## 21.4a Multiple views

As mentioned in Section 21.1, multiple inheritance can often be avoided by simply adding one property as a component as in the case of the type Adventure. Sometimes this is not enough and a closer relationship is required in which the new type can truly be used in either context. This can be done by an interesting use of access discriminants. In Section 18.7 we showed how an inner component of a record can refer to the record as a whole thus

```
type Inner(Ptr: access Outer) is limited ...

type Outer is limited
   record
      ...
      Component: Inner(Outer'Access);
      ...
   end record;
```

The types Inner and Outer can both be extensions of other types. For example, the type Inner might be an extension of some (limited) type Node such as

```
type Node;
type Node_Ptr is access all Node'Class;

type Node is tagged limited
   record
      Left, Right: Node_Ptr;
      Value: T;
   end record;
```

which can be used to form a tree. Note in particular that the access discriminant of Inner could be class wide thus

```
type Inner(Ptr: access Outer'Class) is new Node with ...
```

so that heterogeneous chains can be constructed. The important point is that we can navigate over the tree which consists of the components of type Inner linked together (via the type Node) but at any point in the tree we can reach the enclosing Outer record as a whole by the access discriminant Ptr as illustrated in Figure 1.

The result is that we have two quite different views of such an object and can move from one view to the other. Thus given the outer view of an object O, the inner view is obtained by selecting the component of type Inner thus

```
O.Component
```

whereas given a view I of the inner part we can regain the outer view by

```
I.Ptr.all
```

Note the lack of symmetry since the outer view is dominant. (We could create a symmetrical situation where the outer type is a container for two inner views.)

As a simple example suppose we have a linked list of coloured things using types Cell and Cell_Ptr. So the essence is

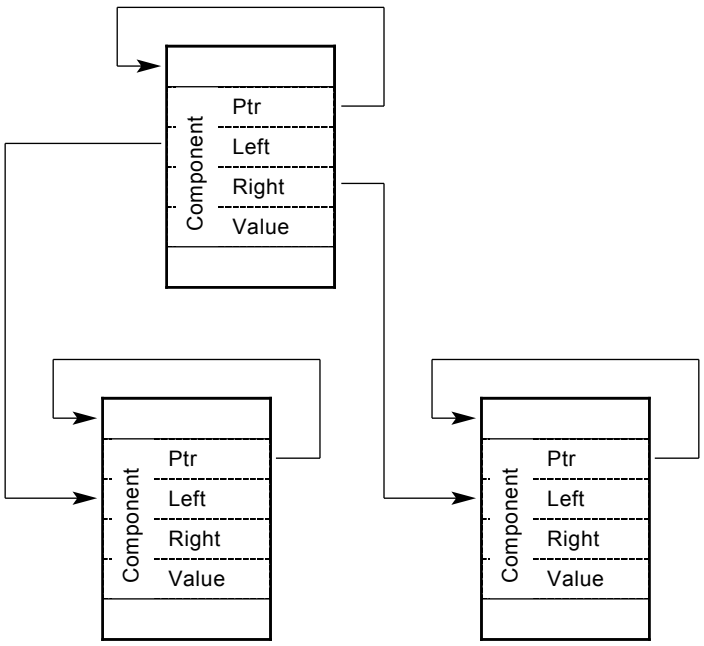


**Figure 1** Navigation over a tree.

```
type Cell_Ptr is access all Cell'Class;

type Cell is tagged limited
   record
      Next: Cell_Ptr;
   end record;

type Coloured is new Cell with
   record
      C: Colour;
   end record;

procedure Set_Colour(This: in out Coloured; To: in Colour);
```

where Set_Colour is a primitive operation of Coloured. Note that the type Cell is a limited type because an extension of it is going to have a limited component and only a limited record type can have limited components.

On the other hand we also have our type Object and its descendants, Circle, Point, Polygon and so on. Now suppose we want to create a type which can act as either a coloured thing on a list or as a geometrical object according to our point of view. This is not quite the same as simply extending an Object with a colour as we shall see. We assume for this application that Object is limited.

