Ada Programming/Generics

Parametric polymorphism (generic units)

The idea of code reusability arises because of the necessity to construct programs on the basis of well established building blocks that can be combined to form an ampler and complex system. The reusability of code improves the productivity and the quality of software. One of the ways in which the Ada language supports this characteristic is by means of generic units. A generic unit is a unit that defines algorithms in terms of types and other operations that are not defined until the user instantiates them. Generic units can be subprograms and packages.

Note to C++ programmers: generic units are similar to C++ templates.

For example, to define a procedure for swapping variables of any (non-limited) type:

```
generic
  type Element_T is private; -- Generic formal type parameter
procedure Swap (X, Y : in out Element_T);
```

```
procedure Swap (X, Y : in out Element_T) is
  Temporary : constant Element_T := X;
begin
  X := Y;
  Y := Temporary;
end Swap;
```

The Swap subprogram is said to be generic. The subprogram specification is preceded by the generic formal part consisting of the reserved word generic followed by a list of generic formal parameters which may be empty. The entities declared as generic are not directly usable, it is necessary to instantiate them.

To be able to use Swap, it is necessary to create an instance for the wanted type. For example:

```
procedure Swap_Integers is new Swap (Integer);
```

Now the Swap_Integers procedure can be used for variables of type Integer.

The generic procedure can be instantiated for all the needed types. It can be instantiated with different names or, if the same identifier is used in the instantiation, each declaration overloads the procedure:

```
procedure Instance_Swap is new Swap (Float);
procedure Instance_Swap is new Swap (Day_T);
procedure Instance_Swap is new Swap (Element_T => Stack_T);
```

Similarly, generic packages can be used, for example, to implement a stack of any kind of elements:

```
generic
  Max: Positive;
  type Element_T is private;
package Generic(Stack is
  procedure Push (E: Element_T);
  function Pop return Element_T;
end Generic;
end Generic_Stack;
```

```
package body Generic(Stack is
  Stack: array (1 .. Max) of Element_T;
```
Top : Integer range 0 .. Max := 0; -- initialise to empty
-- ...
end Generic_Stack;

A stack of a given size and type could be defined in this way:

```
declare
  package Float_100_Stack is new Generic_Stack (100, Float);
  use Float_100_Stack;
begin
  Push (45.8);
  -- ...
end;
```

**Generic parameters**

The generic unit declares *generic formal parameters*, which can be:

- objects (of mode *in* or *in out* but never *out*)
- types
- subprograms
- instances of another, designated, generic unit.

When instantiating the generic, the programmer passes one *actual parameter* for each formal. Formal values and subprograms can have defaults, so passing an actual for them is optional.

**Generic formal objects**

Formal parameters of mode *in* accept any value, constant, or variable of the designated type. The actual is copied into the generic instance, and behaves as a constant inside the generic; this implies that the designated type cannot be limited. It is possible to specify a default value, like this:

```
generic
  Object : in Natural := 0;
```

For mode *in out*, the actual must be a variable.

One limitation with generic formal objects is that they are never considered static, even if the actual happens to be static. If the object is a number, it cannot be used to create a new type. It can however be used to create a new derived type, or a subtype:

```
generic
  Size : in Natural := 0;
package P is
  type T1 is mod Size; -- illegal!
  type T2 is range 1 .. Size; -- illegal!
  type T3 is new Integer range 1 .. Size; -- OK
  subtype T4 is Integer range 1 .. Size; -- OK
end P;
```

The reason why formal objects are nonstatic is to allow the compiler to emit the object code for the generic only once, and to have all instances share it, passing it the address of their actual object as a parameter. This bit of compiler technology is called *shared generics*. If formal objects were static, the compiler would have to emit one copy of the object code, with the object embedded in it, for each instance, potentially leading to an explosion in
object code size (*code bloat*).

(Note to C++ programmers: in C++, since formal objects can be static, the compiler cannot implement shared
generics in the general case; it would have to examine the entire body of the generic before deciding whether or not
to share its object code. In contrast, Ada generics are designed so that the compiler can instantiate a generic *without looking at its body.*)

**Generic formal types**

The syntax allows the programmer to specify which type categories are acceptable as actuals. As a rule of thumb: The syntax expresses how the generic sees the type, i.e. it assumes the worst, not how the creator of the instance sees the type.

This is the syntax of RM 12.5:

```
formal_type_declaration ::= 
    type defining_identifier[discriminant_part] is formal_type_definition;

formal_type_definition ::= formal_private_type_definition
    | formal_derived_type_definition
    | formal_discrete_type_definition
    | formal_signed_integer_type_definition
    | formal_modular_type_definition
    | formal_signed_integer_type_definition
    | formal_floating_point_definition
    | formal_ordinary_fixed_point_definition
    | formal_decimal_fixed_point_definition
    | formal_array_type_definition
    | formal_access_type_definition
    | formal_interface_type_definition
```

This is quite complex, so some examples are given below. A type declared with the syntax `type T (<>)` denotes a type with *unknown discriminants*. This is the Ada vernacular for indefinite types, i.e. types for which objects cannot be declared without giving an initial expression. An example of such a type is one with a discriminant without default, another example is an unconstrained array type.

