Le langage Ada 95 – 7 : La généricité

Yves Bailly

EN DEUX MOTS **Assez naturelle dans les langages interprétés comme Perl ou Python (quoique l'on parle plutôt de typage dynamique), la généricité est considérée comme une fonctionnalité de très haut niveau dans les langages compilés, apportant un outil extrêmement puissant pour réaliser l'abstraction des données et ainsi facilitant grandement la réutilisation du code laborieusement écrit. Voyons comment Ada se positionne.**

Le principe de la généricité, si vous ne le connaissez pas déjà, est en fait assez simple. L'idée est de pouvoir écrire du code, d'implémenter un algorithme, sans connaître précisément le type des données mises en jeu dans les traitements. C'est une manière de s'affranchir (partiellement) du typage statique habituellement rencontré.

Par exemple, un algorithme de tri reste identique, qu'il s'agisse de trier des entiers, des réels ou des pommes ; il suffit de disposer d'un moyen pour comparer deux objets. On est alors plus intéressé par une opération disponible sur le type que par le type lui-même, qui est finalement sans grand intérêt. Mais cette idée simple, comme nous allons le voir, peut donner lieu à des développements assez sophistiqués.

[1] Document STEELMAN : http://en.wikisource.org/wiki/Steelman_language_requirements

Petit historique

La généricité faisait partie des exigences du cahier des charges émis par le Département de la Défense des États-Unis d'Amérique en 1977 (document « STEELMAN » [1]), alors demandeur d'un nouveau langage de programmation pour remplacer les nombreux langages qui étaient utilisés à ce moment.

Ayant remporté le concours, cette facilité figure donc au nombre des nombreuses fonctionnalités du langage Ada dès cette époque – en fait, Ada est le premier langage compilé à proposer pleinement la généricité.

Celle-ci apparaîtra plus tard dans des langages orientés objet comme Eiffel (1985) ou C++, ce dernier popularisant la notion au moyen des *templates*. Les langages plus récents comme Java ou C# ne proposeront une forme de généricité que bien plus tard.

On peut donc dire que Ada, dans ce domaine comme dans bien d'autres, a été un précurseur.

Sous-programme générique

Considérons dans un premier temps un exemple d'un classicisme à faire peur : nous souhaitons créer une procédure permettant d'échanger la valeur de deux variables. Son code sera trivial, mais imaginez qu'il s'agit de quelque chose de plus compliqué. L'approche consistant à surdéfinir la procédure, et donc dupliquer le code, pour chaque type concerné n'est pas vraiment efficace. La généricité permet de simplifier grandement la tâche.

Commençons par un programme simple, placé dans un fichier `swap_1.adb` :

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 procedure Swap_1 is
 4    generic
 5       type Un_Type is private ;
 6    procedure Generic_Swap(v1: in out Un_Type ;
 7                           v2: in out Un_Type) ;
 8
 9    procedure Generic_Swap(v1: in out Un_Type ;
10                           v2: in out Un_Type) is
11       temp: Un_Type := v1 ;
12    begin
13       v1 := v2 ;
14       v2 := temp ;
15    end Generic_Swap ;
16
17    procedure Integer_Swap is new Generic_Swap(Integer) ;
18    procedure Float_Swap is new Generic_Swap(Un_Type => Float) ;
19
20 i1: Integer := 1 ;
21 i2: Integer := 2 ;
22 f1: Float := 11.0 ;
23 f2: Float := 22.0 ;
24
25 begin
26    Put_Line("i1 = " & Integer'Image(i1) &
27       ", i2 = " & Integer'Image(i2)) ;
28    Integer_Swap(i1, i2) ;
29    Put_Line("i1 = " & Integer'Image(i1) &
30       ", i2 = " & Integer'Image(i2)) ;
31
32    Put_Line("f1 = " & Float'Image(f1) &
33       ", f2 = " & Float'Image(f2)) ;
34    Float_Swap(f1, f2) ;
35    Put_Line("f1 = " & Float'Image(f1) &
36       ", f2 = " & Float'Image(f2)) ;
37 end Swap_1 ;
```

Notre procédure générique est déclarée lignes 4 à 7. Cette déclaration (sans code) est nécessaire pour tout sous-programme générique. Elle est introduite par le mot-clef `generic`, suivi des paramètres génériques formels. Ici un seul paramètre est utilisé : il s'agit d'un type de données (`type`),

connu dans la procédure sous le nom `Un_Type`, dont le contenu est inconnu (`is private`). Ce peut être aussi bien un type fondamental qu'un enregistrement ou un type tableau. Le préfixe `Generic_` dans le nom de la procédure n'a rien d'obligatoire, ce n'est qu'une convention de *nommage*.

Ensuite vient le code proprement dit de la procédure, lignes 9 à 15. Le type générique est utilisé naturellement, la seule hypothèse qui est faite étant qu'il est possible de réaliser des affectations entre objets de ce type.

Il est tout simplement interdit de réaliser toute autre hypothèse sur le type : par exemple, tenter d'additionner les deux paramètres v1 et v2 résulterait en une erreur de compilation, même si le type réel permet une telle opération. C'est là une différence fondamentale avec le mécanisme des templates du C++. Voici une déclaration à peu près équivalente en C++ :