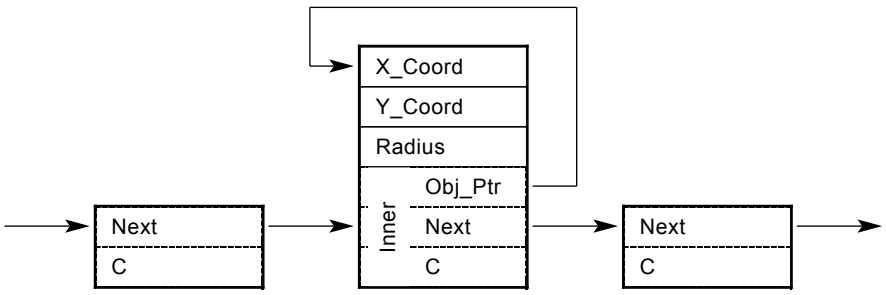


**Figure 2** The coloured circle in the list of colours.

We do this in two stages. First we create a sort of gluon which can stick the types together

```
type Coloured_Gluon(Obj_Ptr: access Object'Class) is
                                        new Coloured with null record;
```

This is of course directly derived from Coloured and will inherit Set_Colour; we could redefine this operation to use the properties of the Object available through the access discriminant.

The second stage is to extend say a Circle by

```
type Coloured_Circle is new Circle with
  record
    Inner: Coloured_Gluon(Coloured_Circle'Access);
  end record;
```

This is of course a circle and will inherit the properties of the type Circle, its Radius and functions such as Area; we could redefine this operation to take account of the associated colour available through the internal component. We have now created the structure shown in Figure 2 which illustrates one coloured circle threaded on a list with two plain colours either side.

The object is truly a coloured circle but from the point of view of the list of colours it is a circular colour. The situation is of course not symmetric; we leave the reader to create the reverse structure as an exercise.

In Section 21.4 we stated that one problem with the extension approach is that an object can be on only one list at a time. This can be overcome using the technique we have just discussed by making the object have several components that are derived from linking types. This provides several views of the one object and each view can be on its own list.

As a whimsical example, suppose we have a collection of objects such as garments. They will have various properties such as style (long, short, French), pattern (spotted, striped, plain), and of course they will come in different colours and different sizes. We need to search the stock for various combinations. To make

this easier, we might decide to keep them on lists according to any of these properties.

The neatest way to do this is to introduce just one gluon type

```
type Cell_Gluon(Ptr: access Garment) is new Cell with null record;
```

and we can now add several components of this type according to the number of lists required. Suppose we decide to have two lists, one for size and one for colour, so that each garment will be on two lists, its size list and its colour list. Of course several garments may have the same combination of colour and size. We can now write the type Garment as

```
type Garment is limited
   record
      Style: Style_Type;
      Pattern: Pattern_Type;
      C: Colour;
      S: Size;
      Colour_Link: Cell_Gluon(Garment'Access);
      Size_Link: Cell_Gluon(Garment'Access);
   end record;
```

and then garments can be placed on various lists for each colour and size resulting in cross-linked chains as shown in Figure 3.

The figure shows six garments, two large red ones and one each of large blue, small white, small red and small blue. Note how the lists are simply chains of pointers that happen to thread through the records. Of course the components Next are actually Colour_Link.Next and Size_Link.Next and so are distinguished. Note that the size and colour are given by direct components of the type Garment in contrast to the technique used for the coloured circle.

Finally note that the type Cell_Gluon could have a class wide discriminant Garment'Class so that the chains could link through garments with different properties. We could take this to its extreme by defining a root type Stuff from which all things to be linked are derived thus

```
type Stuff is abstract tagged limited null record;
type Cell_Gluon(Ptr: access Stuff'Class) is new Cell with null record;

type Garment is new Stuff with
   record
      ...
      A_Link: Cell_Gluon(Garment'Access);
   end record;
```

Other types also derived from Stuff could be linked into the chains in a similar manner – indeed, so could any object of type Cell but it has to be part of a gluon to change its view.

