DEFINITION DU LANGAGE Sum

JL.DORMOY

SYNTHESE :

Sum est la troisième brique du synthétiseur de programmes Descartes. Son langage est dit "algorithmique de bas niveau", c'est-à-dire qu'il ressemble fortement aux langages procéduraux habituels comme Pascal ou C. Son rôle est de servir de pivot aux traductions dans les langages propres aux applications (C, Cobol, AMPS).

La définition que nous donnons de Sum en fait un langage proche de C. Toutefois, certaines notions sont rajoutées. C'est notamment le cas des types “ensembles finis”, dont les éléments peuvent être d'un autre type, et qui peuvent servir d'indices d'un tableau. On pourra ainsi avoir un tableau indicé par des chaînes de caractères, par exemple. Un autre aspect est la hiérarchisation des types par une relation *sous-type*, qui rapproche Sum d'un langage objet. Cette relation définit une compatibilité entre types un peu spéciale. Cette particularité vise à la mise en œuvre aisée de certains clichés, comme *empiler* ou *dépiler*.

Sum, comme ErgoAlg, a une autre particularité qui l'éloigne des langages habituels : ses instructions ne sont pas nécessairement complètement ordonnées. De plus, l'ordre (partiel) entre instructions peut être spécifié de façon “non standard” par l'utilisation d'une notion de voisinage d'instruction. Cela est particulièrement utile pour les générateurs de programmes, dont les opérateurs ne peuvent agir que localement sur le programme construit, ce qui implique qu'il leur est très difficile de maintenir en permanence un ordre total.

Enfin, contrairement à C, Sum n'admet pas les appels récursifs. Cogito doit en effet avoir, tout en haut de la chaîne Descartes, "dérécursivé" la spécification.

Une autre note décrit la spécification formelle du traducteur Sum --> C.

1 Introduction

La description donnée ici est celle des éléments et de la sémantique d'un programme Sum. Comme pour ErgoAlg, sa syntaxe n'est pas définie, car Sum est en quelque sorte un langage "virtuel", servant uniquement de pivot avant génération dans un langage dépendant de l'application (C, Cobol, AMPS, …).

On verra qu'il est assez proche de C, avec toutefois quelques différences qui facilitent la tâche des générateurs de programmes.

La description commence par les types, qui sont ce qu'il y a de plus "exotique" en Sum, et par la compatibilité entre types. On verra en effet que Sum contient une notion de *sous-type* qui lui donne certains attributs des langages à objets. Puis nous décrivons les variables, les constantes et les expressions, puis l'ordre entre les instructions, qui n'est pas non plus très "standard", et enfin la déclaration des programmes Sum. Nous ajoutons quelques remarques sur l'allocation dynamique de mémoire.

La synthèse d'un programme C ou autre à partir d'un programme Sum n'est pas ici évoquée. Elle a été entièrement spécifiée en Descartes, et fait l'objet d'une autre note [Dormoy 93a].

2 Types en Sum

Une grande partie des différences entre Sum et C est relative aux types.

Comme dans C, on a les 4 catégories de types : scalaires, tableaux, enregistrements[1] et pointeurs. Les différences portent :

- sur l'existence de *types scalaires finis* définis par l'utilisateur, qui sont des ensembles finis dont les éléments sont des constantes d'un type homogène, mais quelconque;
- sur les tableaux, dont les ensembles d'indices peuvent être un type scalaire fini;
- sur l'ensemble des types, qui sont hiérarchisés par une relation *SousType*. Cette relation définit les compatibilités existant entre types (notamment les possibilités d'affectations entre quantités de types différents).

Nous décrivons maintenant chaque catégorie de type, puis la relation SousType.

2.1 Types scalaires

[1] *Structures* en C. Le terme enregistrement vient en fait de Pascal (Record). Les termes choisis pour Sum mélangent d'ailleurs allègrement ceux venant de C et de Pascal.

