insérer(x,i,e),j) = ajouter(retirer(x,j-1),i,e);
  i = j ==>
        retirer(insérer(x,i,e),j) = x;
  i > j ==>
        retirer(insérer(x,i,e),j) = ajouter(retirer(x,j),i+1,e);
  i < j ==>
        valeur(insérer(x,i,e),j) = valeur(x,j+1,e);
  i = j ==>
        valeur(insérer(x,i,e),j) = e;
  i > j ==>
        valeur(insérer(x,i,e),j) = valeur(x,j,e);
  0 <= i < longueur(x)-1 ==>
        succ(accès(x,i)) = accès(x,i+1);
  longueur(x) > 0 ==>
        succ(accès(x,longueur(x)-1)) = nil;
  1 <= i < longueur(x) ==>
        pred(accès(x,i)) = accès(x,i-1);
  longueur(x) > 0 ==>
        pred(accès(x,0)) = nil;
  (c <> nil) etc (c = accès(x,i)) ==>
        insérerCS(x,c,e) = insérer(x,i,e);
  (c <> nil) etc (c = accès(x,i)) ==>
        insérerCP(x,c,e) = insérer(x,i+1,e);
  (c <> nil) etc (c = accès(x,i)) ==>
        retirerC(x,c,e) = retirer(x,i);
fin Liste.
```

Il est souvent plus pratique de libéraliser l'antécédent de l'opérateur accès en admettant la valeur longueur(x) dans les indices légitimes, soit

```
  accès(x,i) dssi 0 <= i <= longueur(x);
```

Le cas supplémentaire est alors pris en compte par l'axiome suivant :

```
  accès(x,longueur(x)) = nil
```

Dans ce cas, il est possible de retirer nil du noyau.

0.8.3 Type abstrait File de priorité

...

```
sorte
  FileDePriorité, Place;
utilise
  Élément, Booléen, Entier;
opérations
(* opérations propres aux files de priorité *)
  (* manipulateurs : *)
  (* constructeur : *)
  fp_vide : --> FileDePriorité;
  (* transformateurs : *)
  cons : FileDePriorité x Élément x Entier --> FileDePriorité;
  insérer : FileDePriorité x Élément x Entier --> FileDePriorité;
  corps : FileDePriorité --> FileDePriorité;
  (* observateur : *)
  est_vide : FileDePriorité --> Booléen;
(* opérations propres aux places *)
  (* manipulateurs : *)
  (* constructeur : *)
  nil : --> Place;
  (* transformateur : *)
  succ : Place --> Place;
  (* observateurs : *)
  contenu : Place --> Élément;
  priorité : Place --> Entier;
(* opérations mixtes : *)
  tête : FileDePriorité --> Place;
  retirer : FileDePriorité x Place --> FileDePriorité;
noyaux
  FileDePriorité : {fp_vide, cons}, {fp_vide, insérer};
  Place : {tête, succ};
soit
  x : FileDePriorité; e : Élément; c : Place; i : Entier;
extensions
  premier(x) ==
        contenu(tête(x));
  priomin(x) ==
        priorité(tête(x));
  présent(x,c) ==
        (x<>fp_vide) etc (tête(x)=c ou présent(corps(x),c));
  déplacer(x,c,i) ==
        insérer(retirer(x,c),contenu(c),i);
antécédents
  cons(x,e,i) dssi (x=fp_vide) ouc (priomin(x) > i);
  corps(x) dssi x <> fp_vide;
  succ(c) dssi c <> nil;
  contenu(c) dssi c <> nil;
  priorité(c) dssi c <> nil;
  retirer(x,c) dssi présent(x,c);
  premier(x) dssi x <> fp_vide;
  priomin(x) dssi x <> fp_vide;
  déplacer(x,c,i) dssi présent(x,c);
```