```
1 template <class Un_Type>
2 void generic_swap(Un_Type& v1, Un_Type& v2)
3 {
4    Un_Type temp = v1 + v2 ;
5    v1 = v2 ;
6    v2 = temp ;
7 }
```

L'addition en ligne 4 ne poserait aucun problème, tant que le type réel donné pour `Un_Type` accepte cette opération pour chacune des instanciations rencontrées. L'erreur de compilation ne surviendra que dans le cas contraire (par exemple, en tentant d'invoquer `generic_swap()` avec des pointeurs en paramètres).

Ada est beaucoup plus restrictif : même si toutes les instanciations permettent l'addition, celle-ci sera refusée par le compilateur, car elle sous-entend une hypothèse sur le type générique `Un_Type` qui est interdite par la qualification `is private`. À noter une autre différence fondamentale avec le C++, ou du moins les compilateurs les plus répandus : en Ada, les sous-programmes génériques sont effectivement compilés à part, et non pas seulement au moment de l'instanciation. L'obligation qui existe avec la plupart des compilateurs C++ de faire apparaître dans une même unité de compilation le code générique et le code qui l'utilise n'existe pas en Ada, comme nous le verrons plus loin avec les paquetages génériques.

Voyons justement comment effectuer l'instanciation de notre procédure générique. Cela revient en fait à créer une nouvelle procédure, en indiquant le type réel souhaité : deux exemples sont donnés en lignes 17 et 18. Cette instanciation doit être explicite : contrairement au C++, il est impossible d'invoquer `Generic_Swap()` directement. La ligne 18 montre qu'il est possible de nommer le paramètre générique lorsqu'on lui affecte sa « valeur », de la même manière que les paramètres d'un sous-programme peuvent être nommés. Cela n'a rien d'obligatoire, mais clarifie grandement le code et facilite sa relecture. Un principe clef dans la conception du langage Ada était que le code est écrit une fois, mais relu de nombreuses fois.

La suite du programme ne fait qu'illustrer la mise en œuvre de la procédure générique instanciée. Les deux instanciations sont utilisées naturellement, comme s'il s'agissait de procédures sans rien de particulier. Voici la sortie de ce programme :

```
$ gnatmake swap_1 && ./swap_1
gnatgcc -c swap_1.adb
gnatbind -x swap_1.ali
gnatlink swap_1.ali
i1 =  1, i2 =  2
i1 =  2, i2 =  1
f1 =  1.10000E+01, f2 =  2.20000E+01
f1 =  2.20000E+01, f2 =  1.10000E+01
```

L'instanciation explicite peut paraître lourde, mais encore une fois elle permet de clarifier le code. Elle est de plus nécessaire, étant donné qu'en Ada il n'existe tout simplement pas de conversions implicites d'un type vers un autre. Par ailleurs, nous verrons plus loin comment limiter ou renforcer les restrictions imposées aux paramètres génériques.

Poursuivons. Notre procédure fonctionne. Elle est manifestement d'une utilité très générale : plutôt que de la placer ainsi directement dans un programme, il serait préférable de l'empaqueter dans un paquetage. Il suffit de placer la déclaration dans une spécification et l'implémentation dans un corps. Voici ce que deviendrait notre programme si la procédure était placée dans un paquetage `Swap` (dans les fichiers `swap.ads` et `swap.adb`) :

Paquetage générique

Un sous-programme générique, c'est bien. Un ensemble de sous-programmes regroupés dans un paquetage générique, c'est mieux. Imaginons par exemple une série de sous-programmes qui manipuleraient des tableaux, comme rechercher un élément, trier les éléments... Quels seraient les paramètres (types) génériques ?

- Clairement, le type d'un élément de tableau ;
- Très probablement, le type tableau lui-même, de préférence non contraint (bornes indéfinies) ;
- Moins évident peut-être, mais nécessaire, le type de l'indice du tableau : rappelez-vous qu'en Ada, les indices d'un tableau peuvent être de n'importe quel type discret (types entiers ou énumérés), ces indices ne commençant pas nécessairement à zéro...

Cela nous fait trois paramètres. Il serait fastidieux de les répéter pour chaque sous-programme : la création d'un paquetage générique s'impose d'elle-même. Notre paquetage va fournir quatre sous-programmes :

Procédure générique dans un paquetage

swap.ads