Sum connaît les types scalaires "habituels" : Entier, Reel, Caractere, Booleen[2]. Sum étant assez "proche" de la machine, ces types de base sont associés dans notre esprit aux quatre unités de stockage élémentaires disponibles sur les ordinateurs actuels : le mot (géré comme un entier ou comme un réel), l'octet et le bit[3]. Toutefois, on le verra, un entier *est* un réel.

Sum connaît aussi d'autres types scalaires, finis ceux-là, définis par l'utilisateur. Ces types sont des ensembles finis constitués de constantes d'un type quelconque, mais homogène.

Par exemple, le type suivant peut être défini en Sum :

JDS = {lundi,mardi,mercredi,jeudi,vendredi,samedi,dimanche}

Ici, ses éléments sont des constantes prises comme chaînes de caractères, représentées d'une manière ou d'une autre, par exemple comme des pointeurs vers un caractère.

Attention ! Il s'agit bien *d'ensembles*, donc non ordonnés (alors que les types scalaires énumérés de Pascal le sont).

Un type scalaire fini particulier est un intervalle fini d'entiers ou de caractères, comme [1,19] ou ['a','z'].

Remarque : L'ensemble des types mentionnés dans un programme Sum (un ensemble de procédures et fonctions Sum) constitue un type scalaire fini spécial, noté typeSum. On verra l'utilisation de ce "type ensemble des types" au §2.5.1. Bien évidemment, typeSum appartient à lui-même.

2.2 Tableaux

Les tableaux sont définis comme habituellement, c'est-à-dire qu'ils ont des ensembles d'indices et une image de types donnés, mais les ensembles d'indices sont des types scalaires finis quelconques. Ainsi, le type tableau suivant est correct :

Entier[JDS]

Une variable de ce type sera un tableau associant, par exemple, un nombre d'heures de cours à chaque jour de la semaine. On voit donc que l'on peut avoir un tableau dont les indices sont des chaînes de caractères (ou tout autre ensemble de constantes).

2.3 Enregistrements

Les enregistrements sont des types caractérisés par une collection de couples (champ, type du champ). Par exemple, le type enregistrement suivant est correct :

---

[2] Les types Caractere et Booleen sont en fait des types scalaires finis, cf. infra.

[3] Quoique la spécification actuelle du traducteur Sum considère les booléens comme des caractères, i.e. occupant un octet.

```
Record    NumSemaine : Entier
          CoursSemaine : Entier[JDS];
```

Ce type enregistrement contient deux champs, NumSemaine et CourSemaine, de types respectifs Entier et Entier[JDS] (un type tableau).

Il n'y a pas en Sum d'enregistrements variables.

2.4 Pointeurs

On peut définir des types pointeurs vers n'importe quel type. Un pointeur vers un certain objet mémorise une référence à cet objet, en général une adresse - physique ou virtuelle - d'un emplacement mémoire où cet objet est mémorisé (selon des conventions ad hoc). Cette vision suffit à savoir utiliser les pointeurs correctement, quoique Sum prenne certaines libertés avec la mise en œuvre physique de ses structures de données (libertés toutefois bien plus restreintes que pour ErgoAlg).

2.5 Hiérarchie sur les types

2.5.1 Motivation

La motivation est double : assurer une mise en œuvre facile de certains *clichés* par *généricité*, et définir convenablement les compatibilités entre types.

Les clichés sont des "patrons de programmes" qu'il suffit d'adapter pour traduire un morceau de programme ne dépendant que de caractéristiques identifiées. Par exemple, la fonction *empiler*, lorsque la pile est représentée par une liste chaînée, effectue toujours les mêmes actions, quels que soient les objets que l'on mémorise dans l'élément de pile. On aura alors pour *empiler* un cliché, qui ne dépend que du type de l'élément de pile.