```
axiomes
  (* FileDePriorité *)
  (* équivalence des noyaux *)
  (x = fp_vide) ouc (priomin(x)) > i) ==>
        insérer(x,e,i)) = cons(x,e,i);
  (x <> fp_vide) etc (priomin(x)) <= i) ==>
        insérer(x,e,i) = cons(insérer(corps(x),e,i),premier(x),priomin(x));
  (* définition de est_vide *)
  vrai ==>
        est_vide(fp_vide) = vrai;
  vrai ==>
        est_vide(cons(x,e,i)) = faux;
  vrai ==>
        corps(cons(x,e,i)) = x;
  (* Place *)
  vrai ==>
        nil = tête(fp_vide);
  vrai ==>
        nil = succ(cons(fp_vide,e,i));
  vrai ==>
        premier(cons(x,e,i)) = e;
  vrai ==>
        priomin(cons(x,e,i)) = i;
  x <> fp_vide ==>
        contenu(succ(tête(cons(x,e,i)))) = premier(x);
  vrai ==>
        priorité(succ(tête(cons(x,e,i)))) = priomin(x);
  (* Mixtes *)
  présent(x,c) etc tête(x)=c ==>
        retirer(x,c)=corps(x);
  présent(x,c) etc tête(x)<>c ==>
        retirer(x,c) = cons(retirer(corps(x),c),premier(x),priomin(x));
fin FileDePriorité.
```

0.8.4 Exercice

Définir un type abstrait Automates munis des opérations de chargement, de débit et d'examen. L'automate, ou «machine distributrice», considéré ici permet la vente automatisée d'un choix restreint de menus articles contre des sommes payables en pièces de monnaie. Parmi les opérations à modéliser, on retrouve celles-ci : chargement des stocks d'articles et fixation de leur prix, chargement de pièces de monnaie (nécessaire pour pouvoir rendre la monnaie), vente d'un article avec éventuellement remboursement des sommes payées en trop, consultation du niveau des stocks de menus articles et de pièces de monnaie. Vous devez prendre en compte les possibilités suivantes : épuisement d'un article, incapacité de pouvoir rendre la monnaie.

```
La méthode de définition la plus simple sera systématiquement utilisée,
à savoir nous ne définirons pas de noyau; cette approche est un peu plus
verbeuse mais peut-être plus immédiate. Il existe, cela va de soi,
plusieurs autres bonnes solutions; bonne lecture!

sorte
  Monnaie, Pièce;
utilise
  Entier, Booléen;
définition
  Pièce = (cinq, dix, vingt-cinq);
opérations
  monnaie_vide : --> Monnaie;
  ajouter : Monnaie x Pièce x Entier --> Monnaie;
  retirer : Monnaie x Pièce x Entier --> Monnaie;
  qt : Monnaie x Pièce --> Entier;
soit
  m, m1, m2: Monnaie; p, p1, p2 : Pièce; a, b, c, n : EntierNaturel;
extensions
```

```
  somme(m) ==
        qt(m,cinq)*5 + qt(m,dix)*10 + qt(m,vingt-cinq)*25;
  dans(m1,m2) ==
        qt(m1,cinq) <= qt(m2,cinq) et
        qt(m1,dix) <= qt(m2,dix) et
        qt(m1,vingt-cinq) <= qt(m2,vingt-cinq);
  exprimable(m,n) ==
        somme(m2)=n et dans(m2,m);
  exprimable(n) ==        (* surcharge de l'opérateur *)
        n div 5 = 0;
antécédent
  retirer(m,p,n) dssi qt(m,p) >= n;
axiomes
  qt(monnaie_vide,p) = 0;
  qt(ajouter(m,p,n),p) = qt(m,p) + n;
  p1 <> p2 ==> qt(ajouter(m,p1,n),p2) = qt(m,p2);
  qt(retirer(m,p,n),p) = qt(m,p) - n;
  p1 <> p2 ==> qt(retirer(m,p1,n),p2) = qt(m,p2);
fin Monnaie.

sorte
  Automate;
utilise
  Pièce, Monnaie, Entier;
définition
  nbArticle == 5;         (* valeur exemplaire *)
  qtMax == 20;            (* valeur exemplaire *)
  Article == [0..nbArticle-1];
  Quantité == [0..qtMax];
  Prix == Entier;
opérations
  (* manipulateurs : *)
  (* constructeur : *)
  auto_vide : --> Automate;
  (* transformateurs : *)
  chargerArticle : Automate x Article x Quantité x Prix --> Automate;
  chargerMonnaie : Automate x Monnaie --> Automate;
  vendre : Automate x Article x Monnaie --> Automate;
  (* observateurs : *)
  evalQuantité : Automate x Article --> Entier;
  evalPrix : Automate x Article --> Entier;
  evalMonnaie : Automate --> Monnaie;
soit
  a : Automate; x : Article; m : Monnaie; n : Quantité; p : Prix;
  pm : Pièce;
```

