Par :: Yves Bailly :: http://www.kafka-fr.net ::

Le langage Ada 4 : utilisation des tableaux

Après avoir examiné les structures de contrôle le mois dernier, nous allons maintenant commencer à découvrir ce qui fait la véritable force du langage Ada : ses structures de données fondamentales, aussi simples que souples. Mais avant cela, je voudrais vous toucher un mot de la prochaine version du langage.

Ada 2005... ou 2006 ? Il semble en effet que le comité de standardisation estime que la prochaine norme Ada ne pourra être validée que début 2006.

La prochaine version du langage sera donc probablement désignée par « Ada2006 » (mais le nom « officiel » du langage demeure simplement Ada).

Voici un très bref aperçu de ce qui va changer [1], du moins ce qui me paraît le plus remarquable :

- Support pour les standards Unicode 2.0 et 4.0 jusque dans le code source (une constante écrite littéralement « π » a même été définie) ;
- Le principe des interfaces, introduit notamment par Java, permettra une forme limitée d'héritage multiple ;
- La bibliothèque standard du langage proposera des types conteneurs (listes, tableaux associatifs...), des fonctions et procédure de manipulation des fichiers et répertoires, ainsi que des outils mathématiques comme les vecteurs et les matrices ;
- Diverses améliorations pour les applications et systèmes multitâches et critiques.

Une bonne partie de tout cela est déjà disponible dans la dernière version du compilateur GNAT fournie avec la collection GCC 4.0, sortie tout récemment.

Si la lecture d'un standard ne vous effraie pas trop, une ébauche de ce que sera le prochain standard Ada [2] est disponible.

En attendant sa finalisation, nous continuerons à nous concentrer sur la norme Ada 95.

Types énumérés

Avant d'attaquer les tableaux, voyons les possibilités pour définir un type énuméré – similaire aux `enum` du C/C++.

Il s'agit de définir un type en donnant la liste exhaustive des valeurs qui le composent, sous la forme d'identifiants. Par exemple :

```
type Rgb is (Rouge, Vert, Bleu) ;
```

Ceci définit simplement un type nommé `Rgb`, dont les valeurs sont `Rouge`, `Vert` et `Bleu`. Ces valeurs sont implicitement ordonnées : Rouge < Vert est vrai, mais Vert > Bleu est faux. En fait, ce type `Rgb` est un type discret ressemblant fortement à un entier. Pour ce genre de types (tous les types énumérés et tous les types entiers), un ensemble d'attributs fonctionnels sont disponibles. Ils sont résumés dans le tableau ci-dessous, agrémenté d'un exemple.

Essayez de deviner ce qu'affiche ce petit programme...

Voici :

```
$ gnatmake enum.adb
$ ./enum
ROUGE
1
BLEU
ROUGE
BLEU
VERT
VERT
ROUGE
BLEU
```

:: Tab. 1 :: Attributs pour les types discrets ::

```
1 with Text_IO ; use Text_IO ;
2 procedure Enum is
3    type Rgb is (Rouge, Vert, Bleu) ;
4 begin
5    Put_Line(Rgb'Image(Rouge)) ;
6    Put_Line(Integer'Image(Rgb'Pos(Vert))) ;
7    Put_Line(Rgb'Image(Rgb'Val(2))) ;
8    Put_Line(Rgb'Image(Rgb'First)) ;
9    Put_Line(Rgb'Image(Rgb'Last)) ;
10   Put_Line(Rgb'Image(Rgb'Succ(Rouge))) ;
11   Put_Line(Rgb'Image(Rgb'Pred(Bleu))) ;
12   Put_Line(Rgb'Image(Rgb'Min(Rouge, Bleu))) ;
13   Put_Line(Rgb'Image(Rgb'Max(Rouge, Bleu))) ;
14 end Enum ;
```

`Image(a)`	Donne une chaîne de caractères représentant la valeur.
`Pos(a)`	Indice de la valeur dans la liste des valeurs du type.
`Val(i)`	Valeur du type d'indice `i` dans la liste des valeurs.
`First`	Première (plus petite) valeur du type.
`Last`	Dernière (plus grande) valeur du type.
`Succ(a)`	Valeur suivante de la valeur `a`.
`Pred(a)`	Valeur précédente de la valeur `a`.
`Min(a, b)`	La plus petite des deux valeurs données.
`Max(a, b)`	La plus grande des deux valeurs données.

