TYPES ALGEBRIQUES ET SEMANTIQUE DES LANGAGES DE PROGRAMMATION

D. BERT (1)

ABSTRACT : The notion of abstract data types appears in recent languages (CLU, ALPHARD, EUCLID, ADA), for reasons of modularity and protection. Although an algebraïc specification of abstract data types has been developped (Guttag, ADJ), the connection between program semantics and data type semantics has not yet been established, like it was made for the axiomatization of PASCAL. In this paper, we show that the language semantics is composed by a fixed part : the control structure semantics and by an extensible part : the data type semantics. As a consequence, we can, for example, deduce the axiom of the assignement to a subscripted variable from the axioms of the array data type. Finally, we study how to prove that a representation module of an abstract data type is correct w.r.t. its algebraic specification i.e., that it satisfies the algebraic equations.

---

(1) Laboratoire IMAG, BP 53X - 38041 GRENOBLE cédex - France

1. INTRODUCTION

La notion de types abstraits dans les langages de programmation s'est révélée être un outil intéressant pour la structuration, la lisibilité et la vérification des programmes [17][18][25].

L'abstraction consiste essentiellement (1) à caractériser le type par un ensemble de fonctions (procédures ou opérateurs), (2) à cacher la représentation sous-jacente. Les langages récents [16][1] possèdent tous un mécanisme d' "encapsulation" de déclarations et par là même, un moyen de définir des types abstraits. Parallèlement, des méthodes de spécification des types abstraits ont été développées:ce sont essentiellement la méthode de Hoare [14] que nous appellerons "axiomatique", qui est utilisée systématiquement en ALPHARD, et la méthode "algébrique" [9][11][12] qui a suscité beaucoup de travaux récents. L'une et l'autre de ces deux méthodes permettent de spécifier les types abstraits et leurs opérations, puis, étant donné un module de représentation, de faire la preuve que ce module satisfait les spécifications.

Le but de cet article est de montrer comment on peut réunir les avantages des deux méthodes dans la spécification, la programmation et la vérification des types. On propose (1) de spécifier les types abstraits à l'aide de la méthode algébrique ; (2) de représenter les types à l'aide de modules algorithmiques spécifiés axiomatiquement ; (3) de montrer que la représentation satisfait les axiomes de la spécification par un processus automatisable.

Mais pour ce faire, il faut passer de la spécification algébrique du type abstrait à une spécification "algorithmique" qui puisse être utilisée dans un langage de programmation classique ; aussi on traitera successivement les points suivants : au §2, on associe à chaque procédure du type abstrait algorithmique une "sémantique" exprimée à l'aide des opérateurs algébriques. On aborde aussi le problème de la spécification de certains opérateurs de base, génériques pour une classe de types, comme l'affectation. Le §3 rappelle les éléments d'une représentation d'un type abstrait ainsi que la spécification de cette représentation ; ensuite on donne la technique de vérification d'un module de représentation par rapport à la spécification algébrique.

2 - SEMANTIQUE ALGEBRIQUE DES TYPES ABSTRAITS ALGORITHMIQUES

2.1. Types abstraits algébriques

Voici comme premier exemple de type abstrait algébrique le type tableau qui, pour simplifier, est de taille infinie ; les indices sont du type nat (entiers positifs). Les éléments du tableau sont d'un type formel noté elem. Suivant la syntaxe donnée en [5] qui suit de près [6], le type tableau est défini algébriquement par :

```
type Tableau[elem:tyfo] ;
      tvide : →tableau[elem]
      ranger : tableau[elem] × nat × elem → tableau[elem]
      val : tableau[elem]× nat → elem
exceptions
      indefini : →elem
eqns
      val(ranger(TAB,I,M),J)=
         si eq(I,J) alors M sinon val(TAB,J) fsi
      ranger (ranger(TAB,I,M),J,N)=
         si eq(I,J) alors ranger(TAB,J,N)
         sinon ranger (ranger(TAB,J,N),I,M) fsi
      val(tvide,I) = indefini
fin-type
```

