Généraliser un problème

Chapitre 13

Généricité

Toute unité (fonction, procédure, paquetage), pour être ré-utilisée, doit être rendue indépendante du contexte dans lequel elle a été conçue.

Une telle unité est appelée :

`unité générique`

Une `unité générique` est paramétrée avec des types et/ou des fonctions et/ou procédure.

Exemple (1) : fonction **auCarre**

La fonction auCarre a été développée dans le contexte d'une application spécifique :

```
function auCarre(x:Integer) return  Integer is
begin
   return x*x;
end auCarre;
```

Exemple (2) : généralisation

Une application de calcul scientifique a besoin d'une fonction `auCarre` appliquée à des `Float`.

Pour pouvoir ré-utiliser la fonction précédente, il aurait fallu la généraliser afin qu'elle soit applicable non seulement à des entiers, mais aussi à des flottants et éventuellement à d'autres types.

Exemple (3) : paramètres génériques

Quels sont les paramètres génériques ?

```
function auCarre(x:Integer) return  Integer is
begin
   return x*x;
end auCarre;
```

2 types de paramètres :

- paramètre de type
- paramètre fonctionnel

Exemple (4) : fonction générique

Déclaration de l'interface de l'unité

```
generic
   type Elt is private;
   with function "*"(u,v:Elt) return Elt is <>;
function auCarre(x:Elt) return  Elt;
```

Déclaration du corps de l'unité

```
function auCarre(x:Elt) return  Elt is
begin
    return x*x;
end auCarre;
```

Exemple (5)

`Elt` et la fonction ¨*¨ sont des **paramètres formels génériques :**

- `Elt` est un paramètre de type
- ¨*¨ est un paramètre fonctionnel

Exemple (6) : utilisation

Pour utiliser une fonction générique, il faut tout d'abord l'instancier, c'est à dire spécialiser ses paramètres formels génériques.

```
function carreInt is new auCarre(Integer);
function carreFloat is new auCarre(Float);
```

Le paramètre générique effectif correspondant à `"*"` peut être omis car déclaré avec **`is <>`**

```
with function "*"(u,v:Elt) return Elt is
   <>;
```

```
with auCarre;
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;use Ada.Integer_Text_io;
with Ada.Float_Text_io;use Ada.Float_Text_io;
procedure test_auCarre is
   function carreFloat is new auCarre(Float);
   function carreInt is new auCarre(Integer);
   i:Integer:=5;
   f:Float:=7.0;
begin
   put( "le carre de : " );put(i,1);
   put( " = " );put(carreInt(i),2);new_line;
   put( "le carre de : " );put(f,1,1,0);
   put( " = " );put(carreFloat(f),1,1,0);
end test_auCarre;
```

Les paramètres génériques (1)

Paramètre de type

```
type <ident_type> is private;
```

- Le type pourra être spécialisé par n'importe quel type
- Seules les opérations =, /=, := sont prédéfinies sur ce type

Les paramètres génériques (2)

Paramètre de type

```
type <ident_type> is (<>);
```

- Le type ne peut être spécialisé que par un type discret
- Les attributs des types discrets sont exploitables sur les objets du type

Les paramètres génériques (3)

Paramètre de type

```
type <ident_type> is range <>;
```

- Le type ne peut être spécialisé que par un type entier
- Les attributs des types discrets sont exploitables sur les objets du type
- Toutes les opérations définies sur des entiers sont utilisables

Les paramètres génériques (4)

Un paramètre usuel

```
<ident>:<type_param>:=<expression>;
```

Une fonction

```
with function <ident_fonction>
  (<ident_param>:<ident_type>
 {;<ident_param>:<ident_type>)}
                    return <ident_type> [is <>];
```

Déclaration d'une fonction générique : sémantique

L'évaluation de la déclaration se fait en 2 étapes :

- évaluation de l'interface
  - ajout de la liaison dans l'environnement associée à une valeur indéfinie
  - typage de la fonction
- évaluation du corps de la fonction
  - détermination de la fermeture dans l'environnement courant
  - remplacement de la valeur indéfinie par la fermeture

Exemple (1)

Evaluation de la déclaration de la fonction `auCarre` dans env.

