Gem #97: Подсчет ссылок в Ада - Часть 1

Автор: Emmanuel Briot, AdaCore

Краткое содержание: В данной серии из трёх публикаций будет описано возможная реализация автоматического управления памятью с помощью подсчёта ссылок. В части 1 объясняется, как применять контролируемый тип для реализации автоматического подсчёта ссылок, а также описываются некоторые аспекты правильной обработки подсчёта. В части 2 будут проанализированы некоторые проблемы, связанные с многозадачностью. В части 3 будет описана реализация подсчёта слабых ссылок.

Давайте начнём…

Управление памятью, как правило, является сложной проблемой для адресации при создании приложения, и тем более при создании библиотеки, которая будет повторно использоваться сторонними приложениями. Необходимо документировать, какая часть кода выделяет память, а какая часть должна освобождать эту память. Как мы видели в предыдущем Gem, существует ряд инструментов для обнаружения утечек памяти (gnatmem, GNATCOLL.Memory или valgrind). Но, конечно, было бы удобнее, если бы память автоматически управлялась.

Некоторые языки включают автоматический сборщик мусора. Справочное руководство Ada (Ada Reference Manual) имеет разрешение на реализацию этого алгоритма, позволяющее предоставить его, хотя ни один из общедоступных компиляторов не делает этого. Дизайн Ada позволяет реализациям использовать стек во многих ситуациях, когда другие языки используют кучу; это уменьшает потребность в сборщике мусора.

Альтернативной реализацией для автоматического управления памятью является использование подсчёта ссылок: каждый раз, когда выделяется какой-либо объект, с ним связан счётчик. Этот счётчик записывает, сколько ссылок на этот объект существует. Когда этот счётчик опускается до нуля, это означает, что объект больше не ссылается в приложении и поэтому может быть безопасно освобождён.

Остальная часть этого Gem покажет, как реализовать такой механизм в Ada. Как мы увидим, существует ряд незначительных, но деликатных проблем, поэтому внедрение таких типов не так тривиально, как кажется на первый взгляд. Коллекция GNAT Components (GNATcoll) теперь включает в себя многоразовый общий пакет, который упрощает это, и мы кратко обсудим это в конце этого Gem.

Как указано выше, нам нужно связать счётчик с объектами всех типов, которые мы хотим контролировать. Самое простое решение - создать иерархию с тегами, где корневой тип определяет счётчик:

```
   type Refcounted is abstract tagged private;
   procedure Free (Self : in out Refcounted) is null;

private

   type Refcounted is abstract tagged record
      Refcount : Integer := 0;
   end record;
```

Этот подход в основном подходит для создания библиотеки повторного использования для типов с подсчётом ссылок, таких как `GNATcoll`. Если вы просто хотите сделать это один или два раза в своём приложении, вы можете просто добавить новое поле `Refcount` в свой тип записи (который не нужно помечать).

Затем нам нужно определить, когда нужно увеличивать и уменьшать этот счётчик. В некоторых языках этот счётчик должен быть изменён вручную приложением всякий раз, когда создаётся новая ссылка или когда она уничтожается. Это, например, то, как написан интерпретатор Python (в C). Но мы можем добиться большего успеха в Ada, воспользовавшись контролируемыми типами. Компилятор вызывает специальные примитивные операции каждый раз, когда значение такого типа создаётся, копируется или уничтожается.

Если мы обёртываем компонент простого типа доступа в тип, полученный из `Ada.Finalization.Controlled`, мы можем затем заставить компилятор автоматически увеличивать или уменьшать счётчик ссылок назначенного объекта каждый раз, когда ссылка будет установлена или удалена. Таким образом, мы создаём умный указатель: указатель, который управляет жизненным циклом блока памяти, на который он указывает.

```
   type Refcounted_Access is access all Refcounted'Class;
   type Ref is tagged private;

   procedure Set (Self : in out Ref; Data : Refcounted'Class);
   function Get (Self : Ref) return Refcounted_Access;
   procedure Finalize (P : in out Ref);
   procedure Adjust   (P : in out Ref);

private

   type Ref is new Ada.Finalization.Controlled with record
       Data : Refcounted_Access;
   end record;
```