N'est-ce pas merveilleux ? Les valeurs sont affichées, sans qu'il soit besoin de recourir à une fonction de correspondance laborieuse pour obtenir des chaînes. Du tableau précédent, on peut affirmer que, quelle que soit une valeur `X` d'un type discret `T`, on a toujours :

```
T'Val(T'Pos(X)) = X
T'Succ(X) = T'Val(T'Pos(X)+1)
T'Pred(X) = T'Val(T'Pos(X)-1)
```

Les positions des valeurs sont indicées à partir de 0. Naturellement, si on cherche à prendre la valeur précédente de la première valeur ou la valeur suivante de la dernière, on risque des ennuis, plus précisément la levée d'une exception nommée `Constraint_Error`. Par ailleurs, il n'est pas possible d'associer explicitement une valeur numérique à une valeur symbolique, comme cela se fait couramment en C/C++.

Nous allons maintenant voir que ces types énumérés peuvent servir à autre chose que simplement servir de réceptacles à des valeurs symboliques.

Les tableaux à une dimension

En Ada, il est d'usage de déclarer un type de tableau avant de s'en servir. Un tel type est déclaré par exemple ainsi :

```
type TTableau is array (Integer range 1..3) of Integer ;
```

Cette ligne déclare un type nommé `TTableau`, qui est un tableau (`array`) dont les indices sont du type `Integer`, compris entre 1 et 3 et dont les éléments sont de type `Integer`. Relisez bien la phrase précédente : la représentation des tableaux en Ada est assez différente que ce que l'on trouve dans la plupart des autres langages de programmation.

Première remarque : l'indice du tableau est typé. En C/C++, Python ou de nombreux autres langages, l'indice est implicitement un type entier « universel », sur la nature exacte duquel on ne s'attarde pas. Ici notre indice possède bien un type. Deuxième remarque : la définition du tableau précise l'intervalle de validité de l'indice (la partie `range 1..3`). Cela signifie que l'indice de la première valeur du tableau **n'est pas forcément 0**, contrairement à la plupart des langages. Ce premier indice peut être n'importe quelle valeur valide pour le type de l'indice – il en va naturellement de même pour la borne supérieure. Le type tableau `TTableau` est donc destiné à contenir 3 entiers, lesquels seront référencés par des indices compris entre 1 et 3 inclus. Un tableau de même taille aurait été obtenu en écrivant `range -3..-1`, sauf que les indices auraient tous été négatifs. Dans la pratique, la base d'indice préférée en Ada est 1 plutôt que 0. Notez que si la première valeur de l'intervalle est strictement inférieure à la deuxième (par exemple, `range 3..1`), le type correspondant ne pourra pas contenir la moindre valeur, bien qu'une telle déclaration soit syntaxiquement correcte. On a alors un type « vide ».

Voici deux autres manières de définir un type tableau, les deux types ayant exactement la même taille :

```
type Entier is new Integer range 1..3 ;
type TPetitTab is array (Entier)of Integer ;
type Rgb is (Rouge, Vert, Bleu) ;
type TRgbTab is array (Rgb) of Integer ;
```

La première ligne crée un nouveau type d'entiers `(Entier)`, dont les valeurs sont limitées entre 1 et 3. La deuxième ligne déclare un type tableau `TPetitTab`, mais sans donner explicitement l'intervalle des indices : celui-ci est tout simplement déduit du type de l'indice. La répétition du `range 1..3` est donc superflue. Ce type de tableaux contient trois entiers.

