Chapitre 7

Introduction à la généricité en ADA

En ADA, on utilise la *généricité* quand on veut paramétrer une procédure, une fonction par des constructions qui ne sont pas des valeurs du langage (types, exceptions, fonctions, procédures et paquetages). Ça s'utilise aussi pour paramétrer un paquetage tout entier. On a déjà vu ces idées mais sans les mettre vraiment en œuvre, voir notamment les procédures de tri (section 6.2) et les paquetages de piles et files bornées (section **??**). La généricité permet de réutiliser ces procédures ou paquetages dans différents contextes.

Ici, on ne détaille pas tous les aspects de la généricité ADA (voir [Bar00] ou [ISO95] pour plus de détails). On va plutôt revisiter les exemples mentionnés plus haut en montrant comment les rendre effectivement génériques (et en illustrant l'utilisation).

7.1 Exemple d'une procédure générique

Pour créer une procédure générique, on doit déclarer les paramètres génériques sur la *spécification* de la procédure. Dans l'implémentation de la procédure, on peut utiliser les paramètres génériques : ils sont implicitement dans le contexte. On va illustrer ça sur l'exemple du tri par minimums conservatif instable (voir section 6.2).

Dans cet exemple, on met la spécification de la procédure et son implémentation dans deux fichiers séparés (un `.ads` et un `.adb`). Ceci permet de réutiliser cette procédure générique dans plusieurs fichiers. Mais on peut aussi faire des procédures génériques "locales" : il suffit de mettre la spécification et la déclaration de la procédure dans la même zone de déclaration, en commençant par la spécification.

7.1.1 Spécification et signification des paramètres génériques

On donne la spécification ci-dessous. La liste des paramètres est donnée sous forme d'une suite de déclarations qui commencent au mot-clef **generic** et se termine ici à la première occurrence de **procedure** (ou **function**) non immédiatement précédé d'un **with**. Ce mot-clef **with** sert justement à indiquer que la déclaration de procédure, de fonction ou de paquetage qui suit immédiatement est un paramètre (et pas l'unité générique).

```
generic
   type T_Element is private ;
   with function "<=" (A, B : T_Element) return Boolean ;
   type T_Tab is array (Integer range <>) of T_Element ;
procedure TriMin (T : in out T_Tab) ;
```

Ici, il y a donc 3 paramètres :

1. Le type `T_Element` qui du point de vue de l'implémentation représente un type dont on ne connaît pas la représentation.
2. Une fonction nommée ici `"<="`.
3. Le type `T_Tab` qui représente un nom de type tableau ayant la même définition. On verra ci-dessous comment se passe l'instanciation.

7.1.2 Implémentation et utilisation des paramètres génériques

L'implémentation de cette procédure correspond exactement à ce qu'on a déjà écrit dans le cours section 6.2 :

```
procedure TriMin (T : in out T_Tab) is

   function "<" (A, B : T_Element) return Boolean is
   begin
     return A<=B and not (B<=A) ;
   end ;

   Min: T_Element ;
   Ind_Min : Integer range T'Range ;
begin
   for K in T'First..T'Last - 1 loop
         -- INVARIANT: le tableau T(T'First..K-1) est trié
         --   si K>T'First alors pour tout I de K..T'Last,
         --           T(K-1) <= T(I)
         --   T est une permutation de T0.

         -- algo classique de recherche de l'indice du Min
         -- dans T(K..T'Last).
         Min:=T(K) ; Ind_Min:= K ;
         for I in K+1..T'Last loop
            -- INVARIANT: Min  minimum de T(K..I-1)
            --         T(Ind_Min)=Min  avec Ind_Min ds K..I-1
            if T(I) < Min then
               Min:=T(I) ;
               Ind_Min:= I ;
            end if ;
         end loop ;

         T(Ind_Min):=T(K) ;
         T(K):=Min ;
      end loop ;
end ;
```

7.1.3 Utilisation et instanciation des paramètres génériques

A l'utilisation, si on veut instancier `T_Element` par un type `Mes_Elements`, la fonction `"<="` par une fonction `Compare` et le type `T_Tab` par `Mes_Tab`, on doit avoir :

```
function Compare (A,B:Mes_Elements) return Boolean ;
type Mes_Tab is array (Integer range <>) of Mes_Elements ;
```

L'instanciation doit se faire dans une *zone de déclaration*, sous la forme :

```
procedure Mon_Tri is new TriMin(Mes_Elements,Compare,Mes_Tab) ;
```

Cela crée une nouvelle procédure nommée `Mon_Tri` avec l'interface suivante :

```
procedure Mon_Tri (T : in out Mes_Tab) ;
```

Attention, comme l'instanciation doit se faire dans une zone de déclaration, il n'est pas possible de faire en sorte que le nom `Mon_Tri` soit associée dans la suite du programme à une procédure qui instancie `TriMin` avec des paramètres choisis à l'exécution, du genre :

```
if ... then
   procedure Mon_Tri is new TriMin(Mes_Elements1,Compare1,Mes_Tab1) ;
else
   procedure Mon_Tri is new TriMin(Mes_Elements2,Compare2,Mes_Tab2) ;
end if ;
```