Une autre manière de procéder est de considérer une procédure Sum fixe, ainsi définie :

```
empiler(p,t)

ptrPileGenerique = *pileGenerique
pileGenerique = Record
                suivant : ptrPileGenerique;

p, paux : ptrPileGenerique
t : typeSum

paux = alloue(t)
paux->suivant = p
```

[4] Ceci est la notation C de ce qui s'exprime en Pascal par
`paux@.suivant := p`

```
p = paux
```

On voit que la variable p passée en paramètre est déclarée comme étant d'un type liste "minimum" (il n'y a qu'un champ pour l'élément de pile inférieur), et que le type de la pile pour laquelle on appelle empiler est passé en paramètre (ce type est une constante, l'ensemble des types mentionnés dans un programme Sum constituant un type scalaire fini spécial, noté typeSum, cf. 2.1). L'allocation du nouvel élément de pile obéit donc au type de la pile pour laquelle on appelle empiler, mais les instructions génériques d'empilement sont définies pour le type générique.

Considérons le morceau de programme suivant :

```
ptrPileEJDS : pileEJDS
pileEJDS : Record
           EJDS : Entier[JDS]
           suivant : ptrPileEJDS;

pJDS : ptrPileJDS

…  empiler(pJDS,pileEJDS) …
```

Nous allons donner une définition qui permettra de considérer ptrPileEJDS comme sous-type de ptrPileGenerique. En particulier, l'unique champ de ptrPileGenerique (suivant) se retrouve dans ptrPileEJDS. Les opérations qui ne portent que sur ce champ peuvent donc être définies sur la structure "la plus pauvre", et utilisées avec une structure plus riche.

De plus, considérer ptrPileJDS comme sous-type de ptrPileGenerique est relativement naturel : le premier type caractérise des objets du même genre que ceux du second, sinon que l'on connaît des particularités supplémentaires. C'est du genre "les autos rouges sont des autos"[5].

Donc, tout çà fait qu'il est relativement naturel, et surtout pratique, de considérer que les piles (i.e. les enregistrements mentionnant le champ suivant) sont toutes sous-types de la pile générique.

Nous avions aussi mentionné un rapport entre les sous-types et la compatibilité des types. Dans l'exemple, la procédure empiler est appelée avec un paramètre sous-type du paramètre formel. Lors de l'appel, il y a une sorte d'affectation du paramètre appelant au paramètre formel. On a donc affectation d'un sous-type vers un sur-type. A la sortie de la procédure empiler, il y a affectation inverse, donc d'un sur-type vers un sous-type.

Nous autorisons donc dans Sum les affectations entre types différents, mais seulement de sur-type vers sous-type, ou l'inverse. Nous verrons cependant (§2.5.3) que l'affectation vers un sous-type peut conduire à une *erreur*.

(ou : la valeur du champ `suivant` de l'enregistrement pointé par `paux` doit prendre la valeur `p`).

[5] Ou plutôt les autos dont la couleur est précisée sont des autos.

Pour terminer, on voit que la notion de sous-type dans Sum reflète partiellement celle d'héritage dans les langages à objets. La fonction empiler est définie pour le type ptrPileGenerique, mais ses sous-types peuvent en "hériter".

2.5.2 Définition

Les types sont hiérarchisés par une relation d'ordre partiel, baptisée SousType, et notée $\langle$ dans ce qui suit . Cette relation se définit pour chaque catégorie de type par :

- Types scalaires : $t_1 \langle t_2$ ssi $t_1 \subset t_2$ (comme ensembles)
- Types tableaux : $t_1 \langle t_2$ ssi indice(t_1) $\langle$ indice (t_2) et
  image(t_1) $\langle$ image(t_2)
- Types enregistrements :
  $t_1 \langle t_2$ ssi champs(t_1) $\supset$ champs(t_2) et
  pour tout champ c commun à t_1 et t_2,
  valeur(t_1,c) $\langle$ valeur(t_2,c)
- Types pointeurs :
  $t_1 \langle t_2$ ssi typepointé(t_1) $\langle$ typepointé(t_2)