On retrouve en troisième ligne le type énuméré `Rgb` introduit plus haut. Voyez comment on l'utilise pour déclarer un type de tableau `TRgbTab` dont les indices sont de type `Rgb` : cela définit un tableau contenant trois entiers (car le type `Rgb` ne comporte que trois valeurs), les indices des éléments du tableau étant compris entre les valeurs `Rouge` et `Bleu`. Rien ne nous empêche de limiter l'intervalle des indices, même pour un type énuméré, par exemple :

```
type CoulComp is (Rouge, Vert, Bleu, Teinte, Saturation, Valeur) ;
type TRgbTab is array (CoulComp range Rouge..Bleu) of Integer ;
type THsvTab is array (Teinte..Valeur) of Integer ;
```

Les deux types ont exactement la même taille (ils contiennent tous deux trois entiers) et des indices de même type, mais les bornes valides pour les indices ne sont pas les mêmes. Remarquez la deuxième définition, qui fait l'économie du type de l'indice : cela n'est autorisé que parce qu'il n'y a aucune ambiguïté possible. Les identifiants `Teinte` et `Valeur` étant de type `CoulComp`, le type de l'indice du type tableau `THsvTab` est alors forcément de type `CoulComp`.

Utilisation des tableaux

Maintenant que nous avons déclaré plein de types de tableaux, déclarons quelques variables et utilisons-les dans un petit programme :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure Tableaux1 is
 3    type TTableau is array (Integer range 1..3) of Integer ;
 4    type Entier is new Integer range 1..3 ;
 5    type TPetitTab is array (Entier) of Integer ;
 6    type Rgb is (Rouge, Vert, Bleu) ;
 7    type TTab is array (Rgb) of Integer ;
 8    type CoulComp is (Rouge, Vert, Bleu, Teinte, Saturation, Valeur) ;
 9    type TRgbTab is array (CoulComp range Rouge..Bleu) of Integer ;
10    type THsvTab is array (Teinte..Valeur) of Integer ;
11 t1: TTableau ;
12 t2: TPetitTab ;
13 t3: TRgbTab ;
14 t4: THsvTab ;
15 t5: TTab ;
16 begin
17    for i in 1..3
18    loop
19       t1(i) := 2*i ;
20    end loop ;
21    for i in t2'First..t2'Last
22    loop
23       t2(i) := 3*Integer(i) ;
24    end loop ;
25    for i in t3'Range
26    loop
27       t3(i) := 0 ;
28    end loop ;
29    for i in THsvTab'Range
30    loop
31       t4(i) := 0 ;
32    end loop ;
33    for i in Rgb
34    loop
35       t5(i) := 0 ;
36    end loop ;
37 end ;
```

Chacune des boucles remplit simplement le contenu de l'une des variables déclarées lignes 11 à 15. Remarquez que l'indice pour accéder à un élément est donné entre parenthèses, et non entre crochets.

Si la première boucle ne présente rien de particulier, examinons les en-têtes des suivantes. Ligne 21, on fait appel aux attributs `First` et `Last`, que nous avons rencontrés dans la première partie. Appliqués à un type tableau, ces attributs donnent respectivement le premier et le dernier indice du tableau – donc ici, respectivement 1 et 3.

Il est ainsi possible de parcourir un tableau sans s'encombrer l'esprit de l'intervalle d'indices valides. Remarquez le transtypage explicite ligne 23 : il est nécessaire, car les éléments de `t2` sont de type `Integer`, tandis que ses indices sont de type `Entier`.

Cela implique que la variable de boucle `i` est elle-même de type `Entier`, donc le résultat de l'opération `3*i` serait également de type `Entier`. Le typage strict de Ada nous interdit d'affecter une valeur de type `Entier` à une variable de type `Integer` : la conversion ne peut donc être évitée. Cela tient au fait que `Entier` est un type dérivé de `Integer`, un nouveau type ; si la ligne 4 avait plutôt déclaré un sous-type, avec :

```
4    subtype Entier is Integer range 1..3 ;
```

...alors le transtypage n'aurait pas été nécessaire.

La ligne 25 fait appel à l'attribut `Range`, qui est en fait un raccourci pour `t3'First..t3'Last`. Il est appliqué à la variable, mais il peut également être appliqué au type tableau lui-même, comme cela est fait ligne 29 (les attributs `First` et `Last` sont également applicables au type). Enfin, ligne 33 on donne simplement le type de l'indice (`Rgb`), duquel sera déduit l'intervalle que devra parcourir la variable de boucle `i`. Dernier mot, l'attribut `Length` donne le nombre d'éléments contenus dans un tableau sous la forme d'un entier (par exemple, `t1'Length` ou `TTab'Length` valent tous deux `3`).

Jusque-là, mis à part les contraintes liées au typage fort, rien de véritablement extraordinaire.

Agrégats et affectations

Mais voici quelques possibilités fort intéressantes, qui rappellent un peu ce que l'on trouve en langage Python (sauf que Python a été conçu bien après Ada). Tout d'abord, il peut être pertinent d'initialiser le contenu du tableau dès la déclaration :