- Ajout de la liaison

(auCarre,??)	env

- Détermination du type de la fonction générique

`Elt*(Elt→Elt)→(Elt→Elt)`

type de "*"

Exemple (2)

Evaluation de la déclaration du corps de la fonction `auCarre` dans envDef.

- Calcul de la fermeture

```
F=<<Elt*(Elt→Elt)→
                (Elt → corps de auCarre),envDef>>
```

- Remplacement de ?? dans env

(auCarre,F)	env

Sémantique de l'instanciation

Dans env, la liaison `(carreInt, F)` avec

`F=<<Elt*(Elt→Elt)→corps de auCarre,env>>` est déjà disponible

La déclaration dans env' de :

```
function carreInt is new auCarre(Integer);
```

crée une nouvelle fermeture `F'` pour la fonction `carreInt`, construite à partir de `F`

Son corps: `C=x→corps de auCarre appliqué aux paramètres génériques effectifs,`

Son environnement envInst est construit à partir de env par ajout des liaisons entre paramètres génériques formels et effectifs

envInst=<<C, | (Elt,Integer) | ("*",F*) | env | >>

`F*` est la fermeture de l'opération `*` trouvée dans l'environnement initial

Généralisation d'une fonction : Exemple (1)

Soit `min` la fonction qui retourne le plus petit entier parmi deux entiers donnés

```
function min(x,y:Integer) return Integer is
begin
   if x<=y
   then return x;
   else return y;
   end if;
end min;
```

2 paramètres génériques à la place de `Integer` et `<=`

Généralisation d'une fonction : Exemple (2)

On peut généraliser `min` pour tout type entier et non seulement `Integer`

```
generic
   type T is range <>;
function min(x,y:T) return T;
```

- un seul paramètre générique est retenu.

```
type Montant is new Integer range 0..800;
function minInt is new min(Montant);
```

- T→Montant
- <= → <= du type Montant

Généralisation d'une fonction :

```
function min(x,y:T) return T is
begin
  if x<=y
  then return x;
  else return y;
  end if;
end min;
```

Généralisation d'une fonction : Exemple (3)

On peut généraliser `min` pour tout type discret (entier et énuméré)

```
generic
   type T is (<>);
function min(x,y:T) return T;
```

- un seul paramètre générique est retenu.

```
type Couleur is (rouge,vert,bleu,jaune,violet,orange);
function minInt is new min(Couleur);
```

- T→Couleur
- <= → <= du type Couleur

Généralisation d'une fonction : Exemple (4)

On peut généraliser `min` pour tout type. Dans ce cas, il faut abstraire `<=`

```
generic
   type T is private;
   with function "<="(x,y:T) return Boolean;
function min(x,y:T) return T;
```

- La fonction <= n'existe pas pour une spécialisation quelconque de `T`. Elle sera définie avant l'instanciation de `min` et ajoutée comme paramètre générique effectif.

Généralisation d'une fonction : Exemple (5)

```
procedure test_min is
   type Complexe is
      record
         re:Float;
         im:Float;
      end record;
   function "<="(x,y:Complexe) return Boolean is
   begin
      return x.re**2+x.im**2<=y.re**2+y.im**2;
   end "<=";
   function minComplexe is new min(Complexe,"<=");
   u:Complexe:=(5.73,8.5);
   v:Complexe:=(7.27,8.99);
begin
 put( "le plus petit complexe est : " );
 put( "(" );put(minComplexe(u,v).re,1,2,0);
 put( "," );put(minComplexe(u,v).im,1,2,0);put_line( ")" );
end test_min;
```

Correspondance de types

Définition du type formel	Signification	Opérations possibles
`private`	Tout type	`= /= :=`
`(<>)`	Type discret	`= /= := < <= > >=` `first last pos val pred succ`

`range <>`	Tout type entier	`= /= := < <= > >=`
		`first last pos val pred succ`
		`+ - * / mod rem ** abs`

Procédures génériques (1)

```
-- specification :
procedure permuter(g,d : in out Integer);
-- Corps :
procedure permuter(g,d : in out Integer) is
   tmp : Integer;
begin
   tmp := g;
   g := d;
   d := tmp;
end permuter;
```