Pour compléter l'exemple, on doit donner les types Nat et Bool :

```
type Nat ;
      0  : →nat
      s  : nat → nat                          ≠ successeur
      +  : nat × nat → nat
      eq, inf : nat × nat → bool              ≠ égal, inférieur strict
eqns
      x+0 = x
      x+s(y) = s(x+y)
      eq(0,0) = vrai
      eq(0,s(x)) = faux
      eq(s(x),0) = faux
      eq(s(x),s(y)) = eq(x,y)
      inf(x,0) = faux
      inf(0,s(x)) = vrai
      inf(s(x),s(y)) = inf(x,y)
fin-type ;
type Bool ;
      vrai :  →bool
      faux :  →bool
        ¬  : bool → bool
        ∧  : bool × bool → bool
eqns
     ¬ vrai = faux
     ¬ faux = vrai
      b ∧ vrai = b
      b ∧ faux = faux
fin-type
```

D'un point de vue formel, un type ainsi défini par des opérateurs Σ et des équations E est une algèbre isomorphe à l'algèbre initiale dans la catégorie des Σ-E-algèbres [9].

D'autre part, on a introduit les termes d "erreur" ou "exceptions", comme en [10]; enfin, les instances particulières de tableau (ex.: Tableau[Nat]) sont les algèbres qui étendent l'algèbre du type passé en paramètre (ex.: Nat) par les opérations spécifiques des tableaux. Ces algèbres sont aussi initiales [24]. D'un point de vue pratique, on utilise l'algèbre des termes (ou algèbre de Herbrand) pour représenter les éléments des types abstraits ; les opérateurs des types abstraits sont donc des symboles fonctionnels (dans lesquels on inclut les symboles de prédicat). Toute expression bien formée représente un élément d'un type abstrait ; les équations définissent des classes d'équivalence entre ces termes.

Notes : - On utilisera le symbole "=" pour la congruence engendrée par les équations (au lieu de "≡" qui est souvent utilisé dans ce cas). Pour éviter les confusions, l'opérateur d'égalité dans Nat est noté "eq".

- On distingue, pour un type donné, l'identificateur de type (anglais "sort que l'on note en minuscule (ex.:nat) de l'algèbre définie (ex.: Nat).

2.2. Sémantique d'un langage algorithmique

On sait comment, à l'aide de la sémantique dénotationelle [19][21][22] associer une "signification" à un programme. Le mini-langage que l'on considère comporte des déclarations de variables, des instructions et des expressions. Seul le type Bool = {vrai,faux} est nécessaire pour déterminer la sémantique des constructions générales. L'environnement associe à chaque identificateur sa "dénotation" et son type puisque le contrôle des types est essentiel dans le langage. L'état de mémoire associe à chaque variable sa valeur ; la notion de variable est préférable à celle de "location" car elle donne un point de vue plus "abstrait" de la mémoire [7].

Syntaxe générale :

```
Cmd ::= début Decl ; Cmd fin
      | Cmd ; Cmd | si Exp alors Cmd sinon Cmd fsi
      | tantque Exp faire Cmd refaire | rien
      | Ide(Exp,Exp,...)
Decl ::= Decl ; Decl | Ide:Ty
Exp ::= Ide | Ide(Exp,Exp,...)
```

Domaines sémantiques : Selon l'habitude, on donne les domaines sémantiques avec leur méta-variable standard. Les domaines élémentaires (Ide, Var, Bool, Ty) doivent être complétés par $\perp$ et $\top$ afin de leur donner une structure de treillis.

I : Ide	identificateurs
χ : Var	variables
Den=Var+Pr+F+{sf,m}	objets identifiables (dénotations)
β : V=Bool	valeurs de types abstraits
τ : Ty={bool}	identificateurs de types abstraits
TPr = Ty^* → Bool	types de procédures
TF = Ty^* → Ty	types de fonctions
Pr = $(V^* \times Var)$ → Mem → Mem	domaine des procédures
F = V^* → V	domaine des fonctions
ρ : Env = Ide → Den × T	environnement
σ : Mem = Var → V	état mémoire
T = Ty+TPr+TF	

Fonctions sémantiques d'interprétation

$\mathcal{S}$: Cmd → Env → Mem → Mem	interprétation des commandes
$\mathcal{D}$: Decl → Env → Mem → (Env × Mem)	interprétation des déclarations
$\mathcal{E}$: Exp → Env → Mem → V	interprétation des expressions
$\mathcal{T}$: Exp → Env → Ty	type des expressions

L'interprétation des constructions du langage est tout à fait classique, sauf pour les éléments donnés ci-après. On note :

. init(τ) : la valeur initiale d'un objet de type τ (cf. 2.4)

. $\sigma\,[\beta/\chi]$: la valeur de la variable χ est remplacée par β dans l'état σ ,

. si $\alpha \in$ Den × T, Den(α) et T(α) sont les projections sur chacune des composantes du produit cartésien.