```
type TTab is array (1..3) of Float ;
t: TTab := (1.0, 2.0, 3.0) ;
```

Le type `TTab` est un type tableau contenant trois valeurs en virgule flottante, indicées par des entiers entre 1 et 3. La deuxième ligne déclare une variable `t` de ce type, et l'affectation permet tout simplement de donner le contenu du tableau. La liste entre parenthèses à droite de l'affectation est un agrégat. Cette notation va bien quand on n'a que quelques valeurs, mais qu'en est-il pour un tableau de grande taille ? On fait appel au mot-clef `others`, que nous avons déjà rencontré le mois dernier dans le cadre de l'instruction `case` :

```
type TTab is array (1..100_000) of Float ;
t: TTab := (others => 0.0) ;
```

Cette fois `TTab` contient 100000 valeurs (remarquez l'utilisation du caractère de soulignement dans l'écriture du nombre, qui n'a rien d'obligatoire mais permet de le rendre plus lisible). L'agrégat pourrait se lire comme « pour tous les autres indices, affecter la valeur 0.0 » : à la suite de l'affectation, toutes les valeurs du tableau prennent donc la valeur 0.0.

Les agrégats prennent tout leur intérêt lorsqu'ils sont utilisés en dehors de la déclaration du tableau. De plus, ils peuvent prendre des formes sensiblement plus sophistiquées. Par exemple, en reprenant notre grand tableau, supposons que l'on souhaite initialiser toutes les valeurs à `0.0`, sauf les valeurs d'indices `7` et `77` qui doivent prendre la valeur `1.0`, les indices `101` et `1010` qui doivent prendre la valeur `2.0`, les indices entre `2000` et `3000` qui doivent prendre la valeur `3.0`... tout cela peut s'exprimer en une seule instruction :

```
t := (7|77 => 1.0, 101|1010 => 2.0, 2000..3000 => 3.0, others => 0.0) ;
```

On retrouve en fait là une notation similaire à celle disponible dans l'instruction `case` (voir l'article du mois dernier). Encore plus fort : on peut limiter la portion du tableau affectée par l'affectation, en donnant un intervalle à gauche du signe `:=` :

```
t(123...456) := (others => 0.0) ;
```

Ceci a pour effet de mettre à `0.0` les valeurs aux indices compris entre `123` et `456` inclus, sans toucher aux autres valeurs du tableau. L'agrégat donné à droite peut naturellement prendre une forme aussi complexe que vous voulez, l'important étant qu'il représente exactement autant de valeurs que la « tranche » spécifiée à gauche.

Opérateurs pour tableaux

Tout un ensemble d'opérateurs sont définis pour les tableaux à une seule dimension. Supposons que nous ayons les déclarations suivantes :

```
type Tab2Int is array (Integer range 1..2) of Integer ;
type Tab4Int is array (Integer range 1..4) of Integer ;
type Tab2Bool is array (Integer range 1..2) of Boolean ;
t1: Tab2Int ;
t2: Tab2Int ;
tb1: Tab2Bool := (True, False) ;
tb2: Tab2Bool := (True, True) ;
tb3: Tab2Bool ;
t4: Tab4Int ;
```

Alors le tableau suivant résume les différentes opérations possibles (Tab. 2)

Pour les habitués des langages C/C++, si ce n'était pas encore clair, remarquez bien qu'un tableau en Ada n'a rien à voir avec un pointeur.

C'est un type à part entière, qui ne se contente pas de contenir des données mais transporte avec lui pas mal d'informations « administratives ».

Les opérateurs précédents, notamment l'affectation et l'égalité, en sont la meilleure preuve.

Les tableaux anonymes

Jusqu'ici nous avons toujours déclaré un type de tableaux avant d'en créer une variable.

Ce n'est pas réellement obligatoire, il est possible de déclarer ainsi une variable représentant un tableau :