```
1 package Swap is
2    generic
3       type Un_Type is private ;
4    procedure Generic_Swap(v1: in out Un_Type ;
5                           v2: in out Un_Type) ;
6 end Swap ;$
```

swap.adb

```
1 package body Swap is
2    procedure Generic_Swap(v1: in out Un_Type ;
3                           v2: in out Un_Type) is
4       temp: Un_Type := v1 ;
5    begin
6       v1 := v2 ;
7       v2 := temp ;
8    end Generic_Swap ;
9 end Swap ;
```

swap_2.adb

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 with Swap ;
 4 procedure Swap_2 is
 5
 6    type Tableau is array(1..10) of Float ;
 7
 8    procedure Integer_Swap is new Swap.Generic_Swap(Integer) ;
 9    procedure Tableau_Swap is new Swap.Generic_Swap(Un_Type => Tableau) ;
10
11 i1: Integer := 1 ;
12 i2: Integer := 2 ;
13 t1: Tableau := (others => 11.0) ;
14 t2: Tableau := (others => 22.0) ;
15
16 begin
17
18    Put_Line("i1 = " & Integer'Image(i1) &
19       ", i2 = " & Integer'Image(i2)) ;
20    Integer_Swap(i1, i2) ;
21    Put_Line("i1 = " & Integer'Image(i1) &
22       ", i2 = " & Integer'Image(i2)) ;
23
24    Put_Line("t1(3) = " & Float'Image(t1(3)) &
25       ", t2(3) = " & Float'Image(t2(3))) ;
26    Tableau_Swap(t1, t2) ;
27    Put_Line("t1(3) = " & Float'Image(t1(3)) &
28       ", t2(3) = " & Float'Image(t2(3))) ;
29
30 end Swap_2 ;
```

Juste pour l'exemple, la deuxième instanciation permet d'échanger les valeurs contenues dans deux tableaux dont le type est déclaré ligne 6. Le paquetage Swap est intégré ligne 3, le nom de la procédure générique étant simplement préfixé (lignes 8 et 9). L'écriture est finalement assez naturelle : déclarez le sous-programme générique dans une spécification d'un paquetage, implémentez-le dans le corps du paquetage, puis instanciez et utilisez ce sous-programme dans un programme principal ou un autre paquetage.

- Une procédure pour échanger deux valeurs à deux indices différents d'un tableau ;
- Une procédure pour trier les éléments du tableau, indépendante de la fonction de comparaison ;
- Une fonction pour vérifier si un tableau est trié ou non, également indépendante de la fonction de comparaison ;
- Enfin, une procédure pour afficher le contenu d'un tableau.

Définition

Voici quelle pourrait être une spécification d'un tel paquetage :

```
 1 generic
 2    type Type_Elements is private ;
 3    type Type_Indice is (<>) ;
 4    type Type_Tableau is
 5       array (Type_Indice range <>) of Type_Elements ;
 6 package Generic_Tableaux is
 7
 8    procedure Echanger(tab  : in out Type_Tableau ;
 9                       ind_1: in     Type_Indice ;
10                       ind_2: in     Type_Indice) ;
11
12    generic
13       with function "<" (e1: in Type_Elements ;
14                          e2: in Type_Elements)
15          return Boolean ;
16    procedure Generic_Trier(tab: in out Type_Tableau) ;
```

```
17
18    generic
19       with function "<" (e1: in Type_Elements ;
20                          e2: in Type_Elements)
21          return Boolean ;
22    function Generic_Est_Trie(tab: in Type_Tableau)
23       return Boolean ;
24
25    generic
26       with function Element_To_String(elem: in Type_Elements)
27          return String ;
28    procedure Generic_Put_Line(tab: in Type_Tableau) ;
29
30 end Generic_Tableaux ;
```

Quelques commentaires. D'abord sur les paramètres génériques. Le premier d'entre eux, `Type_Elements`, représente le type des éléments du tableau : on retrouve la même notation (`is private`) que dans la section précédente, qui nous indique qu'aucune hypothèse n'est faite sur le type, en dehors de l'affectation standard prédéfinie.

Le deuxième, `Type_Indice`, représente le type utilisé pour les indices du type tableau que nous allons vouloir manipuler. En lieu et place de `private`, on trouve le symbole (`<>`) (que l'on peut lire « boîte », ou « *box* » an anglais). Celui-ci indique que le type en question doit être un type « discret », c'est-à-dire un type entier ou un type énuméré. On restreint dès lors fortement les types pouvant être employés pour ce paramètre, ce qui améliore l'intégrité sémantique du programme. En contrepartie, il est possible de faire certaines hypothèses sur les objets de type `Type_Indice`, en particulier qu'ils sont ordonnés, qu'il existe un premier et un dernier, qu'ils possèdent tous un prédécesseur et un successeur, qu'il est possible de les afficher (d'obtenir une chaîne de caractères représentant la valeur de chacun d'eux).