<table>
<thead>
<tr>
<th>Generic formal type</th>
<th>Acceptable actual types</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type T(&lt;&gt;)</code> is limited private;</td>
<td>Any type at all. The actual type can be limited or not, indefinite or definite, but the generic treats it as limited and indefinite, i.e. does not assume that assignment is available for the type.</td>
</tr>
<tr>
<td><code>type T(&lt;&gt;)</code> is private;</td>
<td>Any nonlimited type: the generic knows that it is possible to assign to variables of this type, but it is not possible to declare objects of this type without initial value.</td>
</tr>
<tr>
<td><code>type T is private;</code></td>
<td>Any nonlimited definite type: the generic knows that it is possible to assign to variables of this type and to declare objects without initial value.</td>
</tr>
<tr>
<td><code>type T(&lt;&gt;)</code> is abstract tagged limited private;</td>
<td>Any tagged type, abstract or concrete, limited or not.</td>
</tr>
<tr>
<td><code>type T(&lt;&gt;)</code> is tagged limited private;</td>
<td>Any concrete tagged type, limited or not.</td>
</tr>
<tr>
<td><code>type T(&lt;&gt;)</code> is abstract tagged private;</td>
<td>Any nonlimited tagged type, abstract or concrete.</td>
</tr>
<tr>
<td><code>type T(&lt;&gt;)</code> is tagged private;</td>
<td>Any nonlimited, concrete tagged type.</td>
</tr>
<tr>
<td><code>type T(&lt;&gt;)</code> is new Parent;</td>
<td>Any type derived from Parent. The generic knows about Parent's operations, so can call them. Neither T nor Parent can be abstract.</td>
</tr>
</tbody>
</table>
In the body we can only use the operations predefined for the type category of the formal parameter. That is, the
generic specification is a contract between the generic implementor and the client instantiating the generic unit. This
is different to the parametric features of other languages, such as C++.

It is possible to further restrict the set of acceptable actual types like so:

<table>
<thead>
<tr>
<th>Generic formal type</th>
<th>Acceptable actual types</th>
</tr>
</thead>
<tbody>
<tr>
<td>type T (&lt;&gt;):abstract new Parent with private;</td>
<td>Any type, abstract or concrete, derived from Parent, where Parent is a tagged type, so calls to T's operations can dispatch dynamically.</td>
</tr>
<tr>
<td>type T (&lt;&gt;):is new Parent with private;</td>
<td>Any concrete type, derived from the tagged type Parent.</td>
</tr>
<tr>
<td>type T is ();</td>
<td>Any discrete type: integer, modular, or enumeration.</td>
</tr>
<tr>
<td>type T is range ();</td>
<td>Any signed integer type</td>
</tr>
<tr>
<td>type T is mod ();</td>
<td>Any modular type</td>
</tr>
<tr>
<td>type T is delta ();</td>
<td>Any (non-decimal) fixed point type</td>
</tr>
<tr>
<td>type T is digits ();</td>
<td>Any decimal fixed point type</td>
</tr>
<tr>
<td>type T is digits &lt;&gt;;</td>
<td>Any floating point type</td>
</tr>
<tr>
<td>type T is array (I) of E;</td>
<td>Any array type with index of type I and elements of type E (I and E could be formal parameters as well)</td>
</tr>
<tr>
<td>type T is access O;</td>
<td>Any access type pointing to objects of type O (O could be a formal parameter as well)</td>
</tr>
</tbody>
</table>

### Generic formal subprograms

It is possible to pass a subprogram as a parameter to a generic. The generic specifies a generic formal subprogram,
complete with parameter list and return type (if the subprogram is a function). The actual must match this parameter profile. It is not necessary that the names of parameters match, though.

Here is the specification of a generic subprogram that takes another subprogram as its parameter:

```ada
generic
    type Element_T is private;
    with function "*" (X, Y: Element_T) return Element_T;
function Square (X: Element_T) return Element_T is
begin
    return X * X;  -- The formal operator "*".
end Square;
```

And here is the body of the generic subprogram; it calls parameter as it would any other subprogram.

```ada
function Square (X: Element_T) return Element_T is
begin
    return X * X;  -- The formal operator "*".
end Square;
```

This generic function could be used, for example, with matrices, having defined the matrix product.

```ada
with Square;
with Matrices;
procedure Matrix_Example is
    function Square_Matrix is new Square
```
It is possible to specify a default with "the box" (is <>), like this:

```ada
generic
type Element_T is private;
with function "*" (X, Y: Element_T) return Element_T is <>;
```

This means that if, at the point of instantiation, a function "*" exists for the actual type, and if it is directly visible, then it will be used by default as the actual subprogram.