Cette procédure est trop spécialisée. On souhaiterait pouvoir permuter des entiers, des réels, des chaînes de caractères, …

Procédures génériques (2)

```
generic
   type Elt is private;
procedure permuter(g,d : in out Elt);

procedure permuter(g,d : in out Elt) is
   tmp : Elt;
begin
   tmp := g;
   g := d;
   d := tmp;
end permuter;
```

Procédures génériques (3)

Instanciations

```
procedure permuter_int is new permuter(Integer);

procedure permuter_car is new permuter(Character);

procedure permuter_chaine is new permuter(String);

procedure permuter_float is new permuter(Float);

procedure permuter_complexe is new permuter(Complexe);
```

Une puissante procédure générique : énoncé

Ecrire un programme qui convertit un caractère en son rang dans le type (un entier naturel) et accumule le carré de cet entier un nombre donné de fois

Une puissante procédure générique : interface

```
generic
   type T is private;
   type R is private;
   with function "+"(x:R;y:R) return R;
   with function F(a:T) return R is <>;
procedure accumule
             (x:in T;accu:in out R;n:in Natural);
-- accumule n fois F(x) dans accu
```

Une puissante procédure générique : corps

```
procedure accumule
      (x:in T;accu:in out R;n:in Natural) is
begin
   for i in 1..n loop
       accu:=F(x)+accu;
   end loop;
end accumule;
-- accumule n fois F(x) dans accu
```

Une puissante procédure générique : utilisation

```
with accumule;
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;use Ada.Integer_Text_io;
procedure test_accumule is
  function c2i_auCarre(c:Character)
                                  return Integer is
      i:Integer:= Character'pos(c);
  begin
      return i*i;
  end c2i_auCarre;
  procedure sommeCarres is new
     accumule(Character,Integer,"+",c2i_auCarre);
```

Une puissante procédure générique : utilisation

```
   c:Character:='Z';
   i:Integer:=0;
   n:Natural:=10;
begin
   put(Character'pos('Z'));
-- affiche 90
   sommeCarres(c,i,n);
-- accumule n fois c2i_auCarre(x) dans i
   put( "i=" );put(i,3);
end test_accumule;
-- résultat affiché : 81000
```

Paquetages génériques : déclaration

Permet d'abstraire un type de données et ses opérateurs.

Syntaxe : déclaration de l'interface

```
<déclaration_paquetage_générique>::=
generic
  <paramètres_génériques>
package <ident_module> is
  <liste_déclarations>
private
  <liste_déclarations>
end <ident_module>
```

Paquetages génériques : instanciation

Un paramètre générique peut être :

- Un paramètre usuel
- Un paramètre fonctionnel
- Un paramètre de type

Syntaxe

```
<instanciation_paquetage_générique>::=
package <ident_paquetage_instancié> is new
<ident_paquetage_générique>
      (<liste_paramètres_génériques_effectifs>);
```

Exemple (1/5)

```
-- déclaration dans env
generic
 type T is private;
 with function F(x:T) return T;
 N:Integer:=20;
package generique is
 subtype Entier is Integer range 1..N;
 type R is private;
 function lireChamp1(x:R) return T;
 function lireChamp2(x:R) return Entier;
 procedure ecrireChamp1(x:in T;y:out R);
 procedure ecrireChamp2(x:in Entier;y:out R);
```

Exemple (2/5)

```
private
   type R is
      record
         champ1:T;
         champ2:Entier;
      end record;
end generique;
```

Exemple (3/5)

```
package body generique is
 function lireChamp1(x:R) return T is
 begin
    return x.champ1;
 end lireChamp1;
 function lireChamp2(x:R) return Entier is
 begin
    return x.champ2;
 end lireChamp2;
 procedure ecrireChamp1(x:in T;y:out R) is
 begin
    y.champ1:=F(x);
 end ecrireChamp1;
 procedure ecrireChamp2(x:in Entier;y:out R) is
 begin
    y.champ2:=x;
 end ecrireChamp2;
end generique;
```