$\mathcal{D}$ [[I:τ]] ρσ = ρ'σ' avec ρ' = ρ ∪ <I,χ,τ >
σ' = σ ∪ <χ,init(τ)>
χ = nouvelle_variable

$\mathcal{S}$ [[I(Exp$_1$,Exp$_2$,...)]] ρσ =
cas Den (ρ[I]) ∈ Pr alors
si T(ρ[I])(T[[Exp$_1$]] ρ, T[[Exp$_2$]] ρ,...)
alors Den(ρ[I])(E[[Exp$_1$]] ρσ, E [[Exp$_2$]] ρσ,...) σ
sinon $\perp_{Mem}$ fsi
|Den(ρ[I])=m alors
si T(ρ[I])(T[[Exp$_1$]] ρ, T[[Exp$_2$]] ρ ,...)
alors soit I' le paramètre modifié et χ=Den(ρ[I'])
σ[I(E [[Exp$_1$]] ρσ, E[[Exp$_2$]] ρσ,...)/χ]
sinon $\perp_{Mem}$ fsi
sinon $\perp_{Mem}$ fcas

E [[I]] ρσ=
cas Den(ρ[I]) ∈ Var alors σ[Den(ρ[I])]
|Den(ρ[I]) ∈ F alors Den(ρ[I])
|Den(ρ[I]) = sf alors I
sinon · $\perp_V$ fcas

E [[I(Exp$_1$,Exp$_2$,...)]] ρσ =
cas Den(ρ[I]) ∈ F alors Den(ρ[I])(E[[Exp$_1$]] ρσ, E[[Exp$_2$]] ρσ,...)
|Den(ρ[I]) = sf alors I(E[[Exp$_1$]] ρσ, E[[Exp$_2$]] ρσ,...)
sinon $\perp_V$ fcas

T[[I]] ρ = si Den(ρ[I]) ∈ Var+F+{sf}
alors T(ρ[I]) sinon $\perp_{Ty}$ fsi

T [[I(Exp$_1$,Exp$_2$,...)]] ρ =
si Den(ρ[I]) ∈ F+{sf}
alors T(ρ[I])(T [[Exp$_1$]] ρ, T [[Exp$_2$]] ρ,...) sinon $\perp_{Ty}$ fsi

2.3. Effet d'une déclaration de type algébrique

Pour l'instant, le domaine des valeurs du langage de base ne comprend que les valeurs de vérité, nécessaires pour définir la sémantique des constructions générales. La façon d'étendre le domaine de ces valeurs est de définir des types abstraits algébriques. Il n'est pas utile ici de donner la définition formelle d'une déclaration de type ; disons simplement que si Dom est l'union des domaines sémantiques du langage, une déclaration de type est interprétée par :

Decl-type : type → Dom → Dom

Par exemple, après les déclarations de type du §2.1., les domaines sémantiques modifiés sont :

V = Bool + Nat + Tableau[V]

Ty = {bool,nat,λτ.tableau[τ]}

Evidemment, une déclaration de type est valide si elle est une extension (aux symboles d'erreur près) des types existants [9][10].

On veut maintenant, dans le langage de programmation avoir des expressions de type bool, nat ou tableau, déclarer des variables de ces types et avoir des instructions classiques comme l'affectation, etc... sur ces variables. Pour cela, il faut étendre l'environnement par des déclarations appropriées, comme il est indiqué au paragraphe suivant.

2.4. Interface algorithmique d'un type abstrait

Pour passer d'un type algébrique à un type algorithmique on propose une démarche systématique qui consiste à choisir un sous-ensemble d'opérateurs générateurs, avec au moins un paramètre du type considéré, comme modificateurs de ce paramètre. Les paramètres modifiés ne pourront être que des variables (paramètres var en PASCAL, in out en ADA, etc...). Les autres opérateurs du type abstrait restent des symboles fonctionnels (ou des constantes) qui n'ont pas d'effet sur l'état mémoire, à l'inverse des modificateurs. La notion d'interface apparaît dans [20] ; celle de modificateur (modification) dans [8] . Voici comment on peut spécifier l'interface du type Tableau[elem] :

interface Tableau[elem:tyfo] ;
    init tvide ;
    proc ranger (mod tableau[elem],nat,elem) ;
Cmd ::=Ide[Exp_1] := Exp_2 => ranger(Ide,Exp_1,Exp_2) ;
Exp ::= Ide[Exp] => val(Ide,Exp) ;
fin-interface