One of the main uses is passing needed operators. The following example shows this (follow download links for full example):

File: Algorithms/binary_search.adb

```ada
generic
type Element_Type is private;
...
with function "<" (Left : in Element_Type; Right : in Element_Type)
return Boolean is <>;
procedure Search
(Elements : in Array_Type;
 Search   : in Element_Type;
 Found    : out Boolean;
 Index    : out Index_Type'Base)
...
```

Generic instances of other generic packages

A generic formal can be a package; it must be an instance of a generic package, so that the generic knows the interface exported by the package:

```ada
generic
with package P is new Q (<>);
```

This means that the actual must be an instance of the generic package Q. The box after Q means that we do not care which actual generic parameters were used to create the actual for P. It is possible to specify the exact parameters, or to specify that the defaults must be used, like this:

```ada
generic
-- P1 must be an instance of Q with the specified actual parameters:
with package P1 is new Q (Param1 => X, Param2 => Y);

-- P2 must be an instance of Q where the actuals are the defaults:
with package P2 is new Q;
```
It is all or nothing: if you specify the generic parameters, you must specify all of them. Similarly, if you specify no parameters and no box, then all the generic formal parameters of Q must have defaults. The actual package must, of course, match these constraints.

The generic sees both the public part and the generic parameters of the actual package (Param1 and Param2 in the above example).

This feature allows the programmer to pass arbitrarily complex types as parameters to a generic unit, while retaining complete type safety and encapsulation. *(example needed)*

It is not possible for a package to list itself as a generic formal, so no generic recursion is possible. The following is illegal:

```ada
with A;
generic
    with package P is new A (<>);
package A; -- illegal: A references itself
```

In fact, this is only a particular case of:

```ada
with A; -- illegal: A does not exist yet at this point!
package A;
```

which is also illegal, despite the fact that A is no longer generic.

**Instantiating generics**

To instantiate a generic unit, use the keyword **new**:

```ada
function Square_Matrix is new Square
    (Element_T => Matrices.Matrix_T, "*" => Matrices.Product);
```

Notes of special interest to C++ programmers:

- The generic formal types define *completely* which types are acceptable as actuals; therefore, the compiler can instantiate generics without looking at the body of the generic.
- Each instance has a name and is different from all other instances. In particular, if a generic package declares a type, and you create two instances of the package, then you will get two different, incompatible types, even if the actual parameters are the same.
- Ada requires that all instantiations be explicit.
- It is not possible to create special-case instances of a generic (known as "template specialisation" in C++).

As a consequence of the above, Ada does not permit template metaprogramming. However, this design has significant advantages:

- the object code can be shared by all instances of a generic, unless of course the programmer has requested that subprograms be inlined; there is no danger of code bloat.
- when reading programs written by other people, there are no hidden instantiations, and no special cases to worry about. Ada follows the Law of Least Astonishment.
Advanced generics

Generics and nesting

A generic unit can be nested inside another unit, which itself may be generic. Even though no special rules apply (just the normal rules about generics and the rules about nested units), novices may be confused. It is important to understand the difference between a generic unit and instances of a generic unit.

**Example 1.** A generic subprogram nested in a nongeneric package.

```ada
package Bag_Of_Strings is
  type Bag is private;
  generic
    with procedure Operator (S : in out String);
  procedure Apply_To_All (B : in out Bag);
private
  -- omitted
end Bag_Of_Strings;
```

To use `Apply_To_All`, you first define the procedure to be applied to each String in the Bag. Then, you instantiate `Apply_To_All`, and finally you call the instance.

```ada
with Bag_Of_Strings;
procedure Example_1 is
  procedure Capitalize (S : in out String) is separate; -- omitted
  procedure Capitalize_All is
    new Bag_Of_Strings.Apply_To_All (Operator => Capitalize);
  begin
    Capitalize_All (B);
  end Example_1;
```

**Example 2.** A generic subprogram nested in a generic package

This is the same as above, except that now the Bag itself is generic:

```ada
generic
  type Element_Type (<>) is private;
package Generic_Bag is
  type Bag is private;
  generic
    with procedure Operator (S : in out Element_Type);
  procedure Apply_To_All (B : in out Bag);
private
  -- omitted
end Generic_Bag;
```

As you can see, the generic formal subprogram `Operator` takes a parameter of the generic formal type `Element_Type`. This is okay: the nested generic sees everything that is in its enclosing unit.