Enfin, le paramètre `Type_Tableau` décrit précisément le type tableau attendu : il s'agit d'un tableau non contraint (`range <>`) dont les éléments sont de type `Type_Elements`, indexés par des valeurs de type `Type_Indice`. Au sein du corps du paquetage, nous pourrons donc utiliser toutes les facilités offertes par Ada pour manipuler les tableaux, ce dont nous n'allons pas nous priver.

Jetons un œil aux sous-programmes proposés. Le premier, `Echanger()`, ne présente rien de particulier : on y retrouve simplement les types d'indice et de tableaux pour réaliser l'échange de deux valeurs. Le deuxième, `Generic_Trier()`, est plus intéressant : c'est une procédure de tri, elle-même générique – un peu comme une méthode template au sein d'une classe `template` en C++. Mais le paramètre générique n'est pas un type : c'est un sous-programme, en l'occurrence une fonction, qui sera utilisée pour comparer les éléments entre eux.

Ce paramètre générique est indispensable : tel qu'a été déclaré le paramètre générique correspondant au type des éléments du tableau, encore une fois, on ne peut faire aucune hypothèse quant aux fonctionnalités du type (en-dehors de l'affectation prédéfinie). En particulier, il n'existe pas de relation d'ordre implicite. C'est pourquoi il est nécessaire d'en « passer » une à la procédure de tri. Le choix a été fait de nommer cette fonction de comparaison `"<"`, afin de pouvoir comparer deux objets `a` et `b` de type `Type_Elements` à l'aide de la simple écriture `a < b`. On aurait tout aussi bien pu l'appeler (par exemple) `Comparer()`, mais le code de la procédure de tri s'en serait trouvé inutilement alourdi. Une solution alternative eût été d'ajouter un paramètre à la procédure de tri, qui aurait été un pointeur sur une fonction de comparaison – mais nous n'avons pas encore étudié les pointeurs (ou ce qui en tient lieu) en Ada...

La procédure suivante, `Generic_Est_Trie()`, a pour vocation de vérifier si le contenu d'un tableau est trié ou non. À nouveau, nous devons passer une fonction de comparaison. Enfin, la dernière procédure, `Generic_Put_Line()`, affiche le contenu d'un tableau. Cette fois le paramètre générique attendu est une fonction effectuant la conversion d'un objet de type `Type_Elements` en une chaîne de caractères.

Le corps du paquetage ne présente pas grand-chose de particulier. Examinons tout de même le code de la procédure de tri (basée sur l'algorithme récursif QuickSort) où apparaît une ou deux petites subtilités :

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 package body Generic_Tableaux is
   .....
17
18    procedure Generic_Trier(tab: in out Type_Tableau) is
19       i_inf: Type_Indice := tab'First ;
20       i_sup: Type_Indice := tab'Last ;
21       i_mid: Type_Indice :=
22          Type_Indice'Val(
23             (Type_Indice'Pos(i_inf)+Type_Indice'Pos(i_sup))/2) ;
24       pivot: Type_Elements ;
25    begin
26       if tab'Length < 2
27       then
28          return ;
29       elsif tab'Length = 2
30       then
31          if tab(i_sup) < tab(i_inf)
32          then
33             Echanger(tab, i_sup, i_inf) ;
34          end if ;
35       else
36          if tab(i_mid) < tab(i_inf)
37          then
38             Echanger(tab, i_mid, i_inf) ;
39          end if;
40          if tab(i_sup) < tab(i_mid)
41          then
42             Echanger(tab, i_sup, i_mid) ;
43             if tab(i_mid) < tab(i_inf)
44             then
45                Echanger(tab, i_mid, i_inf) ;
46             end if;
```