```
extensions
  ventePossible (a,x,m) ==
        evalQuantité(a,x) > 0 et evalPrix(a,x) <= somme(m) etc
              exprimable(evalMonnaie(a),somme(m)-evalPrix(a,x));
  chargementPossible (a,x,n,p) ==
        evalQuantité(a,x)+n <= qtMax et exprimable(p);
antécédents
  vendre(a,x,m) dssi ventepossible(a,x,m);
  chargerArticle(a,x,n,p) dssi chargementPossible(a,x,n,p);
  (* nous n'avons pas modélisé de capacité maximale pour la monnaie *)
axiomes
  evalQuantité(auto_vide,x) = 0;
  evalPrix(auto_vide,x) = 0;
  evalMonnaie(auto_vide) = monnaie_vide;

  vrai ==>
        evalQuantité(chargerArticle(a,x,n,p),x) = evalQuantité(a,x) + n;
  x1 <> x2 ==>
        evalQuantité(chargerArticle(a,x1,n,p),x2) = evalQuantité(a,x2);
  vrai ==>
        evalPrix(chargerArticle(a,x,n,p),x) = p;
  x1 <> x2 ==>
        evalPrix(chargerArticle(a,x1,n,p),x2) = evalPrix(a,x2);
  vrai ==>
        evalMonnaie(chargerArticle(a,x,n,p)) = evalMonnaie(a);

  vrai ==>
        evalQuantité(chargerMonnaie(a,m),x) = evalQuantité(a,x);
  vrai ==>
        evalPrix(chargerMonnaie(a,m),x) = evalPrix(a,x);
  vrai ==>
        qt(evalMonnaie(chargerMonnaie(a,m),pm) =
              qt(evalMonnaie(a),pm)+qt(m,pm);

  vrai ==>
        evalQuantité(vendre(a,x,m),x) = evalQuantité(a,x)-1;
  x1 <> x2 ==>
        evalQuantité(vendre(a,x1,m),x2) = evalQuantité(a,x2);
  vrai ==>
        evalPrix(vendre(a,x1,m),x2) = evalPrix(a,x2);
  somme(evalMonnaie(vendre(a,x,m))) =
        somme(evalMonnaie(a)) + evalPrix(a,x);
  qt(evalMonnaie(vendre(a,x,m)),pm) <=
        qt(evalMonnaie(a),pm) + qt(evalMonnaie(m),pm)
fin Automate.
```

0.9 Un exemple en Ada

```
--* Piles_AS [Élément <= ObjetComparable] <= ObjetComparable
--* -----
--* Représentation de piles homogènes ajustables (non bornées a priori)
--* dont les composants sont gérés statiquement.

--  Définition
--  ----------
--  Afin de rendre la présente interface auto-suffisante, nous allons
--  tout d'abord présenter le type abstrait Pile en utilisant la
--  notation exposée dans [1].
--
--  sorte
--      Pile [Élément <= ObjetComparable];
--  utilise
```

```
--      Booléen, Naturel, Élément;
--  opérations
--    (* manipulateurs *)
--      (* constructeur *)
--      pileVide : --> Pile;
--      (* transformateurs *)
--      empiler : Pile x Élément --> Pile;
--      dépiler : Pile --> Pile;
--   (* observateurs *)
--      sommet : Pile --> Élément;
--      vide : Pile --> Booléen;
--  noyau
--      { pileVide, empiler }
--  soit
--      p : Pile; e : Élément;
--  antécédents
--      dépiler(p) dssi non vide(p)
--      sommet(p) dssi non vide(p)
--  axiomes
--      dépiler(empiler(p,e))) = p;
--      vide(pileVide) = vrai;
--      vide(empiler(p,e)) = faux;
--      sommet(empiler(p,e)) = e;
--  extensions
--      profondeur : Pile --> Naturel;
--  règles
--      profondeur(pileVide) = 0;
--      profondeur(empiler(p,e)) = 1 + profondeur(p);
--  fin Pile.
pragma page;

--  Remarques
--  ---------
--  nil.

--  Références
--  ----------
--  [1] Luc LAVOIE;
--      «Petit guide pratique de la théorie des types abstraits
--       axiomatiques»;
--      Société Genilog, Montréal, 1995.

--  Historique
--  ----------
--   •  1993-06-15  (1.02)  Luc LAVOIE
--      Correction de coquilles diverses.
--   •  1993-05-15  (1.01)  Luc LAVOIE
--      Adaptation pour Oerlikon Aérospatiale.
--   •  1991-06-10  (1.00)  Luc LAVOIE
--      Spécification originale en Ada.
--   •  1985-02-10  (0.10)  Luc LAVOIE
--      Spécification originale en Modula-2.