```
t1: array (1..10) of Integer ;
t2: array (1..10) of Integer := (others => 2) ;
```

Les deux variables `t1` et `t2` semblent parfaitement équivalentes... pourtant elles ne sont pas de même type et c'est là l'inconvénient principal de cette méthode.

Cela implique qu'il est impossible de réaliser des affectations comme `t1 := t2`, des comparaisons, etc. Aussi l'utilisation des tableaux anonymes est-elle vraiment déconseillée.

De plus, ils ne peuvent pas apparaître comme type de paramètre passé à un sous-programme.

Tableaux presque dynamiques

Peut-être certains sont-ils un peu inquiets à la suite de la remarque concernant la différence fondamentale entre un tableau Ada et un tableau C/C++.

Comment faire pour déclarer un tableau dont la taille n'est pas connue à l'avance ou, encore pire, Ada ne possède peut-être pas de pointeurs ?

Que tout le monde se rassure, Ada possède bien des pointeurs – nous en parlerons plus tard.

Regardez maintenant le petit programme suivant, qui montre que les pointeurs ne sont pas forcément nécessaires pour déclarer des tableaux de taille inconnue à la compilation :

:: Tab. 2 :: Opérateurs pour tableaux ::

Opérateur	Description	Exemple

:=	Affectation entre tableaux ou agrégats de même type.	`t1 := (3, 4) ;`
		`t2 := t1 ;`
		`-- t2 = (3, 4)`

=, /=	Égalité, inégalité : retourne un booléen résultat d'une comparaison élément par élément. Les tableaux doivent être de même taille.	`t1 = t2 ;  -- vrai`
		`t1(1) := 0 ;`
		`t1 = t2 ;  -- faux`

not, and, or, xor	Opérations booléennes sur des tableaux de mêmes tailles contenant des booléens. Le résultat est un tableau de booléens, dont les bornes sont celles de l'opérande de gauche. Pratique pour manipuler globalement des tableaux représentant des champs de bits.	`tb3 := not tb1 ;`
		`-- tb3 = (False, True)`
		`tb3 := tb1 xor tb2 ;`
		`-- tb3 = (False, True)`

&	Concaténation de tableaux ou d'agrégats, en général pour affecter le résultat à un tableau plus grand. Notez que si vous donnez des tranches de tableaux (comme dans le deuxième exemple), les types de tableau doivent être identiques.	`t4 := (5, 6) & (7, 8) ;`
		`-- t4 = (5, 6, 7, 8)`
		`t4 := t4(3..4) & t4(1..2) ;`
		`-- t4 = (7, 8, 5, 6)`

<, <=, >, >=	Opérations de comparaison sur des tableaux contenant un type discret (entier, énuméré). L'ordre lexicographique est utilisé, c'est-à-dire celui utilisé dans un dictionnaire. Les tableaux n'ont pas forcément la même taille, mais ils doivent être de mêmes types.	`t1 < t2 ; -- vrai`
		`t4 < t4(1..2) ; -- faux`
		`t4 < t4(2..3) ; -- vrai`

