Gem #9: Надклассовые операции, итераторы и типовые алгоритмы

Автор: Matthew Heaney (On2 Technologies)

Краткое содержание: Gem Ada #9 – типовые алгоритмы можно использовать для работы с любыми контейнерами (включая массивы), в которых возможна итерация элементов. В данном примере показано, как можно систематически изменять операцию копирования контейнеров одного класса, превратив ее в типовой алгоритм, подходящий для любого контейнера.

Давайте начнем…

В последнем Gem #8 мы использовали класс стека для демонстрации фабричных функций (для создания объектов итератора) и реализовали операцию присваивания, отправленную по типу целевого стека. Мы упомянули попутно, что эта операция может быть реализована путем отправки в исходный стек, поэтому давайте покажем, как это сделать.

Мы переупорядочиваем параметры так, чтобы исходный стек был первым в списке параметров (так, чтобы он был «отмеченным приемником» вызова в стиле префикса) и изменили его тип от класса к конкретному. Мы также меняем название операции из Assign в Copy, для каждого соглашения. Новая декларация выглядит следующим образом:

```
procedure Copy
  (Source : Stack;
   Target : in out Stack'Class) is abstract;
```

В предыдущем примере (Gem #8) нам пришлось заполнить целевой стек в обратном порядке, чтобы элементы были в правильном порядке. Мы смогли это сделать, потому что операция была реализована конкретным типом и, следовательно, имела прямой доступ к представлению (целевого) стека. Здесь целевой тип является общедоступным, поэтому единственный способ его заполнения - в прямом порядке, используя Push. Это означает, что нам придется перебирать исходный стек в обратном порядке, чтобы элементы были правильно упорядочены в целевом объекте.

Тип ограниченного стека реализуется как массив, поэтому реализовать Copy легко, потому что нижняя часть стека начинается в начале массива:

```
procedure Copy
  (Source : Stack;  -- bounded stack (array-based)
   Target : in out Stacks.Stack'Class)
is
begin
   Target.Clear;

   for I in 1 .. Source.Top_Index loop    -- from bottom to top
      Target.Push (Source.Elements (I));  -- Elements is the array
   end loop;
end Copy;
```

Мы также сказали в более раннем (Gem #8), что операция не должна быть примитивной для типа. Если мы изменяем исходный тип стека на classwide, то сама работа становится classwide:

```
procedure Copy2  -- classwide op, not primitive
  (Source : Stack'Class;
```

```
   Target : in out Stack'Class);
```

Если мы создадим тип исходного стека по классу (classwide), тогда нам понадобится другой способ перебора элементов исходного стека в обратном порядке, поскольку у нас больше нет доступа к его представлению.

Для этого мы изменим тип курсора, чтобы включить некоторые дополнительные операции. Сначала мы добавим новую фабричную функцию, чтобы построить объект-курсор, который (изначально) обозначает элемент в нижней части стека:

```
function Bottom_Cursor
  (Container : not null access constant Stack)
  return Cursor'Class is abstract;
```

Нам также понадобится операция перемещения курсора к элементу, который предшествует текущему элементу:

```
procedure Previous (Position : in out Cursor) is abstract;
```

Это дает нам все необходимое, чтобы превратить Copy в операцию класса, так что это нужно выполнить только один раз:

```
procedure Copy2
  (Source : Stack'Class;
   Target : in out Stack'Class)
is
   C : Cursor'Class := Bottom_Cursor (Source'Access);

begin
   Target.Clear;

   while C.Has_Element loop
      Target.Push (C.Element);
      C.Previous;
   end loop;
end Copy2;
```

Обратите внимание, что мы объявили (classwide) классовую операцию стека в корневом пакете (см. Stacks.ads), но его так же легко объявили как общую дочернюю процедуру:

```
generic
procedure Stacks.Generic_Copy3
  (Source : Stack'Class;
   Target : in out Stack'Class);
```

Фактически, мы могли полностью перенести операцию из иерархии пакетов:

```
with Stacks;
generic
   with package Stack_Types is new Stacks (<>);
   use Stack_Types;
procedure Generic_Stack_Copy4
  (Source : Stack'Class;
   Target : in out Stack'Class);
```