Cette déclaration d'interface donne tvide pour initialiser les variables de type tableau ,et introduit ranger comme modificateur (m) ; par défaut, tvide, indéfini, val restent des symboles fonctionnels (sf) dans l'environnement. Pour se ramener aux écritures habituelles, on donne ensuite de nouvelles formes de commande et d'expression correspondant respectivement à ranger et val. On a maintenant dans l'environnement, les éléments suivants :

Env = ... + {<tvide,sf,λτ.tableau[τ]>,<indéfini,sf,λτ.τ>,
    <val,sf,λτ.tableau[τ] × nat → τ>,
    <ranger,m,λτ.tableau[τ] × nat × τ → vrai>

Le fait de traduire la sémantique des instructions en des expressions dans l'algèbre de Herbrand permet de résoudre facilement certains problèmes. Prenons par exemple le problème de l'affectation à des variables indicées [2][3].
On sait que l'assertion :

    vrai{a[a[2]] := 1} a[a[2]] = 1

n'est pas valide ; plutôt que de modifier la sémantique de l'affectation, on va décrire la plus faible précondition P, à partir des axiomes des tableaux, telle que

    P{a[a[2]] := 1} a[a[2]] = 1

soit valide. En notant "a" la variable associée à l'identificateur a et â sa valeur dans σ, on a :

$\sigma' = S[\![a[a[2]] := 1]\!] \rho\sigma$
$= \sigma[\text{ranger}(\hat{a},\text{val}(\hat{a},2),1)/a]$
et $P \equiv E[\![a[a[2]]]\!]\rho\sigma' = 1$

$\equiv \text{val}(\text{ranger}(\hat{a},\text{val}(\hat{a},2),1),\text{val}(\text{ranger}(\hat{a},\text{val}(\hat{a},2),1),2)) = 1$

En appliquant les axiomes des tableaux et en notant val(â,i) par â[i], on a :

$P \equiv$ si â[2] = (si â[2] = 2 alors 1 sinon â[2] fsi)
alors 1
sinon â[si â[2]=2 alors 1 sinon â[2] fsi] fsi = 1

d'où

$P \equiv$ si â[2]=2 alors â[1]=1 sinon vrai fsi

Le point important de cette sémantique "abstraite" est que les commandes élémentaires sont les modifications, dont l'interprétation, donnée au §2.2, peut être reformulée de la manière suivante : si m(x,y) est un modificateur où x représente le(s) paramètre(s) modifié(s) et y les autres paramètres, on a l'assertion :

$P[m(x,y)/x]\ \{m(x,y)\}\ P$

Le prédicat P est composé de prédicats élémentaires d'équivalence entre expressions dans le domaine des valeurs abstraites ; dans le premier prédicat, m est l'opérateur abstrait, alors que dans l'instruction il s'agit du modificateur. Les preuves que l'on peut faire dépendent des axiomes du type algébrique dans lequel m est défini. L'axiomatisation des types est donc essentielle à la sémantique du langage (cf.[15]); mais il n'est pas nécessaire que les types soient prédéfinis ; la définition d'un nouveau type est une extension véritablement "sémantique" du langage.

2.5. Propriétés génériques

On appelle "propriétés", des spécifications algébriques (et leur interface algorithmique) qui peuvent être attachés à une classe de types effectifs. Ces spécifications (comme les types) comprennent un ensemble d'opérateurs formels et les équations qui les caractérisent. Les propriétés sont relatives à un type formel ; soit la propriété p(t:<Σ,E>) du type formel t avec les opérateurs Σ et les équations E ; on dit qu'un type effectif τ:<Σ_1,E_1> possède la proriété p(t:<Σ,E>) ssi il existe un sous-ensemble d'opérateurs de Σ_1 de mêmes fonctionnalités que ceux de Σ dans lesquelles t est substitué par τ, et si les équations E peuvent être démontrées par induction à partir des équations de E_1. Cette idée de propriétés caractérisant des classes de types apparaît dans [23] et est formulée dans [4] ; elle est analogue à la caractérisation des paramètres formels de procédures de théories dans [6]. Un type effectif peut hériter une propriété, c'est-à-dire, avoir, en plus de ces opérateurs propres, les opérateurs de la propriété (à la substitution de type près). Donnons quelques exemples de propriétés :