--  Auteur
--  ------
--  Luc LAVOIE
--  Genilog
--  4837, rue Boyer, bureau 235
--  Montréal  QC  H2J 3E6
--
--  téléphone :     +1 (514) 525-5656
```

```
--  bélinographe :  +1 (514) 525-5647
--
--* Copyright (©) 1985-1993 par Luc LAVOIE.
--* Copyright (©) 1995-1999 par Genilog.
--
--  Tous droits réservés par l'auteur; permission de copier, modifier,
--  distribuer et utiliser est faite dans la mesure où le présent
--  préambule reste intact et que l'action entreprise ne le soit pas
--  dans un but lucratif.
--
--  Copyright by the author; copying, modifying, distribution and
--  utilization permission is given provided that the present header
--  is left intact and that the action taken is not for profit.
------------------------------------------------------------------------
pragma page;

with EnvGenilog;
use EnvGenilog;

generic
    type Element is private;

package Piles_AS is

type Pile is
    limited private;

procedure Allouer
        (
        p :         out Pile
        );
--* Créer une pile vide.
--
--  Antécédent :
--      non %all(p)
--  Conséquent :
--      %all(p) et %vide(p)
--  Temps :
--      O(1)
--  Espace :
--      E(Allouer) : O(1)
--      E(p) : O(1)
--  Remarque :
--      L'allocation d'une pile est un préalable à l'application de
--  toute autre routine à celle-ci.

procedure Liberer
        (
        p :     in  out Pile
        );
--* Récupérer l'espace associé à une pile, puis la détruire.
--
--  Antécédent :
--      %all(p)
--  Conséquent :
--      non %all(p)
--  Temps :
--      O(%profondeur(p))
--  Encombrement :
--      E(Liberer) : O(1)
--  Remarque :
```

```
--      La libération d'une pile la rend impropre à être utilisée par une
--  routine autre qu'Allouer.

procedure Copier
        (
        copie :         in  out Pile;
        originale :     in      Pile
        );
--* Modifier une pile de telle sorte qu'elle soit la copie d'une autre.
--
--  Antécédent :
--      --
--  Conséquent :
--      copie' = originale
--  Temps :
--      O(%profondeur(originale)+%profondeur(copie))
--  Encombrement :
--      E(Copier) : O(1)
--      E(copie') : E(originale)

function Egalite
        (
        a :         in      Pile;
        b :         in      Pile
        )
        return Booleen;
--* Déterminer si deux piles sont égales composant à composant.
--
--  Antécédent :
--      --
--  Résultat :
--      a = b
--  Temps :
--      O(%profondeur(a))
--  Encombrement :
--      E(Egalite) : O(1)
--  Remarque :
--      En fait le temps est constant pour deux piles de longueurs
--      différentes et proportionnel à la longueur commune sinon.

function Vide
        (
        p :     in      Pile
        )
        return Booleen;
--* Déterminer si une pile est vide.
--
--  Antécédent :
--      --
--  Résultat :
--      %vide(p)
--  Temps :
--      O(1)
--  Encombrement :
--      E(Vide) : O(1)

function Longueur
        (
        p :     in      Pile
        )
        return Naturel;
```

```
--* Déterminer le nombre de composants dans une pile.
--
--  Antécédent :
--      --
--  Résultat :
--      %profondeur(p)
--  Temps :
--      O(1)
--  Encombrement :
--      E(Longueur) : O(1)

function Premier
        (
        p :     in      Pile
        )
        return Element;
--* Déterminer la valeur du premier élément d'une pile (son sommet).
--
--  Antécédent :
--      non Vide(p)
--  Résultat :
--      %sommet(p)
--  Temps :
--      O(1)
--  Encombrement :
--      E(Premier) : O(1)

procedure Adjoindre
        (
        p :         in  out Pile;
        e :         in      Element
        );
--* Ajouter un élément au sommet de la pile (empiler).
--
--  Antécédent :
--      --
--  Conséquent :
--      p' = %empiler(p,e)
--  Temps :
--      O(1)
--  Encombrement :
--      E(Adjoindre) : O(1)
--      E(p') : O(%profondeur(p)+1)

procedure Reduire
        (
        p :         in  out Pile
        );
--* Retirer le premier élément d'une pile (dépiler).
--
--  Antécédent :
--      non Vide(p)
--  Conséquent :
--      p' = %dépiler(p)
--  Temps :
--      O(1)
--  Encombrement :
--      E(Reduire) : O(1)
--      E(p') : O(%profondeur(p)-1)

private
```

```
    type DefPile;
    type Pile is access DefPile;

end Piles_AS;
```

```
--* Piles_AS_T00
--* -----
--* Programme d'essai unitaire du module Piles_AS.

