Gem #128: Итераторы в Ada 2012 - Часть 2.

Автор: Emmanuel Briot — AdaCore

Краткое содержание: В части 1 мы описали базовые формы итераторов в Ada 2012 и привели несколько примеров. В этой части мы обсудим данную тему более детально и покажем, как создавать итераторы пользовательских структур данных. Начнем с изучения двух функций поддержки, введенных в Ada 2012..

Давайте начнём…

В части 1 мы обсудили основные формы итераторов в Ada 2012 и привели несколько простых примеров. Эта часть содержит более подробные сведения о создании итераторов для собственных структур данных. Мы начнем с изучения двух вспомогательных функций, представленных в Ada 2012.

Первый - это новый универсальный пакет `Ada.Iterator_Interfaces`. Этот пакет определяет два абстрактных типа `Forward_Iterator` и `Reverse_Iterator`. Цель состоит в том, что каждый контейнер должен объявить расширения этих и предоставить конкретные реализации для их примитивных операций. Короче говоря, итератор инкапсулирует курсор и контейнер и скрывает `First`, `Has_Element` и `Next` операции.

Вторая новая функция относится к ссылочному типу. Ссылочный тип `A` - это запись с дискриминантом доступа, который определяет аспект "`Implicit_Dereference`". Это фактический тип, управляемый итераторами элемента контейнера, и аспект устраняет необходимость записи "`.all`" каждый раз при ссылке на элемент.

Вот пример такого объявления, взятый из стандартного пакета `Ada.Containers.Doubly_Linked_Lists`:

```
type Constant_Reference_Type
        (Element : not null access constant Element_Type)
           is private with Implicit_Dereference => Element;
```

Всякий раз, когда у нас есть такая ссылка, например `E` типа `Constant_Reference_Type`, мы можем просто использовать имя "`E`", и это автоматически интерпретируется как "`E.Element.all`". Еще одно преимущество этого типа перед простым доступом к элементу заключается в том, что он гарантирует, что пользователь не может случайно освободить элемент.

Теперь, когда мы понимаем, что такое итераторы и ссылки, мы можем начать применять их к нашим собственным структурам данных.

Предположим, мы создаем собственную структуру данных (например, граф, очередь или что-либо, что не является прямым экземпляром контейнера Ada 2005). Следующие примеры оформлены как "`list`", но это действительно относится к любой структуре данных. Давайте также предположим, что контейнер содержит неограниченные элементы типа "`T'Class`", что дает нам более реалистичный и интересный пример, чем пример части 1, который содержит только целые числа.

Чтобы обеспечить итераторы для этой структуры данных, нам необходимо определить ряд аспектов Ada 2012, более подробно описанных ниже.

```
type T_List is ...  -- a structure of such types
   with Default_Iterator  => Iterate,
```

```
         Iterator_Element  => T'Class,
         Constant_Indexing => Element_Value;

   type Cursor is private;
   function Has_Element (Pos : Cursor) return Boolean;
   -- As for Ada 2005 containers

   package List_Iterators is
      new Ada.Iterator_Interfaces (Cursor, Has_Element);

   function Iterate (Container : T_List)
      return List_Iterators.Forward_Iterator'Class;
   -- Returns our own iterator, which in general will be defined in the
   -- private part or the body.

   function Element_Value (Container : T_List; Pos : Cursor)
      return T'Class;
   -- Could also return a reference type as defined in the Part 1 Gem
```

Для тех, кто не знаком с аспектами в Ada 2012, стоит отметить, что они могут быть прямыми ссылками: в приведенном выше случае, например, аспект "`Default_Iterator`" определен до объявления `Iterate` (и мы не могли объявить его первым в любом случае, так как функция `Iterate` должна знать о `T_List`).

Чтобы понять аспекты, давайте посмотрим, как компилятор расширяет обобщенные итераторы.

Этот цикл:

```
for C in List.Iterate loop     -- C is a cursor
   declare
      E : T'Class := Element (C);
   begin
      ...
   end;
end loop;
```

расширен в:

```
 declare
    Iter : Forward_Iterator'Class :=
       List.Iterate;           -- Default_Iterator aspect
    C : Cursor := Iter.First;  -- Primitive operation of iterator
 begin
    while Has_Element (C) loop -- From Iterator_Interfaces instance
       declare
          E : T'Class := List.Element_Value (C);
                                -- Constant_Indexing aspect
       begin
          ...
       end;
       C := Iter.Next (C);    -- Primitive operation of iterator
    end loop;
 end;
```

Подпрограмма `Iterate`, на которую ссылается аспект `default_iterator`, создает и возвращает новый итератор. В общем случае он также будет содержать ссылку на сам контейнер, чтобы обеспечить срок службы контейнера, по крайней мере, до тех пор, пока итератор.