```
1 with Text_IO ; use Text_IO ;
2 procedure Tableaux3 is
3    procedure Carres (n: in Integer) is
4       type TTableau is array (1..n) of Integer ;
5       tab: TTableau ;
6    begin
7       for i in tab'Range
8       loop
9          tab(i) := i*i ;
10      end loop ;
11      for i in tab'Range
12      loop
13         Put(Integer'Image(tab(i))) ;
14         if i < tab'Last
15         then
16            Put(", ") ;
17         else
18            New_Line ;
19         end if ;
20      end loop ;
21   end ;
22 ch: String(1..5) := (others => ' ') ;
23 ch_l: Integer ;
24 nb: Integer ;
25 begin
26    Put_Line("Taille : ") ;
27    Get_Line(ch, ch_l) ;
28    nb := Integer'Value(ch) ;
29    Carres(nb) ;
30 end Tableaux3 ;
```

Intéressons-nous d'abord à la partie principale, entre les lignes 22 et 30. `ch` est une variable chaîne pouvant contenir 5 caractères (si l'écriture vous laisse supposer que le type `String` est en fait un type tableau contenant des caractères, vous avez parfaitement raison).

Cette chaîne récupère l'entrée de l'utilisateur, obtenue ligne 27 par la procédure `Get_Line` fournie par `Text_IO` ; le deuxième paramètre `ch_l` est simplement la longueur de la chaîne lue (nous reviendrons dans un prochain article sur les fonctions d'entrées-sorties). Puis cette chaîne est convertie en entier stocké dans `nb` (ligne 28), grâce à l'attribut `Value` du type `Integer`.

Cet attribut permet de convertir une chaîne en la valeur correspondante pour un type scalaire, c'est-à-dire tous les types numériques et les types énumérés. Par exemple, `Rgb'Value("rOugE")` donnera la valeur `Rouge` du type énuméré `Rgb` présenté au début de cet article – remarquez qu'il n'est fait aucun cas de la casse des lettres, Ada ne faisant lui-même pas la différence entre majuscules et minuscules.

Enfin on invoque une procédure `Carres`, qui va construire un tableau des `nb` premiers carrés puis afficher le contenu de ce tableau. C'est cette procédure qui nous intéresse.

Clairement la taille du tableau à créer n'est pas connue à l'avance. Cela ne pose aucun problème à Ada : la ligne 4 déclare un type tableau `TTableau`, en donnant tout simplement le paramètre `n` reçu comme borne supérieure (le type de l'indice est implicitement entier).

En fait, les deux bornes ne sont pas nécessairement des constantes : on pourrait parfaitement avoir une définition de type comme

```
type T is array (2*n+1..n**3-2) of Integer ;
```

...ce qui peut résulter en un type vide, si `n` vaut `0` (l'opérateur `**` est l'élévation à la puissance : `n**3 = n*n*n`).

Le reste de la procédure ne devrait pas présenter de difficultés particulières. Une variable du type fraîchement créé est déclarée ligne 5, une première boucle remplit le tableau (lignes 7 à 10), une deuxième affiche son contenu (lignes 11 à 20).

La procédure `New_Line` invoquée ligne 18 vient de `Text_IO` et permet simplement de passer à la ligne (l'équivalent d'un `printf("\n")`). Voici un exemple d'utilisation :

```
$ gnatmake tableaux3.adb && ./tableaux3
Taille :
5
1, 4, 9, 16, 25
```

Cela fonctionne donc bien !

Tableaux non contraints

La souplesse d'Ada concernant les tableaux ne s'arrête pas là. Tous les types déclarés jusqu'à maintenant étaient contraints, c'est-à-dire que dès la déclaration du type, les bornes sont connues, fut-ce à l'exécution.

Mais ce n'est pas toujours satisfaisant. Imaginez le problème : écrire une procédure capable d'afficher le contenu d'un tableau d'entiers, quelles que soient les bornes de ce tableau.

Avec ce que nous savons pour l'instant, c'est tout simplement impossible. Alors pour nous sauver, voici les tableaux non contraints, dont la déclaration ressemble à ceci :

```
type Tableau is array (Integer range <>) of Integer ;
```

On donne toujours le type de l'indice et le type des éléments, mais l'intervalle de l'indice a été remplacé par le symbole <> (qui se lit « boîte », ou « *box* » en anglais). Ceci déclare un type de tableau dont les bornes ne sont pas limitées a priori (sauf par les limites du type de l'indice, bien sûr).

Pour déclarer une variable de ce type, il est nécessaire de préciser l'intervalle voulu par exemple ainsi :

```
t1: Tableau(1..10) := (others => 1) ;
-- t1 = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
t2: Tableau(-5..0) := (-5 => 2, -3 => 3, others => 1) ;
-- t2 = (2, 1, 3, 1, 1, 1)
```

Remarquez l'agrégat étrange utilisé pour initialiser le deuxième tableau. Ce qui est intéressant là-dedans, c'est que bien que `t1` et `t2` soient de tailles différentes, ils sont de même type. On peut dès lors s'amuser à certaines manipulations autrement impossibles, par exemple :

```
t2(-4..-2) := t1(7..9) ;
-- t2 = (2, 1, 1, 1, 1, 1)
t1 < t2 ; -- vrai
```

Mais revenons à notre procédure. Voici quel pourrait être son code :