```
47          end if;
48          if tab'Length > 3
49          then
50             -- cas général
51             i_inf := Type_Indice'Succ(i_inf) ;
52             i_sup := Type_Indice'Pred(i_sup) ;
53             pivot := tab(i_mid) ;
54             DECOUPAGE:
55             loop
56                ELEM_INF:
57                loop
58                   exit DECOUPAGE when i_sup <= i_inf ;
59                   exit ELEM_INF when pivot < tab(i_inf) ;
60                   i_inf := Type_Indice'Succ(i_inf) ;
61                end loop ELEM_INF ;
62                ELEM_SUP:
63                loop
64                   exit DECOUPAGE when i_sup <= i_inf ;
65                   exit ELEM_SUP when tab(i_sup) < pivot ;
66                   i_sup := Type_Indice'Pred(i_sup) ;
67                end loop ELEM_SUP ;
68                Echanger(tab, i_inf, i_sup) ;
69                i_inf := Type_Indice'Succ(i_inf) ;
70                i_sup := Type_Indice'Pred(i_sup) ;
71             end loop DECOUPAGE ;
72             Generic_Trier(tab(tab'First..i_inf)) ;
73             Generic_Trier(tab(i_sup..tab'Last)) ;
74          end if ;
75       end if ;
76    end Generic_Trier ;
77
   ......
110 end Generic_Tableaux ;
```

Pour mémoire, l'algorithme récursif QuickSort repose sur le découpage en deux du tableau de départ en une partie inférieure et une partie supérieure, contenant les éléments respectivement plus petits et plus grands qu'une valeur pivot, chacune des deux parties étant triée à son tour. Ici la valeur pivot est extraite du tableau lui-même, en l'occurrence la valeur de l'élément situé au milieu (c'est parfaitement arbitraire et probablement pas optimal). Nous avons donc besoin au minimum de trois indices : un pour parcourir le tableau du début vers la fin (`i_inf`, ligne 19), un pour le parcourir en sens inverse depuis la fin (`i_sup`, ligne 20) et enfin un troisième correspondant à l'indice médian – le milieu du tableau, `i_mid` en ligne 21.

Il va donc nous falloir manipuler des indices, ce qui n'est pas une surprise. Ceux-ci sont représentés par le type générique `Type_Indice`, déclaré comme étant d'un type discret... sans plus de contrainte. Conséquence : si i est de type `Type_Indice`, l'écriture `i+1` n'est pas forcément légitime – donc elle ne l'est pas du tout. En fait, toute opération arithmétique est interdite. Comment faire alors ?

En faisant appel aux attributs. Nous avons vu dans le quatrième article de cette série, consacré aux tableaux, les attributs qui pouvaient être appliqués aux types discrets. Notre paramètre type générique `Type_Indice` étant un type discret (par déclaration), on peut parfaitement lui appliquer ceux-ci. Ainsi, pour incrémenter notre indice inférieur, plutôt que d'écrire `i_inf := i_inf + 1` nous écrirons `i_inf := Type_Indice'Succ(i_inf)`, l'attribut `'Succ` donnant le successeur d'une valeur d'un type discret (ligne 51, par exemple). C'est là la première subtilité dans le code de cette procédure.

La deuxième réside dans la récursivité, mise en œuvre lignes 72 et 73. Si vous avez déjà codé l'algorithme QuickSort, peut-être avez-vous été étonné de ne pas voir en paramètre à la procédure les bornes du sous-tableau à trier. C'est en fait inutile en Ada : les types tableau sont des types « riches », ils contiennent l'information concernant les bornes inférieures et supérieures des indices. Mais regardez tout de même ces lignes 72 et 73 : on invoque la procédure récursivement, en lui passant une tranche du tableau initial.

On pourrait s'attendre à ce que la procédure reçoive une copie de cette tranche, et donc n'agisse pas sur le tableau lui-même. Il n'en est rien : les tranches ainsi données sont en fait des références au tableau de départ (enfin, à une partie). Cela ne fonctionne que parce que le mode de passage du paramètre tableau est `in out`, qui équivaut à un passage par référence (avec `&`) en C++.

Instanciation

Maintenant que nous avons notre paquetage générique, voyons comment l'utiliser dans un programme de test, que voici :

```
 1 with Text_IO ;
 2 use  Text_IO ;
 3 with Generic_Tableaux ;
 4 procedure Test_Tableaux is
 5
 6    type Tab_Integer is array (Positive range <>) of Integer ;
 7
 8    package Integer_Tableaux is
 9       new Generic_Tableaux(Type_Elements => Integer,
10                            Type_Indice   => Positive,
11                            Type_Tableau  => Tab_Integer) ;
12
13    procedure Trier is
14       new Integer_Tableaux.Generic_Trier("<" => "<") ;
15
16    function Est_Trie_1 is
17       new Integer_Tableaux.Generic_Est_Trie("<" => "<=") ;
18    function Est_Trie_2 is
19       new Integer_Tableaux.Generic_Est_Trie("<" => "<") ;
20
21    procedure Put_Line is
22       new Integer_Tableaux.Generic_Put_Line(Integer'Image) ;
23
24 tab_1: Tab_Integer(1..10) := (4, 8, 5, 2, 0, 5, 9, 6, 1, 7) ;
25
26    type Jours is (Lundi, Mardi, Mercredi,
27                   Jeudi, Vendredi, Samedi,
28                   Dimanche) ;
29    type Mois is (Janvier,   Février, Mars,     Avril,
30                  Mai,       Juin,    Juillet,  Août,
31                  Septembre, Octobre, Novembre, Décembre) ;
32    type Tab_Repos is array (Mois range <>) of Jours ;
```