You cannot instantiate `Generic_Bag.Apply_To_All` directly, so you must first create an instance of `Generic_Bag`, say `Bag_Of_Strings`, and then instantiate `Bag_Of_Strings.Apply_To_All.`
with Generic_Bag;
procedure Example_2 is
  procedure Capitalize (S : in out String) is separate; -- omitted
package Bag_Of_Strings is
  new Generic_Bag (Element_Type => String);
procedure Capitalize_All is
  new Bag_Of_Strings.Apply_To_All (Operator => Capitalize);
B : Bag_Of_Strings.Bag;
begin
  Capitalize_All (B);
end Example_2;

Generics and child units

Example 3. A generic unit that is a child of a nongeneric unit.

Each instance of the generic child is a child of the parent unit, and so it can see the parent's public and private parts.

package Bag_Of_Strings is
  type Bag is private;
private
  -- omitted
end Bag_Of_Strings;
generic
  with procedure Operator (S : in out String);
procedure Bag_Of_Strings.Apply_To_All (B : in out Bag);

The differences between this and Example 1 are:

• Bag_Of_Strings.Apply_To_All is compiled separately. In particular, Bag_Of_Strings.Apply_To_All might have been written by a different person who did not have access to the source text of Bag_Of_Strings.
• Before you can use Bag_Of_Strings.Apply_To_All, you must with it explicitly; withing the parent, Bag_Of_Strings, is not sufficient.
• If you do not use Bag_Of_Strings.Apply_To_All, your program does not contain its object code.
• Because Bag_Of_Strings.Apply_To_All is at the library level, it can declare controlled types; the nested package could not do that in Ada 95. In Ada 2005, one can declare controlled types at any level.

with Bag_Of_Strings.Apply_To_All; -- implicitly withs Bag_Of_Strings, too
procedure Example_3 is
  procedure Capitalize (S : in out String) is separate; -- omitted
  procedure Capitalize_All is
    new Bag_Of_Strings.Apply_To_All (Operator => Capitalize);
B : Bag_Of_Strings.Bag;
begin
  Capitalize_All (B);
end Example_3;

Example 4. A generic unit that is a child of a generic unit

This is the same as Example 3, except that now the Bag is generic, too.
generic

  type Element_Type (<>) is private;

package Generic_Bag is

  type Bag is private;

private

  -- omitted

end Generic_Bag;


  with procedure Operator (S : in out Element_Type);

procedure Generic_Bag.Apply_To_All (B : in out Bag);

with Generic_Bag.Apply_To_All;

procedure Example_4 is

  procedure Capitalize (S : in out String) is separate; -- omitted

  package Bag_Of_Strings is
    new Generic_Bag (Element_Type => String);
  procedure Capitalize_All is
    new Bag_Of_Strings.Apply_To_All (Operator => Capitalize);

begin
  Capitalize_All (B);
end Example_4;

Example 5. A parameterless generic child unit

Children of a generic unit must be generic, no matter what. If you think about it, it is quite logical: a child unit sees the public and private parts of its parent, including the variables declared in the parent. If the parent is generic, which instance should the child see? The answer is that the child must be the child of only one instance of the parent, therefore the child must also be generic.

generic

  type Element_Type (<>) is private;

  type Hash_Type is (<>);

  with function Hash_Function (E : Element_Type) return Hash_Type;

package Generic_Hash_Map is

  type Map is private;

private

  -- omitted

end Generic_Hash_Map;

Suppose we want a child of a Generic_Hash_Map that can serialise the map to disk; for this it needs to sort the map by hash value. This is easy to do, because we know that Hash_Type is a discrete type, and so has a less-than operator. The child unit that does the serialisation does not need any additional generic parameters, but it must be generic nevertheless, so it can see its parent's generic parameters, public and private part.

generic

package Generic_Hash_Map.Serializer is

  procedure Dump (Item : in Map; To_File : in String);

  procedure Restore (Item : out Map; From_File : in String);
To read and write a map to disk, you first create an instance of `Generic_Hash_Map`, for example `Map_Of_Unbounded_Strings`, and then an instance of `Map_Of_Unbounded_Strings.Serializer`:

```ada
with Ada.Strings.Unbounded;
with Generic_Hash_Map.Serializer;
procedure Example_5 is
  use Ada.Strings.Unbounded;
  function Hash (S : in Unbounded_String) return Integer is separate; -- omitted
  package Map_Of_Unbounded_Strings is
    new Generic_Hash_Map (Element_Type => Unbounded_String,
                           Hash_Type => Integer,
                           Hash_Function => Hash);
  package Serializer is
    new Map_Of_Unbounded_Strings.Serializer;
  M : Map_Of_Unbounded_Strings.Map;
begin
  Serializer.Restore (Item => M, From_File => "map.dat");
end Example_5;
```

**References**

Article Sources and Contributors


License

Creative Commons Attribution-Share Alike 3.0 Unported
http://creativecommons.org/licenses/by-sa/3.0/