```
procedure AffTab(t: in Tableau) is
begin
   for i in t'Range
   loop
      Put(Integer'Image(t(i))) ;
      if i < t'Last
      then
         Put(",") ;
      else
         New_Line ;
      end if ;
   end loop ;
end ;
```

Aucune hypothèse n'est faite sur les bornes ou la taille du tableau. Même un tableau vide (dont la borne inférieure est plus grande que la borne supérieure) convient : simplement rien ne sera affiché.

On a ainsi gagné une certaine abstraction, une certaine généricité, sans faire appel au mécanisme du même nom. Ada possède en effet dès sa conception (en 1979 !) la notion de généricité, que l'on retrouve en C++ par les classes et fonctions `template`.

Mais c'est une autre histoire que nous aborderons plus tard.

Les tableaux multidimensionnels

Il existe essentiellement deux techniques pour obtenir plusieurs dimensions : soit créer des tableaux de tableaux, soit donner plusieurs indices à un tableau.

Les deux approches présentent divers avantages et divers inconvénients chacune. Préférer l'une ou l'autre dépend de ce que l'on veut faire du tableau.

Le programme suivant déclare un tableau de tableaux :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure Tab_Multi_1 is
 3    type Tab4Int is array (1..4) of Integer ;
 4    type Tab3Tab4Int is array (1..3) of Tab4Int ;
 5    procedure Aff(t: in Tab3Tab4Int) is
 6    begin
 7       Put("(") ;
 8       for l in t'Range
 9       loop
10          if l > t'First
11          then
12             Put(" ") ;
13          end if ;
14          Put("(") ;
15          for c in t(l)'Range
16          loop
17             Put(Integer'Image(t(l)(c))) ;
18             if c < t(l)'Last
19             then
20                Put(",") ;
21             end if ;
22          end loop ;
23          Put(")") ;
24          if l < t'Last
25          then
26             Put_Line(",") ;
27          end if ;
28       end loop ;
29       Put_Line(")") ;
30    end ;
31 t1: Tab3Tab4Int := (others => (others => 0)) ;
32 t2: Tab3Tab4Int := ((11, 12, 13, 14),
33                    (21, 22, 23, 24),
34                    (31, 32, 33, 34)) ;
35 begin
36    t1(2)(2..3) := t2(3)(1..2) ;
37    Aff(t1) ;
38    -- t1 = (( 0,  0,  0,  0),
39    --       ( 0, 31, 32,  0),
40    --       ( 0,  0,  0,  0))
41    t1(1..2) := t2(1..2) ;
42    Aff(t1) ;
43    -- t1 = ((11, 12, 13, 14),
44    --       (21, 22, 23, 24),
45    --       ( 0,  0,  0,  0))
46    t1(1..2) := t1(2) & t1(1) ;
47    Aff(t1) ;
48    -- t1 = ((21, 22, 23, 24),
49    --       (11, 12, 13, 14),
50    --       ( 0,  0,  0,  0))
51 end Tab_Multi_1 ;
```

Il n'y a en fait pas grand-chose de particulier. Les bornes données ne doivent pas nécessairement être connues à la compilation. La plupart des facilités des tableaux sont disponibles, comme les agrégats ou les affectations.

Mais il y a un inconvénient notable. Les éléments d'un tableau doivent nécessairement être contraints.

Une déclaration comme celle-ci est autorisée :

```
type T1 is array (1..10) of Float ;
type T2 is array (Integer range <>) of T1 ;
```

Le type `T1` est contraint, mais pas le type `T2` : on a donc un tableau non contraint de tableaux contraints. Mais si on change la déclaration de `T1` pour un tableau non contraint, la déclaration de `T2` ne sera pas acceptée, qu'elle soit contrainte ou non :

```
type T1 is array (Integer range <>) of Float ; -- T1 non contraint
type T2 is array (1..2) of T1 ;                -- erreur !
type T2 is array (Integer range <>) of T1 ;    -- erreur !
```