```
33
34    package Jours_Tableaux is
35       new Generic_Tableaux(Type_Elements => Jours,
36                            Type_Indice   => Mois,
37                            Type_Tableau  => Tab_Repos) ;
38
39    procedure Trier is
40       new Jours_Tableaux.Generic_Trier("<" => "<") ;
41
42    function Est_Trie_1 is
43       new Jours_Tableaux.Generic_Est_Trie("<" => "<=") ;
44    function Est_Trie_2 is
45       new Jours_Tableaux.Generic_Est_Trie("<" => "<") ;
46
47    procedure Put_Line is
48       new Jours_Tableaux.Generic_Put_Line(Jours'Image) ;
49
50 tab_2: Tab_Repos(Mois) := (Lundi, Vendredi, Mercredi, Mardi,
51                            Jeudi, Lundi, Dimanche, Samedi,
52                            Mercredi, Jeudi, Mardi, Samedi) ;
53
54 begin
55
56    Put("tab_1 = ") ; Put_Line(tab_1) ;
57    Put_Line("Est trié (1) ? -> " & Boolean'Image(Est_Trie_1(tab_1))) ;
58    Put_Line("Est trié (2) ? -> " & Boolean'Image(Est_Trie_2(tab_1))) ;
59    Trier(tab_1) ;
60    Put("tab_1 = ") ; Put_Line(tab_1) ;
61    Put_Line("Est trié (1) ? -> " & Boolean'Image(Est_Trie_1(tab_1))) ;
62    Put_Line("Est trié (2) ? -> " & Boolean'Image(Est_Trie_2(tab_1))) ;
63    Put_Line("------------") ;
64    Put("tab_2 = ") ; Put_Line(tab_2) ;
65    Put_Line("Est trié (1) ? -> " & Boolean'Image(Est_Trie_1(tab_2))) ;
66    Put_Line("Est trié (2) ? -> " & Boolean'Image(Est_Trie_2(tab_2))) ;
67    Trier(tab_2) ;
68    Put("tab_2 = ") ; Put_Line(tab_2) ;
69    Put_Line("Est trié (1) ? -> " & Boolean'Image(Est_Trie_1(tab_2))) ;
70    Put_Line("Est trié (2) ? -> " & Boolean'Image(Est_Trie_2(tab_2))) ;
71
72 end Test_Tableaux ;
```

Il est naturellement nécessaire d'indiquer que nous allons utiliser le paquetage (ligne 3). Le programme définit ensuite un premier type tableau, des entiers indexés par des entiers positifs (ligne 6). L'instanciation du paquetage générique se fait de façon similaire à l'instanciation d'un sous-programme générique, sauf que plutôt que de créer un nouveau sous-programme, on crée un nouveau paquetage, lignes 8 à 11. Le nommage des paramètres est utilisé pour clarifier le code. Cette instruction revient véritablement à créer un nouveau paquetage, exactement comme si on avait intégré une spécification et un corps dans notre programme (ce qui est par ailleurs parfaitement autorisé) pour un paquetage qui serait ici nommé `Integer_Tableaux`.

Nous n'en avons pas terminé avec les instanciations. Notre nouveau paquetage contient trois sous-programmes eux-mêmes génériques : il nous faut les instancier à leur tour pour pouvoir les utiliser. On commence par la procédure de tri, lignes 13 et 14. Le paramètre générique attendu doit être une fonction de comparaison : on lui donne la fonction prédéfinie pour le type des éléments des tableaux. Comme ce sont des entiers, cette fonction prédéfinie a pour nom `"<"`... ce qui donne l'écriture un peu étrange `"<" => "<"`. Le premier `"<"` est le nom donné au paramètre générique formel, le deuxième est le nom de la fonction donnée. N'ayez crainte, le compilateur s'y retrouve fort bien et avec un peu de pratique l'écriture devient naturelle.

Ensuite, on instancie à deux reprises la procédure de vérification du tri, une première fois avec une comparaison stricte, une deuxième avec une comparaison large. La première renverra toujours « faux » (`False`) pour un tableau contenant deux fois la même valeur, qu'il ait été trié ou non. Enfin vient le tour de la procédure d'affichage : on donne comme fonction de conversion en chaîne de caractères tout simplement l'attribut `'Image` du type des éléments, qui existe puisque ce sont des entiers. Cet attribut n'est pas utilisable dans le corps du paquetage générique : le paramètre `Type_Elements` étant déclaré `is private`, l'attribut n'est pas visible à ce niveau. Il est par contre visible au moment de l'instanciation.

Continuons. Après cette première instanciation « évidente », voyons une instanciation utilisant non pas des entiers, mais des types énumérés que nous aurons créés nous-mêmes. Ces types sont déclarés lignes 26 à 31, le type tableau étant défini ligne 32. L'instanciation du paquetage générique puis des sous-programmes génériques qu'il contient se fait exactement de la même manière que précédemment, comme si les types utilisés étaient des types entiers.

Le corps du programme lui-même se contente de faire quelques invocations. Remarquez qu'il est possible de donner des noms identiques à deux instanciations différentes d'une même procédure générique : cela n'est autorisé, naturellement, que si les types des paramètres de cette procédure permettent sans ambiguïté de résoudre la surcharge.

Voici l'affichage du programme :