1 - Expression conditionnelle :

```
prop Exp-cond sur t : tyfo ;
     si : bool x t x t → t
eqns
     si(vrai,x1,x2) = x1
     si(faux,x1,x2) = x2
```

```
interface
    Exp::=si Exp1 alors Exp2 sinon Exp3 fsi => si(Exp1,Exp2,Exp3)
fin-prop ;
```

2 - Affectation

```
prop Aff sur t:tyfo ;
    aff : t x t → t
eqns
    aff(x,y)=y
interface
    proc aff(mod t,t) ;
    Cmd ::= Ide := Exp => aff(Ide,Exp)
fin-prop ;
```

3 - Monoïde :

```
prop Monoïde sur t:tyfo ;
    ℓ : → t
    ∘ : t x t → t
eqns
    ℓ∘x = x
    x∘ℓ = x
    (x∘y)∘z = x∘(y∘z)
fin-prop ;
```

Réécrivons les types du §2.1. :

```
type Bool hérite Exp-cond,Aff ;
    vrai : →bool
    faux : →bool
    ...
type Nat hérite Exp-cond, Aff possède Monoïde(0,+) ;
    0 : →nat
    s : nat → nat
    + : nat x nat → nat
    eq,inf : nat x nat → bool
    ...
type Tableau [elem:tyfo] ;
    ...
```

Par ces déclarations, il est possible de faire des affectations à des variables de type nat ou bool, mais l'affectation de tableaux n'est pas permise. Pour que la clause possède soit valide, on doit vérifier les équations de Monoïde (0,+) :

```
0+x = x
x+0 = x
(x+y)+z = x+(y+z)
```

A titre d'exercice, étudions la sémantique de l'affectation (pour x et y de même type) :

$$\begin{aligned} S[\![\, x:=y \,]\!]\,\rho\sigma &= S[\![\, aff(x,y) \,]\!]\,\rho\sigma \\ &= \sigma[aff(E[\![\, x \,]\!]\,\rho\sigma,\ E[\![\, y \,]\!]\rho\sigma)/Den(\rho[x])\,] \\ &= \sigma[E[\![\, y \,]\!]\,\rho\sigma/Den(\rho[x])\,] \qquad \text{par les axiomes de Aff} \end{aligned}$$

En clair, on obtient l'état dans lequel la valeur de la variable x est remplacée par la valeur de l'expression y, ce qui est la sémantique classique de l'affectation simple.

3 - VERIFICATION DE LA REPRESENTATION D'UN TYPE ABSTRAIT

Soit le type ensemble d'entiers naturels entre 0 et 100, que l'on peut définir par:

```
type Ens hérite Exp-cond ;
   e-vide : → ens
   ins : nat × ens → ens                 insertion d'un élément
   enl : nat × ens → ens                 suppression d'un élément
   app : nat × ens → bool                appartenance
eqns
   ins(x1,ins(x2,y)) = ins(x2,ins(x1,y))
   ins(x1,ins(x2,y)) = si eq(x1,x2) alors ins(x2,y) fsi
   enl(x1,e-vide) = e-vide
   enl(x1,ins(x2,y)) = si eq(x1,x2) alors enl(x1,y)
        sinon ins(x2,enl(x1,y))fsi
   app(x1,e-vide) = faux
   app(x1,ins(x2,y)) = si ¬inf(x1,100) alors faux
        sinon si eq(x1,x2) alors vrai
              sinon app(x1,y) fsi fsi
interface
   init e-vide ;
   proc ins(nat,mod ens) ;
   proc enl(nat,mod ens) ;
fin type ;
```

On peut maintenance écrire des programmes utilisant des ensembles et imaginer que le système qui supporte ce langage est capable d'exécuter de tels programmes, mais on comprend que la représentation des ensembles par les termes de Herbrand n'est pas très efficace. Aussi, il doit être possible de définir des représentations de types abstraits, c'est-à-dire des algèbres isomorphes à l'algèbre des termes, mais dont les opérations, construites à partir d'objets plus primitifs, s'exécutent de manière plus efficace. Le § 3.1. indique comment représenter un type abstrait et comment spécifier cette représentation ; le § 3.2. donne les outils qui permettent de vérifier qu'une représentation satisfait l'axiomatisation abstraite.