Pour être tout à fait correct, cette définition n'est en fait qu'une contrainte de "point fixe", car la définition des types peuvent contenir des circuits. On choisit de définir la relation SousType comme étant la solution maximale de cette équation de point fixe. Un exemple fera disparaître l'inconfort de ces vocables un peu compliqués. Dans l'exemple suivant, les types ptr1 et rec1 sont bien sous-types respectifs de ptr2 et rec2 dès que t1 est sous-type de t2.

```
ptr1 = *rec1
rec1 = Record
            champ : t1
            suivant : ptr1;

ptr2 = *rec2
rec2 = Record
            champ : t2
            suivant : ptr2;
```

En effet, on a

rec1 $\langle$ rec2 ssi t1 $\langle$ t2 et ptr1 $\langle$ ptr2

et

ptr1 $\langle$ ptr2 ssi rec1 $\langle$ rec2

d'où finalement

rec1 $\langle$ rec2 ssi t1 $\langle$ t2 et rec1 $\langle$ rec2

Cela est de la forme A ⇔ C et A, et il est clair que, si C est vrai, la valeur de vérité de A peut être choisie arbitrairement tout en respectant l'équivalence. C'est là que la "maximisation" intervient : on choisit dans ce cas A vrai, c'est-à-dire rec1 ⟨ rec2 (A) vrai dès que t1 ⟨ t2 (C) vrai. Par contre, si C est faux, A est faux sans ambiguïté.

Remarque : Le type Entier est sous-type du type Reel, puisque qu'Entier est considéré comme sous-ensemble de Reel.

2.5.3 Compatibilité entre types

Nous considérons comme légales les affectations d'un sous-type vers un sur-type, ou l'inverse. Nous définissons ici exactement comment se passe cette affectation. Cependant, l'affectation vers un sous-type peut produire une erreur. Nous considérons dans la suite deux variables v et V, de types t et T, où t est sous-type de T, et les affectations v = V et V = v. Il est clair que t et T sont de la même catégorie (scalaires, tableaux, etc)[6], et nous étudions chaque cas.

- Types scalaires

  L'affectation V = v ne pose aucun problème, puisque T contient t. Par contre, v = V est admise, mais produira une erreur si la valeur de V n'est pas élément de t.

- Types tableaux

  L'affectation V = v signifie que les valeurs de v pour tous ses indices sont recopiés aux indices correspondants de V (les autres indices de V étant laissés tels quels). L'affectation v = V ne recopie dans v que les valeurs de V pour les indices de t, ce qui peut produire une ou plusieurs erreurs, puisque pour chaque indice on a affectation vers un sous-type.

- Types enregistrements

  L'affectation V = v copie la valeur des champs communs à t et T de v dans les champs correspondants de V. v = V effectue la même chose (les champs de v qui ne sont pas champs de V étant laissés intacts), sauf qu'ici une ou plusieurs erreurs peuvent se produire, l'affectation de chaque valeur de champ étant une affectation vers un sous-type.

- Type pointeurs

  Les deux affectations v = V et V = v sont évidentes et ne produisent pas d'erreur. Cependant, ces affectations pourraient toutes deux produire dans la suite du déroulement du programme des erreurs, cette fois non contrôlées par Sum. Ainsi, si v et V pointent sur des enregistrements e et E, prendre la valeur de *v via un champ de e qui n'est pas champ de E après v = V conduira à un

[6] Sauf si l'un des deux, est de type scalaire fini, cf. infra.

"dépassement de mémoire", alors que l'instruction est syntaxiquement correcte, puisque v est bien de type *e. De même, prendre la valeur d'un indice qui n'existerait pas pour t de *V après V = v conduirait aussi à un dépassement de mémoire. Il est de la responsabilité du programmeur (et du système de synthèse de programme ErgoAlg) que cela ne se produise pas[7].

Remarques :

- Les définitions précédentes incluent la compatibilité entre entiers et réels. On peut affecter librement un entier à un réel, mais on ne peut affecter un réel à un entier que s'il est entier. Dans la traduction d'un programme Sum, le programme généré s'arrange pour vérifier que le réel est "suffisamment proche" de sa partie entière. Sinon, une erreur est produite. Si l'on n'est pas bien sûr du résultat, on pourra utiliser la fonction *ent* (partie entière).