```
$ gnatmake -gnatif test_tableaux && ./test_tableaux
gnatgcc -c -gnatif test_tableaux.adb
gnatgcc -c -gnatif generic_tableaux.adb
gnatbind -x test_tableaux.ali
gnatlink test_tableaux.ali
tab_1 = [ 1 =>  4,  2 =>  8,  3 =>  5,  4 =>  2,  5 =>  0,  6 =>  5,
7 =>  9,  8 =>  6,  9 =>  1,  10 =>  7]
Est trié (1) ? -> FALSE
Est trié (2) ? -> FALSE
tab_1 = [ 1 =>  0,  2 =>  1,  3 =>  2,  4 =>  4,  5 =>  5,  6 =>  5,
7 =>  6,  8 =>  7,  9 =>  8,  10 =>  9]
Est trié (1) ? -> TRUE
Est trié (2) ? -> FALSE
------------
tab_2 = [JANVIER => LUNDI, FéVRIER => VENDREDI, MARS => MERCREDI,
AVRIL => MARDI, MAI => JEUDI, JUIN => LUNDI, JUILLET => DIMANCHE,
AOûT => SAMEDI, SEPTEMBRE => MERCREDI, OCTOBRE => JEUDI, NOVEMBRE
```

```
=> MARDI, DéCEMBRE => SAMEDI]
Est trié (1) ? -> FALSE
Est trié (2) ? -> FALSE
tab_2 = [JANVIER => LUNDI, FéVRIER => LUNDI, MARS => MARDI, AVRIL =>
MARDI, MAI => MERCREDI, JUIN => MERCREDI, JUILLET => JEUDI, AOûT =>
JEUDI, SEPTEMBRE => VENDREDI, OCTOBRE => SAMEDI, NOVEMBRE => SAMEDI,
DéCEMBRE => DIMANCHE]
Est trié (1) ? -> TRUE
Est trié (2) ? -> FALSE
```

Il semble bien que notre paquetage générique fonctionne.

Aparté : question de performances

Le langage Ada possède une réputation de lenteur, c'est-à-dire que si on considère la vitesse d'exécution de deux codes comparables, l'un en C++, l'autre en Ada, le premier sera toujours beaucoup plus rapide que le second.

Vérifions cela avec notre petit paquetage. Deux programmes, l'un en Ada et l'autre en C++, vont chacun créer un tableau de dix millions d'entiers aléatoires, puis trier ce tableau. Le tri sera effectué en Ada par la procédure `Generic_Trier()` présentée plus haut et par la fonction générique standard `std::sort()` en C++. Pour les tests, deux versions des compilateurs ont été utilisées pour chacun des langages :

[2] Compilateur GNAT GPL 2005 : http://libre.adacore.com/

- Pour le programme en Ada, le compilateur GNAT a été utilisé :
  - la version 3.15p, généralement celle fournie avec les distributions Linux, qui s'appuie sur GCC 2.8.1 ;
  - la version GPL 2005 apparue en août dernier, qui s'appuie sur GCC 3.4.3 [2].
- Pour le programme C++, c'est directement g++ qui a été utilisé :
  - d'abord la version 3.3.5 ;
  - puis la version 3.4.3.

Ce ne sont pas les dernières versions disponibles, mais les versions généralement considérées comme étant les plus stables disponibles. Par ailleurs, les temps d'exécution du programme Ada ont été mesurés avec ou sans le paramètre `-gnatp` passé au compilateur, qui a pour effet de désactiver la plupart des tests et contrôles effectués durant l'exécution, comme ceux sur les indices de tableau pour détecter les débordements.

Chaque test est effectué à trois reprises, la valeur retenue étant la moyenne. Voici les résultats, les temps sont exprimés en secondes :

Options	Vitesses de tri en Ada et C++				
	Ada (GNAT)			C++ (g++)	
	options	*3.15p*	*GPL 2005*	*3.3.5*	*3.4.3*
Aucune	aucune	8.90	6.97	2.96	2.96
	`-gnatp`	4.70	5.13		
`-02`	aucune	4.70	3.61	1.82	1.78
	`-gnatp`	3.64	3.11		
`-03`	aucune	4.73	3.32	1.79	1.78
	`-gnatp`	3.36	2.67		

Les deux valeurs les plus comparables sont celles donnant les temps pour GNAT GPL 2005 avec `-gnatp` d'une part, et pour g++ 3.4.3 d'autre part.

D'abord les deux s'appuient sur la même version de GCC, ensuite les niveaux de « solidité » des deux codes sont comparables (le C++ n'effectue aucune vérification en cas de sortie de tableau).

On constate un avantage pour C++ d'un peu moins d'une seconde, soit un gain d'environ 33% par rapport à Ada. Cela peut paraître important, mais cette valeur est à relativiser :

- Même avec `-gnatp`, Ada (compilé par GNAT) effectue toujours quelques vérifications d'intégrité durant l'exécution.
- L'algorithme de tri est un pur QuickSort implémenté par mes soins : il est donc loin d'être optimal, alors que celui fourni avec la librairie C++ standard qui accompagne g++ est fortement optimisé.