3.1. Module de représentation

Le module de représentation est une implantation du type algorithmique. Comme en ADA [1], il comprend la fonction de représentation des objets et les corps des procédures et des fonctions. Mais l'on doit aussi donner une spécification plus complète de ce module pour faire les vérifications du § 3.2. Suivant [14][25] , on donne aussi :

- l'invariant concret,
- les pré et post conditions des procédures et fonctions de représentation.

L'apport de Guttag [11] est de remarquer qu'il faut adjoindre un prédicat d'égalité qui spécifie l'égalité des objets représentés.

Pour l'exemple choisi, représentons les ensembles par des tableaux de booléens ; le module de représentation s'écrit :

```
module  rep-ens représente ens par tableau [bool] ;
     ≠ ≠  dans le module, le type ens est équivalent à tableau [bool] au sens
          d'ALGOL 68 ;
     invariant (r:ens) est (∀i:nat)(i<100) non val(r,i) = indéfini ;
     égalité (r1,r2:ens) est (∀i:nat)(i<100) val(r1,i) = val(r2,i) ;
     post r = e-vide est (∀i:nat)(i<100) val(r,i)=faux ;
     post r = ins(i,t) est r = ranger(t,i,vrai) ;
     post r = enl(i,t) est r = ranger(t,i,faux) ;
     post b = app(i,t) est b = si i≥100 alors faux sinon val(t,i) fsi
implémentation
     fonction e-vide → r:ens ;
        début i:nat ;
             tantque inf(i,100) faire
                    r[i]:=faux ;
                    i:=s(i)
             refaire
        fin ;
     proc ins(i:nat,mod t:ens) ; t[i]:=vrai ;
     proc enl(i:nat,mod t:ens) ; t[i]:=faux ;
     fonction app(i:nat,t:ens) → b:bool ;
        si ¬inf(i,100) alors b:=faux
        sinon b:=t[i] fsi ;
fin-module
```

Les assertions de la spécification sont des formules logiques du premier ordre qui portent sur des équivalences dans les domaines abstraits de la représentation. Les vérifications de cohérence à l'intérieur du module de représentation sont essentiellement celles d'ALPHARD [25]. De plus, le prédicat d'égalité doit satisfaire les propriétés de l'égalité. Le point nouveau est de voir comment on peut montrer que les procédures d'implantation satisfont les équations algébriques abstraites.

3.2. Vérification des axiomes algébriques

Le principe de cette vérification [13] est de transformer les équations algébriques du type abstrait en formules logiques construites à l'aide des opérateurs abstraits de la représentation, et de montrer que ces formules sont valides grâce aux axiomes des types de la représentation. L'opérateur d'interprétation des équations est de la forme:

I = {équation} → {formule logique}

En fait, on divise cet opérateur I en plusieurs opérateurs auxiliaires :

I(pg=pd) = spec(pg=pd) ⊃ repc(pg,pd)

avec

1) spec(pg=pd) = spec(pg) ∧ spec(pd)
2) spec(si exp alors pd_1 sinon pd_2 fsi) = spec(expr) ∧ spec(pd_1) ∧ spec(pd_2)
3) spec(si exp alors pd fsi) = spec(expr) ∧ spec(pd)

Soit A le type abstrait que l'on représente ;

4) ∀ occurrence de symbole fonctionnel f :

$$\text{spec}(f(\alpha)) = \begin{cases} \text{spec}(\alpha) \wedge (\underline{\text{pre}}\ f(\text{rep}(\alpha)) \supset \underline{\text{post}}\ r=f(\text{rep}(\alpha))) & \underline{\text{ssi}}\ f \in A \\ \text{spec}(\alpha) & \underline{\text{sinon}} \end{cases}$$

Si α est une liste de paramètres α_1 , α_2 ... ,

on a spec(α) = spec(α_1) ∧ spec(α_2) ∧ ...

Dans la post-condition, r est une variable liée notée res(f).