- Ce qui vient d'être dit pour deux variables reste évidemment valable lorsqu'à droite de l'affectation on a n'importe quelle expression légale, et à gauche une expression affectable. Il faut alors considérer le type de ces expressions.

2.5.4 Cas particulier des types scalaires finis

Une variable d'un type scalaire fini a "deux types" : le type scalaire fini lui-même, et le type des constantes constituant ce type. Par exemple, une variable d du type JDS ci-dessus est de type JDS, mais peut aussi dans certaines circonstances être considérée comme étant de type chaîne de caractères.

On appelle *type générique* le type des constantes constituant un type scalaire fini. Ainsi, JDS est de type générique celui des chaînes de caractères (par exemple pointeur vers un caractère).

On considère qu'un type scalaire fini est sous-type de son type générique. Ainsi, on pourra affecter la variable d à une variable cc de type chaîne de caractères :

```
cc = d
```

L'affectation inverse est aussi possible :

```
d = cc
```

mais cette affectation produira une erreur si la valeur de cc n'est pas une des chaînes de caractères de JDS.

En fait, toutes les définitions de compatibilité définies précédemment sur la hiérarchie des sous-types reste valables, une fois que l'on considère un type scalaire fini sous-type de son type générique. Ainsi, on pourra affecter d à une variable ccc d'un type sur-type des chaînes de caractères (si un tel sur-type doit exister) :

---

[7] Toute manipulation de pointeur un peu "libre" a ses risques, les programmeurs en C le savent bien.

ccc = d

L'affectation inverse

d = ccc

est aussi possible, avec l'éventualité d'une erreur.

Mais si c est un sous-type *strict* des chaînes de caractères, les affectations

c = d

ou

d = c

ne seront possibles que si les types de c et de d sont comparables par la relation sous-type. Ce qui implique que c est lui-même d'un type scalaire fini, de type générique chaîne de caractères, et la comparabilité des types de c et d se ramène donc à leurs inclusions.

2.5.5 Affectations par héritage et héritage inverse

Plaçons nous dans le cas où un même type t a deux sous-types non comparables t_1 et t_2. Si v_1 et v_2 sont deux variables de types respectifs t_1 et t_2, les affectations

$v_1 = v_2$ et $v_2 = v_1$

sont illégales. On peut cependant s'en tirer, en utilisant une troisième variable v de type t, et en écrivant à la place de $v_1 = v_2$

$v = v_2$
$v_1 = v$

Sum ne pourrait faire ces transformations lui-même, car il pourrait exister plusieurs types t dont t_1 et t_2 "héritent".

2.5.6 Restrictions sur la hiérarchie des types

Pour des raisons pratiques de génération de code, certaines restrictions sont imposées sur la relation SousType, notamment entre les types enregistrements. La constrainte est que, si un type enregistrement e est sous-type de deux sous-types e_1 et e_2 (eux-mêmes enregistrements), alors e_1 est sous-type de e_2, ou e_2 est sous-type de e_1. Cette contrainte peut être comprise à partir de la spécification de la traduction Sum --> C (ou Cobol, …) [Dormoy 93a].

3 Variables, constantes et expressions

3.1 Variables

Il y a deux sortes de variables dans Sum : les variables globales et les variables locales. Une variable globale est connue de toutes les procédures et fonctions mentionnées (vraiment globale), une variable locale n'est connue que de la procédure ou fonction où elle est déclarée (vraiment locale).

Une variable a un type unique.

3.2 Constantes

Des constantes de n'importe quel type peuvent être définies (même des constantes de type pointeur). Il est interdit d'affecter une constante (même indirectement, en jouant avec les pointeurs, mais cela est laissé à la responsabilité du programmeur).