L'un dans l'autre et toutes choses étant égales par ailleurs, on peut estimer que dans ce petit comparatif Ada s'en sort fort bien.

La réputation de lenteur d'un programme Ada est donc sans doute à reconsidérer (du moins dans ce contexte très limité) et en tout cas à mettre en balance avec la robustesse qu'apporte le langage.

Si on considère une optimisation « normale », avec `-02` mais sans `-gnatp`, l'écart de performance est plus important mais il faut garder à l'esprit qu'un indice qui déborde du tableau provoquera probablement un plantage du programme en C++, tandis que l'erreur pourra être interceptée par une exception et dûment traitée en Ada.

Natures des paramètres génériques

Les différents exemples précédents ont montré que les paramètres d'un paquetage ou d'un sous-programme générique pouvaient être de différentes natures.

Il est en effet possible de préciser assez finement ce que l'on attend d'un paramètre générique. Différentes notations, dont voici quelques-unes, sont disponibles :

Quelques modèles de paramètres génériques	
Notation	**Description**
`Nom: Type`	Le paramètre générique est en fait une valeur d'un type quelconque, tout comme on indiquerait un paramètre à un sous-programme. Au sein du corps du paquetage ou sous-programme générique, `Nom` joue le rôle d'une constante de type `Type`.
`type T is private` `type T is limited private`	La première forme que nous avons vue. Elle indique simplement que `T` désigne un type, sur lequel on ne dispose de rien de plus que l'affectation standard et l'égalité standard. Si on ajoute le mot-clef `limited`, alors même l'affectation et l'égalité ne font plus partie des opérations disponibles pour le type `T`. Le type réel passé en paramètre doit être un type défini.
`type T(<>) is private`	Le type `T` est cette fois indéfini, comme un type tableau non contraint (par exemple, le type chaîne de caractères `String`) ou un type enregistrement possédant un discriminant.
`type T is (<>)`	Le type `T` est un type discret, c'est-à-dire soit un type énuméré, soit un type entier. Mais même si le type réel passé en paramètre est un type entier, les opérations arithmétiques usuelles ne sont pas disponibles dans le corps du paquetage ou du sous-programme générique.
`type T is range <>`	Le type `T` est un type numérique entier signé. Les opérations arithmétiques usuelles sont disponibles.
`type T is digits <>`	Le type `T` est un type numérique en virgule flottante. Les opérations arithmétiques usuelles sont disponibles.
`with function F ... return ...` `with procedure P ...`	Le paramètre générique n'est cette fois pas un type, mais plutôt un sous-programme. Celui-ci doit être déclaré normalement, avec ses paramètres et leurs types, et sa valeur de retour dans le cas d'une fonction.
`with package P is new Q(...)`	Peut-être la forme la plus difficile à appréhender de paramètre générique : ce n'est ni une valeur, ni un type, ni un sous-programme, mais un paquetage complet. L'écriture indique que le paramètre qui sera connu sous le nom de `P` doit être un paquetage construit à partir d'un paquetage `Q`, lui-même étant un paquetage générique.
Ce tableau n'est qu'un résumé, il existe d'autres formes et subtilités. Mais cela devrait déjà vous permettre de faire pas mal de choses.	

Conclusion

Voilà pour ce survol de la généricité en Ada. Si vous connaissez le C++, vous pouvez constater que la généricité ainsi mise en œuvre est à la fois plus robuste (car plus restrictive) et plus souple que le mécanisme des templates. Tout est fait pour éviter les erreurs et les incohérences dans la programmation. Il est remarquable qu'Ada possède une telle fonctionnalité si avancée dès sa conception.

La prochaine fois, nous reviendrons plus en détail sur les chaînes de caractères et les possibilités d'entrées/sorties offertes par Ada.

Yves Bailly,