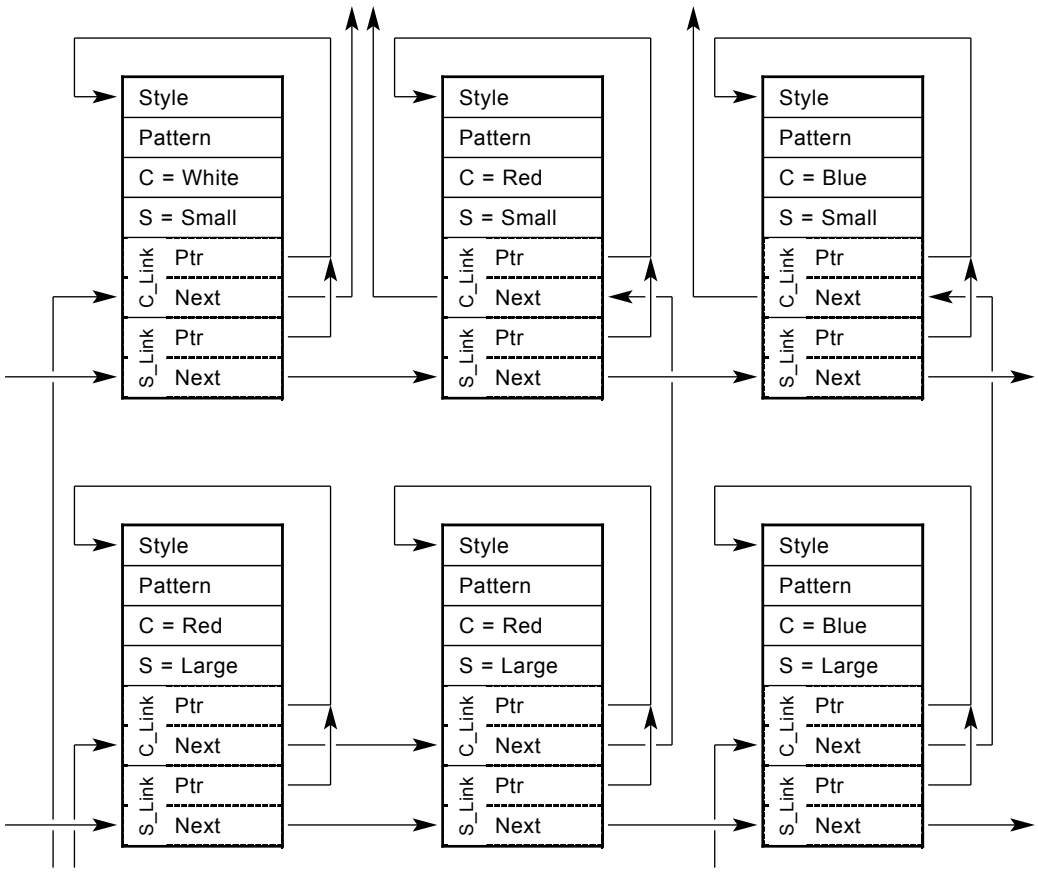


**Figure 3** Double chains of garments.

## Exercise

**1** For the example of the coloured gluon redefine Set_Colour so that it will not update an object with area less than 10.0. Then do it with the condition that the Radius must not be less than 2.0.

**2** Redefine the Area of a Coloured_Circle so that it is deemed to be twice as big if Red.

**3** Do the gluing the other way round and draw the corresponding diagram.

*Answers*

**1**
```
procedure Set_Colour(This: in out Coloured_Gluon;
                     C: in Colour) is
begin
   if Area(This.Obj_Ptr.all) >= 10.0 then
      Set_Colour(Coloured(This), C));
   end if;
end Set_Colour;

procedure Set_Colour(This: in out Coloured_Gluon;
                     C: in Colour) is
   Circle_View: Circle renames Circle(This.Obj_Ptr.all);
begin
   if Circle_View.Radius >= 2.0 then
      Set_Colour(Coloured(This), C);
   end if;
end Set_Colour;
```

Observe the view conversion in the second case. If the gluon has been applied to an object which is not a Circle (or derived from a Circle) then Constraint_Error is raised.

**2**
```
function Area(This: Coloured_Circle) return Float is
   Nominal_Area: Float := Area(Circle(This));
begin
   if This.Inner.C = Red then
      Nominal_Area := Nominal_Area * 2.0;
   end if;
   return Nominal_Area;
end Area;
```

**3**
```
type Circular_Gluon(Col_Ptr: access Coloured'Class) is
     new Circle with null record;

type Circular_Colour is new Coloured with
   record
      Inner: Circular_Gluon(Circular_Colour'Access);
   end record;
```

The diagram shows a circular colour in a list of colours.