3.3 Expressions

Les expressions habituelles des langages de programmation comme Pascal ou C sont admises. Elles incluent :

- Les expressions relatives aux structures des types :
  - élément de tableau. Exemple : v[x,3]
  - accès à un champ d'un enregistrement. Exemple : v.suivant
  - déréférencement *. Exemple : *p est le contenu de l'adresse valeur de p.

- Les expressions utilisant un opérateur :
  - opérateurs arithmétiques +, -, *, /, div, mod, ent, …
  - opérateurs de comparaison ==, !=, <, <=, >, >=
  - opérateurs logiques et, ou, non

- Les appels de fonction. Exemple : empiler(pEJDS).

Il y a évidemment des contraintes de type sur l'usage des expressions, plus des contraintes "dynamiques", c'est-à-dire qu'une expression doit être évaluable au moment de son évaluation.

4 Instructions

Nous définissons d'abord les instructions autorisées en Sum. Nous définissons ensuite l'ordre entre instructions, qui aura la particularité d'être partiel et "non standard".

4.1 Instructions légales

Les instructions de Sum sont :

- L'affectation (pour les compatibilités entre quantités, cf. 2.5.4).
- Le test (Si … alors … Sinon).
- La boucle Tant que.
- return, qui renvoie la valeur d'une fonction et arrête son exécution (sans valeur, return peut être utilisé dans une procédure).
- break, qui interrompt la boucle la plus imbriquée le contenant.
- case of (switch en C). case v of … branche sur le ou les cas correspondant à la valeur de v. Les alternatives du case of ne sont pas a priori ordonnées.
- Cas de case of.
- L'appel de procédure.
- L'instruction vide (continue), qui ne fait rien.
- L'instruction multiple (un bloc d'instructions).

4.2 Ordre entre les instructions

4.2.1 Hiérarchie de blocs et ordre standard

A priori, l'ordre entre instructions n'est pas total. En effet, même si à l'entrée de ErgoAlg on dispose d'un programme complètement ordonné, les opérateurs servant à la synthèse du programme Sum ne vont pouvoir travailler que localement. Il serait donc très difficile de maintenir en permanence un ordre total des instructions du programme en construction. C'est d'ailleurs ce que nous faisons à la main : dans un programme partiellement écrit, on ajoute une nouvelle instruction "surtout avant celle-là", "mais après cette autre", etc[8].

Un programme Sum est donc composé d'instructions, qui sont soit des blocs d'instruction (instructions non élémentaires contenant d'autres instructions, comme les boucles, les tests ou les instructions multiples), soit des instructions élémentaires

[8] Quoique la syntaxe des langages de programmation implique l'existence d'un ordre total, pour l'essentiel contingent.

(comme les affectations ou les appels de procédures). Les blocs d'instructions *contiennent* d'autres instructions. Cela fournit une première structuration, hiérarchique, entre instructions. Cette structuration doit avoir la forme d'un arbre.

Un deuxième niveau de structuration est *l'ordre* entre instructions *d'un même bloc*. Dans Sum, nous définissons cette relation de précédence comme un *ordre partiel* (alors que, dans un langage usuel, il s'agit d'un ordre total sur un même bloc). Cela signifie que cette relation ne présente pas de circuit, mais deux instructions distinctes d'un même bloc peuvent ne pas être ordonnées entre elles. *Des instructions non ordonnées entre elles sont réputées pouvoir être échangées sans troubler la logique du programme*.

4.2.2 Relations de voisinage et ordre non standard

En théorie, ces deux relations devraient suffire pour décrire un programme. Cependant, elles ne sont souvent pas suffisantes pour le générer. En effet, certaines connaissances sont plus faciles à écrire si elles peuvent "glisser" de nouvelles instructions entre des instructions déjà générées, et ce à la bonne place. Il est donc utile de pouvoir décrire la position de nouvelles instructions localement par rapport à d'autres instructions, sans connaître l'ensemble du programme.