--  Définition
--  ----------
--  Essai unitaire du module Piles_AS par application systématique
--  (lorsqu'applicable) des axiomes du type abstrait à trois instances
--  de piles : la pile vide, une pile de longueur un et une pile de
--  longueur trois.
--
--  Les essais sont concluants si le programme se termine normalement en
--  affichant la mention « Essai concluant ». Lors de la détection d'une
--  inconsistance le programme affiche un message approprié ou se termine
--  anormalement.
pragma page;

--  Remarques
--  ---------
--  nil.

--  Références
--  ----------
--  nil.

--  Historique
--  ----------
--   •  1993-05-15  (1.01)  Luc LAVOIE
--      Adaptation pour Oerlikon Aérospatiale.
--   •  1991-06-10  (1.00)  Luc LAVOIE
--      Spécification originale en Ada.
--   •  1985-02-10  (0.10)  Luc LAVOIE
--      Spécification originale en Modula-2.

--  Auteur
--  ------
--  Luc LAVOIE
--  Genilog
--  4837, rue Boyer, bureau 235
--  Montréal  QC  H2J 3E6
--
--  téléphone :     +1 (514) 525-5656
--  bélinographe :  +1 (514) 525-5647
--
--* Copyright (©) 1985-1993 par Luc LAVOIE.
--* Copyright (©) 1995-1999 par Genilog.
--
--  Tous droits réservés par l'auteur; permission de copier, modifier,
--  distribuer et utiliser est faite dans la mesure où le présent
--  préambule reste intact et que l'action entreprise ne le soit pas
--  dans un but lucratif.
--
--  Copyright by the author; copying, modifying, distribution and
--  utilization permission is given provided that the present header
--  is left intact and that the action taken is not for profit.
-------------------------------------------------------------------------
pragma page;

with TEXT_IO;
with EnvGenilog;
```

```
with Piles_AS;
use EnvGenilog;
procedure Piles_AS_T00 is

    -- Environnement et définition du paquetage à soumettre aux essais.
    subtype ElementDeTest is Naturel;
    v0      : constant ElementDeTest    := 3;   -- valeur exemplaire
    v1      : constant ElementDeTest    := 4;
    v2      : constant ElementDeTest    := 5;
    v3      : constant ElementDeTest    := 6;
    v4      : constant ElementDeTest    := 7;
    package Piles is
        new Piles_AS (Element => ElementDeTest);
    use Piles;

    -- Génération du rapport d'essai.
    globalement_Concluant : --  Évaluation d'ensemble de l'essai
        Booleen := vrai;

    procedure Tester
            (
            identification :in      String;
            concluant :     in      Booleen;
            erreur :        in      String
            ) is
    begin
        if not concluant then
            globalement_Concluant := faux;
            TEXT_IO.Put         ("échec ");
            TEXT_IO.Put         (identification);
            TEXT_IO.Put_Line    (" : ");
            TEXT_IO.Put         ("----- ");
            TEXT_IO.Put_Line    (erreur);
        end if;
    end Tester;

    -- Suite d'essais
    procedure Tester_Axiomes_Et_Regles
            (
            identification :    in      String;
            p :                 in      Pile;
            p_Vide :            in      Booleen
            ) is
        p2 :
            Pile;
    begin
        -- Préparation
        Allouer (p2);
        Copier (p2, p);

        -- Essai de Copier vs Egalité
        Tester (identification, Egalite(p,p2), "p = p");
        Adjoindre (p2,v3);
        Tester (identification, not Egalite(p,p2), "p /= empiler(p,e)");

        -- Axiomes
        Reduire (p2);
        Tester (identification, Egalite(p,p2),
            "dépiler(empiler(p,e)) = p");

        Tester (identification, p_Vide=Vide(p2),
```

```
            "vide(pileVide) = vrai; vide(empiler(p,e)) = faux");