Si on a besoin de la souplesse d'un type non contraint sur plusieurs dimensions, on fait appel aux tableaux multidimensionnels. Le programme suivant montre comment les utiliser, ainsi que le transtypage entre types tableaux :

```
 1 with Text_IO ; use Text_IO ;
 2 procedure Tab_Multi_2 is
 3    type Tab3x4Int is array (1..3, 1..4) of Integer ;
 4    type TabMulti is array (Integer range <>, Integerrange
<>) of Integer ;
 5    procedure Aff(t: in TabMulti) is
 6    begin
 7       Put("(") ;
 8       for l in t'Range(1)
 9       loop
10          if l > t'First(1)
11          then
12             Put(" ") ;
13          end if ;
14          Put("(") ;
15          for c in t'Range(2)
16          loop
17             Put(Integer'Image(t(l, c))) ;
18             if c < t'Last(2)
19             then
20                Put(",") ;
21             end if ;
22          end loop ;
23          Put(")") ;
24          if l < t'Last(1)
25          then
26             Put_Line(",") ;
27          end if ;
28       end loop ;
29       Put_Line(")") ;
30    end ;
31 t1: Tab3x4Int ;
32 t2: TabMulti(7..10, -5..-4) := ((11, 12),
33                                (21, 22),
34                                (31, 32),
35                                (41, 42)) ;
36 t3: TabMulti := ((11, 12, 13, 14),
37                 (21, 22, 23, 24),
38                 (31, 32, 33, 34)) ;
39 begin
40    Aff(t2) ;
41    Aff(t3) ;
42    t1 := Tab3x4Int(t3) ;
43    t1(2, 3) := 0 ;
44    t3 := TabMulti(t1) ;
45    Aff(t3) ;
46 end Tab_Multi_2 ;
```

Voyez comment les éléments du tableau sont référencés différemment que dans l'exemple précédent. Quand on écrit `t(i)(j)` pour un tableau de tableaux, on écrit `t(i,j)` pour un tableau à deux dimensions.

Par contre, les agrégats sont tout à fait semblables. Remarquez par ailleurs la déclaration de la variable `t3`, lignes 36 à 38 : elle est d'un type non contraint, pourtant on ne donne pas les bornes (comme cela a été fait pour `t2`, ligne 32).

Les bornes sont en fait déduites de l'agrégat utilisé pour l'initialisation. Utilisez les attributs pour connaître les valeurs de ces bornes.

Les attributs, justement, se voient agrémentés d'un paramètre (voyez le code de la procédure `Aff`, aux lignes 8, 10, 15, 18 et 24). Il indique quelle est la dimension interrogée.

Ainsi `t'Range(1)` donne l'intervalle pour le premier indice, `t'First(2)` donne la borne inférieure du deuxième indice, et ainsi de suite. Encore une fois, voyez la souplesse d'Ada, qui nous permet d'écrire une procédure d'affichage relativement générique sans connaître les bornes des tableaux.

Enfin, les lignes 42 et 44 réalisent deux conversions de type tableaux. Bien qu'ils soient de mêmes dimensions, on ne pourrait pas réaliser d'affectations entre `t1` et `t3` simplement avec `t1 := t3` : ils sont de types différents. Mais parce qu'ils sont de mêmes dimensions, il est possible de les convertir l'un dans l'autre.

Par contre le transtypage entre `t1` et `t2` n'est pas possible : les dimensions sont incompatibles. Bien qu'une affectation comme `t1 := Tab3x4Int(t2)` soit acceptée par le compilateur (mais seulement parce que `t2` est d'un type non contraint), à l'exécution une exception `Constraint_Error` sera levée.

Dernier mot, il n'y a pas de limite théorique au nombre de dimensions, seulement celles imposées par le matériel et le compilateur.

Conclusion

Voilà pour cette présentation des tableaux en Ada. S'ils n'ont pas toute la puissance de ceux que l'on peut trouver dans les langages interprétés comme Python ou Ruby, ils sont considérablement plus souples et pratiques à manipuler que dans les langages comme C/C++ ou Java.

Le mois prochain, nous découvrirons les enregistrements (ou structures), ce qui sera l'occasion de commencer le programme des Tours de Hanoï qui nous accompagnera durant quelques articles.

Références

[1] Ada 2005 sur Wikibooks : http://en.wikibooks.org/wiki/Programming:Ada:2005

[2] Ébauche du prochain standard : http://www.adaic.com/standards/rm-amend/html/AA-TTL.html