Мы можем обобщить это еще больше, так что алгоритм копирования работает для любого типа стека:

```
generic
   type Stack_Type (<>) is limited private;
```

```
   type Cursor_Type (<>) is private;
   type Element_Type (<>) is private;

   with function Bottom_Cursor
     (Stack : Stack_Type)
     return Cursor_Type is <>;
   with procedure Push
     (Stack : in out Stack_Type;
      Item  : Element_Type) is <>;
   with procedure Previous
     (Cursor : in out Cursor_Type) is <>;
   …
procedure Generic_Stack_Copy5
  (Source : Stack_Type;
   Target : in out Stack_Type);
```

Это иллюстрирует разницу между динамическим полиморфизмом меченых типов (tagged types) и статическим полиморфизмом дженериков (generics). Больше нет необходимости в классе стека (с выделенной операцией копирования, которая работает только для типов в этом классе), поскольку общий алгоритм работает для любого стека. (Именно так разработана стандартная библиотека контейнеров. Типы контейнеров отмечены тегами, но они не являются членами общего класса.)

Создание экземпляра этой операции в нашем типе стека легко, так как имена общих фактических операций соответствуют именам общих формальных операций, поэтому нам не нужно их явно указывать (так как общие форматы отмечены как принимающие <> по умолчанию):

```
procedure Test_Copy5 (S : Stack) is
   procedure Copy5 is
      new Generic_Stack_Copy5
     (Stack,
      Cursor,
      Integer); -- default everything else

   T : Stack (S.Length);

begin
   Copy5 (Source => S, Target => T);
end;
```

Но зачем останавливаться? Мы можем написать общий алгоритм копирования для любого типа контейнера. Нам просто нужно немного обобщить итерацию, чтобы означать «посещать эти элементы так, как это имеет смысл для этого исходного контейнера», и обобщать вставку, чтобы означать «добавить этот элемент таким образом, который имеет смысл для этого целевого контейнера». Декларация будет:

```
generic
   type Container_Type (<>) is limited private;
   type Cursor_Type (<>) is private;
   type Element_Type (<>) is private;

   with function First
     (Container : Container_Type)
     return Cursor_Type is <>;
   with procedure Insert
     (Container : in out Container_Type;
      Item      : Element_Type) is <>;
   with procedure Advance
     (Cursor : in out Cursor_Type) is <>;
```

```
procedure Generic_Copy6
  (Source : Container_Type;
   Target : in out Container_Type);
```

Мы можем создать экземпляр этого метода, используя наш тип стека, но обратите внимание, что общие фактические данные больше не соответствуют общим формам, поэтому мы должны указать их явно:

```
procedure Test_Copy6 (S : Stack) is
   procedure Copy6 is
      new Generic_Copy6
     (Stack,
      Cursor,
      Integer,
      First => Bottom_Cursor,
      Insert => Push,
      Advance => Previous);

   T : Stack (S.Length);

begin
   Copy6 (Source => S, Target => T);
end;
```

Одно из предположений, которое мы сделали здесь, состоит в том, что исходный и целевой контейнеры имеют один и тот же тип. Предположим, мы хотели бы скопировать элементы в стек, например, в массив. Один из подходов состоял бы в том, чтобы ввести другой общий формальный тип контейнера (тип «исходного контейнера», который отличается от типа «целевой контейнер»), но есть другой способ. Рассмотрим реализацию алгоритма копирования:

```
procedure Generic_Copy6
  (Source : Container_Type;
   Target : in out Container_Type)
is
   C : Cursor_Type := First (Source);

begin
   Clear (Target);
   while Has_Element (C) loop
      Insert (Target, Element (C));
      Advance (C);
   end loop;
end Generic_Copy6;
```

Обратите внимание, что единственное, что мы делаем с исходным контейнером, это использовать его для построения курсора. Если мы передаем курсор напрямую, это исключает упоминание исходного стека, что, в свою очередь, позволяет контейнерам источника и цели быть разными. Теперь наш алгоритм становится следующим:

```
generic
   type Container_Type (<>) is limited private;
   type Cursor_Type (<>) is private;
   type Element_Type (<>) is private;
   …
procedure Generic_Copy7
  (Source : Cursor_Type;
   Target : in out Container_Type);
```