5) ∀ variable x :

$$spec(x) = \begin{cases} \underline{invariant}(x) & \underline{ssi}\ x \text{ est de type } A \\ \underline{vrai} & \underline{sinon} \end{cases}$$

D'autre part, la fonction repc est définie par :

1) $repc(expr_1, \underline{si}\ expr_2\ \underline{alors}\ pd_1\ \underline{sinon}\ pd_2\ \underline{fsi})$
$= (rep(expr_2) \stackrel{E}{=} vrai \supset repc(expr_1, pd_1))$
$\wedge (rep(expr_2) = faux \supset repc(expr_1, pd_2))$

2) $repc(expr_1, \underline{si}\ expr_2\ \underline{alors}\ pd\ \underline{fsi})$
$= (rep(expr_2) = vrai \supset repc(expr_1, pd))$

3) $repc(expr_1, expr_2) = E(rep(expr_1), rep(expr_2))$
avec

$$E(x,y) = \begin{cases} \underline{égalité}(x,y) & \underline{ssi}\ type(x) = type(y) = A \\ x=y & \underline{sinon} \end{cases}$$

4) ∀ occurrence de symbole fonctionnel f

$$rep(f(\alpha)) = \begin{cases} res(f) & \underline{ssi}\ f \in A \\ f(rep(\alpha)) & \underline{sinon} \end{cases}$$

5) ∀ variable x :
$rep(x) = x$

Appliquons l'opérateur *I* pour vérifier si l'implantation satisfait les équations des ensembles.

(ex.1) I(app(x1,e-vide)=faux)
=> (post b= app(x1,r) ∧ post r=e-vide) ⊃ b=faux
=> (b = si x1 ≥ 100 alors faux sinon val(r,x1)fsi)
∧((∀i:nat)(i<100) val(r,i)=faux) ⊃ b=faux
=> vrai.

(ex.2) I(enl(x1,ins(x2,y)) = si eq(x1,x2) alors enl(x1,y)
sinon ins(x2,enl(x1,y)) fsi)
=> (post r1 = enl(x1,r2) ∧ post r2 = ins(x2,y) ∧ post r3 = enl(x1,y)
∧ post r4 = ins(x2,r5) ∧ post r5 = enl(x1,y) ∧ invariant(y))
⊃ [(eq(x1,x2) = vrai ⊃ égalité(r1,r3))
∧ (eq(x1,x2) = faux ⊃ égalité(r1,r4))]
=> invariant(y) ⊃
[(eq(x1,x2) = vrai ⊃
égalité(ranger(ranger(y,x2,vrai),x1,faux),ranger(y,x1,faux)))
∧(eq(x1,x2) = faux ⊃
égalité(ranger(ranger(y,x2,vrai),x1,faux),ranger(ranger(y,x1,faux),
x2,vrai)))]
=> vrai (par substitution et application des axiomes des tableaux).

Note : Dans l'exemple, les pré-conditions sont le prédicat vrai ; s'il en était autrement, elles serviraient à valider des équations qui spécifient des résultats erreur ou exception (cf. [5]).

On retrouve ici la méthode de vérification classique de représentation d'un type abstrait:

- la fonction de représentation définit le domaine B qui sera l'algèbre de la représentation ;
- les procédures et fonctions de la représentation définissent une algèbre dérivée de B : dB qui appartient à la catégorie du type à représenter ;
- l'invariant concret restreint le domaine de la représentation (I_c(dB)) ;
- le prédicat d'égalité est une relation de congruence dans le domaine de la représentation ($\stackrel{E}{=}_e$).

La preuve précédente consiste à montrer que l'algèbre quotient : $I_c(dB)/\stackrel{E}{=}_e$ satisfait les équations E du type abstrait représenté.

4 - CONCLUSION

On a présenté au §2 une définition sémantique d'un petit langage dans lequel on peut intégrer les définitions algébriques de types abstraits. Le point important est l'articulation entre les deux approches, qui se situe au niveau du choix de l'interface algorithmique. Dans le §3, on indique comment prouver qu'une implantation satisfait à la spécification donnée. Bien que le module de représentation soit spécifié à la manière d'ALPHARD, la méthode de vérification se rattache à la méthode algébrique.

On peut se poser la question de la possibilité et du bien-fondé d'automatiser un système de preuves aussi poussé, alors que l'on sait que l'écriture de spécifications est difficile (et parfois source d'erreurs). La réponse que l'on peut faire est la même que lorsqu'il s'agit de preuves de programmes : l'important n'est pas de pouvoir construire des vérificateurs fort coûteux et peu utiles, mais de donner au programmeur qui a le souci d'écrire des programmes corrects, des méthodes systématiques de vérification. Avec la possibilité de définir des types abstraits (sous forme de "package") dans un langage de grande diffusion comme ADA, de telles méthodes semblent nécessaires.