Exemple (4/5)

```
with generique;
with Ada.Text_io;use Ada.Text_io;
with Ada.Integer_Text_io;use Ada.Integer_Text_io;

procedure test_generique is
   function suivant(x:Character) return Character is
   begin
      return Character'succ(x);
   end suivant;
   package instance is new generique(Character,suivant);
   data:instance.R;
-- R type exporté
```

Exemple (5/5)

```
begin
   instance.ecrireChamp1('w',data);
   instance.ecrireChamp2(15,data);
   put(instance.lireChamp1(data));new_line;
   put(instance.lireChamp2(data),3);new_line;
end test_generique;

-- résultat affiché
x
 15
```

Un paquetage utile (1)

```
generic
 type T is private;
package listes is
 listeVide:exception;
 type Liste is private;
 function cons(x:T;l:Liste) return Liste;
 function vide return Liste;
 function estVide(l:Liste) return Boolean;
 function tete(l:Liste) return T;
 function queue(l:Liste) return Liste;
```

Un paquetage utile (2)

```
private
   type Cellule;
   type Liste is access Cellule;
   type Cellule is
       record
             valeur:T;
             suivant:Liste;
       end record;
end listes;
```

Un paquetage utile (3)

```
package body listes is
   function cons(x:T;l:Liste) return Liste is
   begin return new Cellule'(x,l);end cons;
   function vide return Liste is
   begin return NULL; end vide;
   function estVide(l:Liste) return boolean is
   begin return l=NULL; end estVide;
   function tete(l:Liste) return T is
   begin if l=NULL then raise listeVide;
         else return l.valeur; end if;
   end tete;
   function queue(l:Liste) return Liste is
   begin if l=NULL then raise listeVide;
         else return l.suivant;end if;
   end queue;
end listes;
```

Interfacer Ada et C (1)

Ada fournit des packages spécifiques et des pragma pour l'interface avec d'autres langages

Pragmas

- `Import(C,getenv)` pour utiliser le programme *C* `getenv` dans un programme *Ada*
- `Export(COBOL,saisieCapteur)` pour fournir au compilateur *COBOL* le programme *Ada* `saisieCapteur`

Interfacer Ada et C (2)

Le paquetage `Interfaces.C` contient les définitions des types C en Ada

Il existe d'autres paquetages utiles :

`Interfaces.C.Strings,`

qui contient par exemple le type `chars_ptr` correspondant au type C

`char* Interfaces.C.Pointers`

Interfacer Ada et C: exemple

On souhaite récupérer à partir d'un programme Ada une variable d'environnement du système d'exploitation

Il existe une fonction C

```
char* getenv(char* name);
```

Traduction Ada

```
function getenv(var:chars_ptr) return chars_ptr;
pragma Import(C,getenv);
```

Interfacer Ada et C: exemple (2)

```
with Interfaces.C.Strings; use Interfaces.C.Strings;
function get_env(variable:String) return String is
-- Retourne la valeur d'une variable d'environnement
-- donnée. Si non, retourne la chaîne vide
function getenv(var : Chars_ptr) return Chars_ptr;
pragma Import(C, getenv);
variable_en_format_C:Chars_ptr :=New_String(variable);
-- New_String convertit String en Chars_ptr
result_ptr : Chars_ptr := getenv(variable_en_format_C);
result : String := valeur_sans_exception(result_ptr);
begin
   free(variable_en_format_C);
   return result;
end get_env;
```

Interfacer Ada et C: exemple (3)

```
function valeur_sans_exception(S : Chars_ptr)
                                    return String is
-- Traduit S du type C char* dans le type Ada String.
-- si S est Null_Ptr, retourne "",
-- ne lève pas l'exception.
begin
   if S = Null_Ptr --constante définie en bibliothèque ;
   then return "";
   else return Value(S);
   -- Value(S)convertit Chars_ptr en String
   end if;
end valeur_sans_exception;
pragma Inline(valeur_sans_exception);
```

Interfacer Ada et C: exemple (4)

Il est facile maintenant d'obtenir la valeur d'une variable d'environnement (`Request_Method_Text`) à partir d'un programme Ada :

```
Request_Method_Text : String
                := get_Env ( "REQUEST_METHOD_TEXT" );
```