Mais, en utilisant la possibilité d'imbriquer des blocs, on a quand même une grande souplesse. Dans l'exemple ci-dessous, on trie des couples (entiers,caractères) en demandant à l'utilisateur s'il veut trier sur les entiers ou les caractères :

```
with Ada.Text_IO, Ada.Integer_Text_Io, TriMin ;
use Ada.Text_IO, Ada.Integer_Text_Io ;

procedure TestTriMin is

   type Mes_Elements is record
      Entier : Integer range 0..9;
      Lettre : Character  range 'A'..'Z' ;
   end record ;

   type Mes_Tab is array (Integer range <>) of Mes_Elements ;

   procedure Put(T:in Mes_Tab) is
   begin
      for I in T'Range loop
         Put('(');
         Put(Item=>T(I).Entier,Width=>1) ;
         Put(',');
         Put(T(I).Lettre) ;
         Put(')');
      end loop ;
      New_Line;
   end ;

   Ex: Mes_Tab :=
     ( 1 => (5,'A') ,
```

```
        2 => (2,'A') ,
        3 => (1,'A') ,
        4 => (5,'B') ,
        5 => (5,'C') ,
        6 => (2,'B') ,
        7 => (1,'B')) ;

   X: Character ;
begin
   loop
      Put("Tableau de départ:") ; Put(Ex) ;
      Put_Line("a) Tri sur les entiers") ;
      Put_Line("b) Tri sur les caractères") ;
      Put("Votre choix (a|b):");
      Get_Immediate(X) ;
      Put(X) ; New_Line ;
      case X is
         when 'a' =>
            declare
               function Compare (A,B:Mes_Elements)
                                 return Boolean is
               begin
                  return A.Entier <= B.Entier ;
               end ;

               procedure Mon_Tri is
                  new TriMin(Mes_Elements,Compare,Mes_Tab) ;
            begin
               Mon_Tri(Ex) ;
               exit ;
            end ;
         when 'b' =>
            declare
               function Compare (A, B : Mes_Elements)
                                 return Boolean is
               begin
                  return A.Lettre <= B.Lettre ;
               end ;

               procedure Mon_Tri is
                  new TriMin(Mes_Elements,Compare,Mes_Tab) ;
            begin
               Mon_Tri(Ex) ;
               exit ;
            end ;
         when others =>
            Put_Line("Mauvais choix. Recommencez !") ;
      end case ;
   end loop ;
```

```
   Put("Tableau final:") ; Put(Ex) ;

end ;
```

7.2 Exemple d'un paquetage générique

Les paquetages peuvent aussi être génériques. Le principe est le même que pour les procédures. Une seule petite différence pratique toutefois : la seule façon de paramétrer un paquetage par une valeur du langage (un entier, un tableau, ...) est d'utiliser la généricité (alors que pour les procédures et fonction, on préférera le paramétrage classique, avec ses modes **in**, **out**, ...).

Un paramètre générique qui représente une valeur est une constante (on ne peut pas en modifier la valeur), et est passé par copie au moment de son instanciation (c'est vrai quel que soit le type du paramètre, contrairement aux cas des paramètres de procédures ou fonction).

On en voit ici un exemple avec les piles bornées, où la borne est un paramètre générique.

```
generic

   type Elt is private ;
   MaxLength: Natural ;

package Lifo is

  type Queue is private ;

  EmptyExc, FullExc: exception ;

  function Length(Q: Queue) return Natural ;

  procedure Empty (Q: out Queue) ;
  -- garantit Length(Q)=0

  procedure Insert(Q: in out Queue; X: in Elt) ;
  -- exec normale garantit X inséré dans Q
  -- FullExc levé garantit Q non modifié et Length(Q)=MaxLength

  procedure Destruct(Q: in out Queue; X: out Elt) ;
  -- exec normale garantit X élément supprimé de Q.
  --   l'élement X correspond au dernier élément inséré dans Q.
  -- EmptyExc levé garantit Q non modifié et Length(Q)=0


private

   type Tab is array(1..MaxLength) of Elt ;

   type Queue is record
      Suite: Tab ;
      Taille: Natural :=0 ;
```

```
      -- initialisation par défaut qui évite d'avoir un problème
      -- si l'utilisateur oublie de faire "Empty(Q)" au départ...
   end record ;

end ;
```

A l'utilisation de ce paquetage, on peut donc choisir la borne qu'on veut (éventuellement à l'exécution), comme dans le schéma ci-dessus, où on crée une pile de caractères dont la taille est demandée à l'utilisateur.

```
with Lifo, Ada.Text_Io, Ada.Integer_Text_Io ;
use Ada.Text_Io, Ada.Integer_Text_Io ;

procedure Bidule is
  N: Positive ;
begin
  Put("Entrez la taille de la pile (>0) : ") ;
  Get(N) ;
  declare
     package MaPile is new Lifo(Character,N) ;
     use MaPile ;  -- pour éviter d'avoir à écrire "MaPile.Insert"
                   -- ou "MaPile.Queue"
     X: Character ;
     Q: Queue ;
  begin
     Put("Premier élément de la pile (>0) : ") ;
     Get(X) ;
     Insert(Q,X) ;
     ...
  end ;
end ;
```

L'implémentation du paquetage correspond à ce qu'on a programmé en TD :

```
package body Lifo is

   function Length(Q: Queue) return Natural is
   begin
      return Q.Taille ;
   end ;

   procedure Empty (Q: out Queue) is
   begin
      Q.Taille := 0 ;
   end ;

   procedure Insert(Q: in out Queue; X: in Elt) is
   begin
      if Q.Taille = MaxLength then
         raise FullExc ;
```

```
      end if ;
      Q.Taille := Q.Taille+1 ;
      Q.Suite(Q.Taille) := X ;
   end ;

   procedure Destruct(Q: in out Queue; X: out Elt) is
   begin
      if Q.Taille = 0 then
         raise EmptyExc ;
      end if ;
      X := Q.Suite(Q.Taille) ;
      Q.Taille := Q.Taille-1 ;
   end ;

end ;
```