Затем итератор используется для получения курсора и управления им. Получение элемента из курсора осуществляется с помощью функции, определенной в аспекте `constant_indexing`. (Аналогичный аспект "Variable_Indexing" используется, когда циклу нужно записать элемент, но мы не будем демонстрировать это здесь.)

Функция `Element_Value` записывается здесь в простейшем виде: она напрямую возвращает копию элемента, содержащегося в структуре данных. Мы могли бы вместо этого вернуть ссылочный тип, как описано в части 1 Gem, чтобы избежать копий элементов. (Обратите внимание, что в случае `Variable_Indexing`, тип результата функции должен быть ссылочным типом.)

Аналогично расширяются итераторы элементов контейнера. Разница лишь в том, что курсор `C` не виден.

Для фактической реализации `Iterate` и `Element_Value` рекомендуется рассмотреть реализацию стандартных контейнеров, таких как `Doubly_linked_Lists`. Все контейнеры Ada 2005 были улучшены для поддержки итераторов, и они предоставляют различные примеры кода, который можно повторно использовать для собственных приложений.

Наконец, давайте рассмотрим шаблон кода, который может быть полезен. Тестовый случай выглядит следующим образом: мы реализовали сложную структуру данных, содержащую элементы типа `T'Class`. Когда мы используем итераторы элемента контейнера, `E`, таким образом, имеет тип `T'Class`, который мы можем выразить следующим синтаксисом:

```
for E : T'Class of List loop
   ...
end loop;
```

Теперь давайте рассмотрим тип `TChild`, который расширяет `T`. Мы все еще можем хранить элементы типа `TChild` в структуре данных, но затем нам нужны явные преобразования в цикле выше, чтобы привести `E` к `Tchild'Class`. Мы хотели бы минимизировать объем кода, необходимого для создания контейнера, содержащего элементы `Childclass`. Например:

```
type TChild_List is <see full type below>;

Child_List : TChild_List;

for E of Child_List loop
   -- E is of type TChild'Class, so no conversion is needed.
end loop;
```

Конечно, одна возможность состоит в том, чтобы сделать наш контейнер универсальным и создать его экземпляр один раз для `T'Class`, один раз для `TChild'Class` и так далее. Это, конечно, минимальный объем исходного кода Ada, но он все еще может представлять значительный объем скомпилированного кода и увеличит размер конечного исполняемого файла. Фактически, мы можем просто отразить иерархию `T / TChild` в самих контейнерах и переопределить только минимальное количество аспектов для достижения цели.

```
type TChild_List is new T_List with null record
   with Constant_Indexing => Child_Value,
        Default_Iterator  => Iterate,  -- inherited from T_List
        Iterator_Element  => TChild'Class;
```

```
function Child_Value (Self : TChild_List; Pos : Cursor'Class);
   return TChild'Class is
begin
   return TChild'Class (Element_Value (Self, Pos));
end Child_Value;
```

Количество дополнительного кода минимально (всего одна дополнительная функция, которая, скорее всего, будет встроена), и теперь мы можем написать цикл элемента контейнера без необходимости преобразования. Поскольку сами контейнеры теперь организованы как иерархия, у нас могут быть подпрограммы, работающие с `T_List`, которые также работают с `TChild_List` (обычное повторное использование объектно-ориентированного кода).

Однако новая структура не идеальна. Одно предостережение состоит в том, что можно вставить объект типа `T` в `TChild_List` (потому что список содержит элементы `T'Class`). Следствием этого является то, что итератор вызовет `Constraint_Error` в неявном вызове `Child_Value` в развернутом коде.

Мы надеемся, что этот драгоценный камень помог объяснить некоторые из "магии" итераторов и контейнеров Ada 2012 и позволит вам более эффективно использовать их в своем собственном коде. Даже если они требуют довольно много шаблонного кода, написанного один раз для контейнера, они определенно делают код в клиентах контейнера более легким для чтения и понимания.

Последнее замечание: примеры в этом Gem требуют довольно недавней версии компилятора, которая включает в себя ряд корректировок для отражения последних разъяснений в правилах Ada 2012.

Связанный со статьёй текст программы

Файлы примеров Ada Gems распространяются AdaCore и могут быть использованы или изменены для любых целей без ограничений.

Об авторе

Автор: Emmanuel Briot

AdaCore

Эммануэль Брио работает в AdaCore с 1998 года. Он принимал участие в различных проектах, в частности, ориентированных на графические пользовательские интерфейсы, включая GtkAda, GPS, XML / Ada, GnatTracker и нашу внутреннюю CRM. Он получил степень инженера в Национальной школе телекоммуникаций - Брест, Франция (Ecole Nationale des Telecommunications - Brest, France).

Last Updated: 10/13/2017
Posted on: 6/25/2012

Обсуждение…