        Adjoindre(p2,v4);
        Tester (identification, Premier(p2)=v4,
            "sommet(empiler(p,e)) = e");
        Reduire (p2);

        -- Règles
        if p_Vide then
            Tester (identification, Longueur(p2)=0,
                "profondeur(pVide) = 0");
        end if;
        Adjoindre (p2,v0);
        Tester (identification, Longueur(p2)=1+Longueur(p),
            "profondeur(empiler(p,e)) = 1 + profondeur(p)");

        -- Finalisation
        Liberer (p2);
    end Tester_Axiomes_Et_Regles;

begin -- Piles_AS_T00

    declare
    --  Identification de l'essai
        programme : constant String := "Piles_AS_T00";
        module :    constant String := "Piles_AS";

    --  Partie invariante en fonction de l'essai
        libelle :   constant String := " - Essais unitaires du module ";
        reussite :  constant String := "Essai concluant";

    --  Variables nécessaires aux essais
        p :     Pile;

    begin
        TEXT_IO.Put_Line (programme & libelle & module);

        Allouer (p);
        Tester_Axiomes_Et_Regles ("()", p, vrai);
        Adjoindre (p, v0);
        Tester_Axiomes_Et_Regles ("(v0)", p, faux);
        Reduire (p);
        Adjoindre (p, v3);
        Adjoindre (p, v2);
        Adjoindre (p, v1);
        Tester_Axiomes_Et_Regles ("(v1,v2,v3)", p, faux);
        Liberer (p);

        if globalement_Concluant then
            TEXT_IO.Put_Line (reussite);
        end if;
    end;

end Piles_AS_T00;
```

0.10 Questions

0.10.1 Q1

Énumérer et motiver les différentes parties devant figurer dans une définition d'interface de routine (module). Donner un exemple.

Spécification d'une routine

- ◊ **Nom** : nom de la de la routine, significatif mais respecte les conventions applicables (langage de programmation, environnement de développement, normes internes de l'organisation, unicité au sein de du composant de référence).
- ◊ **But** : but de la fonction, exposé succinctement et clairement.
- ◊ **Entrées** : paramètres susceptibles d'être consultés ; avec indication du domaine de valeur (type).
- ◊ **Sorties** : paramètres susceptibles d'être modifiés ; avec indication du domaine de valeur (type).
- ◊ **Retour** : dans le cas d'une fonction, paramètre associé à la valeur de retour ; normalement il ne devrait pas y avoir paramètres de sortie ***et*** retour.
- ◊ **Antécédent** (pré-conditions) : condition qui doit être respectée préalablement à l'exécution de la routine afin d'en garantir le résultat.
- ◊ **Conséquent** (post-conditions) : condition dans laquelle le système est laissé après l'exécution de la routine.
- ◊ **Commentaire** : ne doit en aucun cas paraphraser ce qui précède (notamment pour éviter les incohérences) mais plutôt motiver le choix de l'interface

Exemple

N'importe quel exemple du TA Piles programmé en Ada (voir laboratoire 1)

0.10.2 Q2

Pourquoi déterminer les critères de conception avant de concevoir un module?

Réponse

Les critères sont aussi importants que la spécification fonctionnelle elle-même car on ne conçoit jamais sans buts, sans contraintes. Ce sont les critères qui permettent de départager les choix multiples en cours de conception. Sans critère le concepteur est condamné à explorer tous les chois et toutes leurs ramifications.

0.10.3 Q3

Pourquoi est-il plus facile de concevoir une interface de module si un type abstrait a été défini au préalable?

Réponse

(a) La définition préalable d'un TA permet de séparer la tâche de concevoir l'interface de celle de définir les opérations. Un même TA peut induire de nombreuses interfaces différentes en fonction des critères de conception et de l'environnement de développement ciblé.
(b) Elle permet aussi d'exprimer formellement les antécédents et conséquents de l'interface de façon compacte et non ambigüe.

0.10.4 Q 4

Pourquoi s'intéresse-t-on à définir des noyaux dans les types abstraits ?

Réponse

(a) Un noyau permet en général de réduire le nombre d'axiomes requis pour définir un TA.
(b) S'il y a plusieurs noyaux, un choix approprié permet souvent de simplifier les axiomes eux-mêmes (voir le TA Queues).
(c) Les noyaux déterminent différents jeux d'opérations de base permettant éventuellement au concepteur de réduire son effort initial de mise en oeuvre.