Теперь мы можем скопировать из целочисленного стека в массив следующим образом:

```
  procedure Copy_From_Stack_To_Array (S : in out Stack) is
    T : Integer_Array (1 .. S.Length);
    I : Positive := T'First;

    procedure Insert
      (Container : in out Integer_Array;
       Item      : Integer)
    is
    begin
       Container (I) := Item;
       I := I + 1;
    end;

    procedure Copy7 is
       new Generic_Copy7
      (Integer_Array,
       Cursor,
       Integer,
       Advance => Next);

 begin
    Copy7 (Source => S.Top_Cursor, Target => T);
 end Copy_From_Stack_To_Array;
```

Целевой «контейнер» - это всего лишь массив. Единственное, что нам нужно сделать, это синтезировать операцию вставки, чтобы передать ее как фактическое обобщение. Мы также можем использовать один и тот же алгоритм для перехода в другую сторону: от массива до стека:

```
  procedure Copy_From_Array_To_Stack (S : Integer_Array) is
    T : Stack (S'Length);

    function Has_Element (I : Natural) return Boolean is
    begin
       return I > 0;
    end;

    function Element (I : Natural) return Integer is
    begin
       return S (I);
    end;

    procedure Advance (I : in out Natural) is
    begin
       I := I - 1;
    end;

    procedure Copy7 is
       new Generic_Copy7
      (Stack,
       Natural,
       Integer,
       Insert => Push);

  begin
    Copy7 (Source => S'Last, Target => T);
  end Copy_From_Array_To_Stack;
```

Теперь исходный контейнер представляет собой массив, а «курсор» - это только индекс массива (целочисленный подтип). У нас есть знакомая проблема обеспечения того, чтобы целевой стек был заполнен в правильном порядке. Как и раньше, мы просто

перебираем массив в обратном направлении, передавая индекс S'Last в качестве исходного значения курсора, а затем «продвигаем» курсор, уменьшая значение индекса.

Алгоритм еще может быть обобщен. В этой окончательной версии мы исключаем общий тип формального элемента. Это означает, что нам нужно будет изменить общую формальную операцию Insert, передав исходный курсор в качестве параметра вместо исходного элемента. Объявление общего алгоритма теперь становится:

```
generic
   type Container_Type (<>) is limited private;
   type Cursor_Type (<>) is private;

   with procedure Insert
     (Target : in out Container_Type;
      Source : Cursor_Type) is <>;
   …
procedure Generic_Copy8
  (Source : Cursor_Type;
   Target : in out Container_Type);
```

Алгоритм теперь агностик относительно сопоставления от курсора к элементу (поскольку он даже не знает об элементах), что является более гибким, поскольку он позволяет клиенту выбирать любой механизм, наиболее эффективный. Чтобы использовать новый алгоритм, все, что нам нужно сделать, это внести небольшое изменение в общую фактическую процедуру Вставки (Insert):

```
procedure Copy_From_Stack_To_Array (S : in out Stack) is
  T : Integer_Array (1 .. S.Length);
  I : Positive := T'First;

  procedure Insert
    (Target : in out Integer_Array;
     Source : Cursor)  -- now a cursor instead of an element
  is
  begin
     Target (I) := Element (Source);
     I := I + 1;
  end;

  procedure Copy8 is
     new Generic_Copy8
    (Integer_Array,
     Cursor,
     Advance => Next);

begin
   Copy8 (Source => S.Top_Cursor, Target => T);
end Copy_From_Stack_To_Array;
```

Основная идея заключается в том, что общий алгоритм может использоваться в широком диапазоне контейнеров (включая типы массивов). Курсор обеспечивает доступ к элементам в контейнере, но, как мы видели, после того, как у вас есть курсор, сам контейнер исчезает. С точки зрения общего алгоритма, контейнер представляет собой просто последовательность элементов.

Обсуждение…