Pour cela, des "zones infinitésimales de proximité" peuvent être définies aux extrémités d'un bloc d'instruction et autour d'une instruction par les notions de *juste avant, juste après, au début, à la fin*. Ainsi, si une instruction *instr* appartient au bloc *bloc*, et si elle est *au début* de *bloc*, alors elle est avant toute instruction de *bloc* qui n'est pas *au début*. Plusieurs instructions peuvent être *au début*. L'ensemble des instructions *au début* d'un bloc est donc complètement séparé des instructions qui n'y sont pas.

De même, une instruction *instr1* peut être *juste avant* une instruction *instr2*, ce qui signifie qu'elle est avant *instr2*, et après toute instruction avant mais pas *juste avant instr2*. Les instructions *juste avant instr2* forment un ensemble complètement séparé des instructions qui ne le sont pas. Idem pour *fin* et *juste après*.

Ces notions ressemblent à s'y méprendre à l'ordre des réels non standards, où, si a est un infinitésimal, 1-a est strictement inférieur à 1, mais strictement supérieur à tout réel *standard* plus petit que 1. On a une notion de *voisinages infinitésimal* d'instructions.

Les instructions *au début* d'un bloc, *juste avant* une instruction, etc, peuvent être ordonnées entre elles. Elles peuvent aussi avoir des instructions *juste avant*, etc, elles.

5 Procédures et fonctions

Nous aurions pu commencer par là. Mais les choses sont ici assez proches de Pascal.

Les procédures ne renvoient pas de valeur, les fonctions si.

Les paramètres des procédures et fonctions sont passés "par adresse", c'est-à-dire que si le paramètre change de valeur dans le cours de l'exécution du sous-programme, alors ce changement est reflété sur le paramètre d'appel.

Mais, ce qui diffère d'un langage comme Pascal, il faut préciser si un paramètre est passé en *entrée seulement*, en *sortie seulement*, ou les deux, comme en ErgoAlg [Dormoy 93b]. En entrée seulement signifie que le paramètre de l'appel a une valeur bien définie, mais que le paramètre ne sera pas modifié par le sous-programme. En sortie seulement signifie que la valeur initiale du paramètre n'est pas utilisée dans le sous-programme, mais que cette valeur sera imposée ou modifiée. Les deux à la fois signifie que la valeur d'appel sera utilisée, et (potentiellement) modifiée. Ces précisions sont très utiles pour optimiser le programme généré par Sum.

6 Allocation de mémoire

La déclaration d'une variable implique automatiquement l'allocation de sa référence (son adresse), et de la place nécessaire à son contenu, justement à cette adresse.

Toutefois, cette allocation "s'arrête aux pointeurs", comme dans les langages contenant des pointeurs. Cela signifie que la place mémoire nécessaire au stockage du contenu de l'adresse valeur d'un pointeur n'est pas créée par la simple allocation du pointeur.

Une fonction spéciale d'allocation existe donc, nous la notons New, qui prend un type Sum en argument, alloue la place mémoire nécessaire au stockage d'un élément de ce type et renvoie l'adresse de ce morceau de mémoire. Par exemple

```
p : ptrPileGenerique
p = New(pileGenerique)
```

alloue la place nécessaire à un enregistrement de type pileGenerique (ici un mot machine), et renvoie l'adresse de cette zone, qui est ici mémorisée comme valeur de la variable p.

Tout cela n'est que très banal, sinon que (cf. §§ 2.1 et 2.5.1) les types utilisés dans un ensemble donné de programmes Sum forment un type scalaire fini, noté typeSum, qu'une variable peut être de ce type, et qu'il est donc possible d'avoir une instruction du genre

```
p = New(t)
```

où la valeur de la variable t est un type Sum. C'est cette dernière possibilité, avec la compatibilité entre types et sous-types, qui permet l'expression de clichés par généricité.

Références

[Dormoy, 1993a] Jean-Luc Dormoy. *Spécification formelle du composant Sum de Descartes*. Note EDF HI21/8343.

[Dormoy, 1993b] Jean-Luc Dormoy. *Définition du langage ErgoAlg*. Note EDF HI21/8340.