Давайте сначала посмотрим, как пользователь будет использовать этот тип. Обратите внимание: Get возвращает доступ к данным. Это может быть опасно, поскольку вызывающий может захотеть освободить данные (которые должны оставаться под контролем Ref). На практике выигрыш в эффективности стоит того, так как он избегает создания копии объекта `Refcounted'Class`. Это также важно, если мы хотим, чтобы пользователь мог легко изменить назначенный объект. Пользователь, в конечном счёте несёт ответственность за то, чтобы срок службы возвращаемого значения был совместим со временем жизни соответствующего умного указателя.

```
declare
  type My_Data is new Refcounted with record
     Field1 : ...;
  end record;

  R1 : Ref;

begin
  Set (R1, My_Data'(Refcounted with Field1 => ...));
  --  R1 holds a reference to the data

  declare
     R2 : Ref;
  begin
     R2 := R1;
     --  R2 also holds a reference to the data (thus 2 references)

     ...
     --  We now exit the block. R2 is finalized, thus only 1 ref left

  end;

  Put_Line (My_Data (Get (R1).all).Field1);
```

```
  --  In practice, the smart pointers would be implemented in a generic
package,
  --  and Get would return an access to My_Data, so we could write the simpler:
  --
  --     Put_Line (Get (R1).Field1);

  --  We now leave R1's scope, thus refcount is 0, and the data is freed.
end;
```

Теперь давайте рассмотрим детали реализации. Сначала рассмотрим две подпрограммы для установки и получения назначенного объекта. Обратите внимание, что значение по умолчанию для счётчика ссылок равно нулю в файле `Refcounted`. Реализация `Set` немного сложна: ему нужно уменьшить счётчик ссылок ранее назначенного объекта и увеличить счётчик ссылок для новых данных. Вместо того, чтобы явно называть «Отрегулировать и завершать – Adjust and Finalize» (что не рекомендуется, когда этого можно избежать), мы используем агрегат и позволяем компилятору генерировать вызовы для нас.

```
procedure Set (Self : in out Ref; Data : Refcounted'Class) is
  D : constant Refcounted_Access := new Refcounted'Class'(Data);
begin
  if Self.Data /= null then
      Finalize (Self); -- decrement old reference count
  end if;

  Self.Data := D;
  Adjust (Self);  -- increment reference count (set to 1)
end Set;

function Get (P : Ref) return Refcounted_Access is
begin
   return P.Data;
end Get;
```

В `GNATCOLL.Refcount` мы предоставляем версию `Set`, которая получает существующий доступ к `Refcount'Class`, и берет на себя ответственность за освобождение, когда она больше не нужна. Реализация очень похожа на вышесказанное (хотя нам нужно быть осторожным, чтобы мы не завершали старые данные, если они совпадают с новыми, поскольку в противном случае мы могли бы освободить память).

Настройка вызывается каждый раз, когда создаётся новая ссылка. Здесь ничего особенного:

```
overriding procedure Adjust (P : in out Ref) is
begin
   if P.Data /= null then
      P.Data.Refcount := P.Data.Refcount + 1;
   end if;
end Adjust;
```

Реализация `Finalize` несколько сложнее: справочное руководство Ada указывает, что процедура `Finalize` всегда должна быть идемпотентной. Компилятор Ada может свободно вызывать `Finalize` несколько раз на одном и том же объекте, в частности, когда происходят исключения. Это означает, что мы должны быть осторожны, чтобы не уменьшать счётчик ссылок каждый раз, когда вызывается `Finalize`, поскольку данный объект имеет только одну ссылку. Следовательно, следующая реализация:

```
overriding procedure Finalize (P : in out Ref) is
   Data : Refcounted_Access := P.Data;
begin
   --  Idempotence: the next call to Finalize will have no effect
```

```
   P.Data := null;

   if Data /= null then
      Data.Refcount := Data.Refcount - 1;
      if Data.Refcount = 0 then
         Free (Data.all);  --  Call to user-defined primitive
         Unchecked_Free (Data);
      end if;
   end if;

end Finalize;
```

Этот пример для основной реализации. Следующий Gem в этой серии обсудит вопросы безопасности задач, связанные со ссылочными типами.

Настройка вызывается каждый раз, когда создается новая ссылка. Здесь ничего особенного:

Связанный со статьёй текст программы

Attached Files отсутствуют

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

Об авторе

Сведений нет.

Last Updated: 10/13/2017
Posted on: 1/17/2011

Обсуждение…

2 ответа на " Gem #97: подсчет ссылок в Ada-Часть 1”

1. 17 января 2011 года в 8:15 вечера

Ань Во (Anh Vo) сказал:

Приятно, первый Gem в 2011 году.

Мне просто интересно, как получилось, что Natural Тип не был использован вместо Integer (Refcount : Integer := 0;), так как было известно, что 0 является самым низким значением.

2. 18 января 2011, в 11:27 am

Эммануэль Бриот (Emmanuel Briot) сказал:

В основном по привычке. Если у вас когда-либо была ошибка в вашей реализации (особенно если Вы не обратили внимания, что Finalize должен быть идемпотентным/idempotent), вы можете оказаться ниже 0. Если Вы использовали Natural, это приведет к возникновению исключения в Finalize, которое сразу же станет фатальным для вашего приложения. Использование целого числа позволяет упростить процесс отладки.