5 - BIBLIOGRAPHIE

[1] Preliminary ADA Reference Manual
SIGPLAN Notices, vol.14, 6, part A, June 1979

[2] J.W. de BAKKER
Correctness proofs for assignment statements
Report IW 55/76, Mathematical Center, Amsterdam, 1976

[3] J.W. de BAKKER
Semantics and the foundations of program proving
Information processing 77, B.Gilchrist, ed., IFIP, 1977, Toronto

[4] BERT D., JACQUET P.
Generic abstract data types
Proc. 5th Annual III Conference, WG 2.1., IFIP, mai 1977

[5] BERT D.
La programmation générique: construction de logiciel, spécification algébrique et vérification
Thèse d'Etat, Université de Grenoble I, Labo.IMAG, 1979

[6] R.M.BURSTALL, J.A.GOGUEN
Putting theories together to make specifications
Proc. Fifth Int. Joint Conf. on artificial intelligence, 1977
MIT, Cambridge, Mass. pp. 1045/1058

[7] J.E.DONAHUE
Locations considered unnecessary
Acta Informatica 8, pp. 221/242, 1977

8] J.P.FINANCE
De la spécification abstraite d'une donnée à sa représentation en mémoire : les états successifs d'une information
Théorie et techniques de l'informatique, Actes du congrès AFCET, novembre 78, pp. 423/433.

[9] J.A.GOGUEN, J.W.THATCHER, E.G.WAGNER
An initial algebra approach to the specification,correctness and implementation of abstract data types
IBM Research report, RC 6487, octobre 1976.

[10] J.A.GOGUEN
Abstract errors for abstract data types
Proc. IFIP Working conference on formal description of programming concepts
MIT, 1977

[11] J.V.GUTTAG, E.HOROWITZ, D.R.MUSSER
Abstract data types and software validation
University of Southern California, report ISI/RR 76 48, août 1976

[12] J.V.GUTTAG, J.J.HORNING
The Algebraic specification of abstract data types
Acta Informatica, 10, pp. 27/52, 1978

[13] J.V.GUTTAG, J.STAUNSTRUP
Algebraic axioms, classes and program verification
Computer science department, University of Southern California
Rapport n°213 741 6288, mars 1978

[14] C.A.R.HOARE
Proof of correctness of data representations
Acta Informatica, 1, 4, 1972, pp. 271/281

[15] C.A.R.HOARE, N.WIRTH
An axiomatic definition of the programming language PASCAL
Acta Informatica, 2, pp. 335/355, 1973

[16] B.W.LAMPSON, J.J.HORNING, R.L.LONDON, J.G.MITCHELL, G.L.POPEK
Report on the programming language EUCLID
SIGPLAN Notices, vol.12, n°2, 1977

[17] B.LISKOV, S.ZILLES
Programming with abstract data types
Proc. of the ACM SIGPLAN Conference on very high level languages
SIGPLAN Notices, 9, 4, 1974, pp. 50/59

[18] B.LISKOV, A.SNYDER, R.ATKINSON, C.SCHAFFERT
Abstraction mechanisms in CLU
CACM, vol.20, n°8, pp. 564/576, 1977

[19] R.E.MILNE, C.STRACHEY
A theory of programming language semantics
Chapman and Hall, London, Wiley New-York, 1977

[20] R.NAKAJIMA, M.HONDA, H.NAKAHARA
Describing and verifying programs with abstract data types
Proc. of IFIP Working conference on the formal description of programming concepts, MIT (1977)

[21] J.E.STOY
Denotational semantics : The Scott-Strachey approach to programming language theory
MIT Press, 1977

[22] R.D.TENNENT
The Denotational Semantics of programming languages
CACM, 19, 8, august 1976

[23] R.D.TENNENT
On a new approach to representation independent data classes
Acta Informatica, vol.8, fasc.4, 1977, pp. 315/324

[24] J.W.THATCHER, E.G.WAGNER, J.B.WRIGTH
Data type specification : parameterization and the power of specification techniques
Proc. 10th SIGACT Symposium on theory of computing, San Diego, CA, 1978

[25] W.A.WULF, R.L.LONDON, M.SHAW
Abstraction and verification in ALPHARD : introduction to language and methodology
ISI/RR, 74-76, University of California, 1976