Gem #107: Предотвращение освобождения памяти в типах с подсчитанными ссылками

Автор: Ada Magica, C.K.W. Grein

Краткое содержание: Gem #107 – продолжает тему, обсуждение которой было начато в ранее опубликованном Gem. Кристоф представит несколько способов того, как можно обезопасить API типов с подсчитанными ссылками с помощью некоторых возможностей Ада 2005 .

Давайте начнём…

В Gem # 97 был представлен указатель на подсчёт ссылок, где функция `Get` возвращает доступ к данным. Это может быть опасно, поскольку код вызывающий `Get` в дальнейшем может захотеть освободить данные (которые должны оставаться под контролем ссылочного типа). В этом Gem #107 мы представляем метод предотвращения неправильного использования результата функции `Get`.

Повторим соответствующие декларации:

```
  type Refcounted is abstract tagged private;
  type Refcounted_Access is access Refcounted'Class;

  type Ref is tagged private;  -- our smart pointer

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

private

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

Функция `Get` позволяет извлекать и изменять объект доступа. Проблема с этой функцией заключается в том, что она ставит под угрозу безопасность типа указателя `Ref`, например, что вызывающий объект может скопировать объект доступа к результату и освободить объект доступа:

```
  Copy: Refcounted_Access := Get (P);
  Free (Copy);
```

где `Free` – соответствующий экземпляр `Unchecked_Deallocation`.

Чтобы вылечить ситуацию, мы больше не возвращаем прямой доступ к данным. Вместо этого мы определяем метод доступа в виде ограниченного типа с таким доступом как дискриминант, и пусть `Get` возвращает объект такого типа:

```
  type Accessor (Data: access Refcounted'Class) is limited null record;
  function Get (Self: Ref) return Accessor;
```

Ограничение типа запрещает копирование, а доступ к дискриминантам неизменен. Дискриминатор также не может быть скопирован в переменную типа `Refcounted_Access`. В результате дискриминант может использоваться только для чтения и записи объекта, но не для освобождения. Таким образом, мы достигли нашей цели обеспечения безопасного доступа.

Теперь пользователь может объявить некоторый тип, полученный из `Refcounted`, и изменить значение объекта доступа таким образом:

```
declare
  type My_Refcount is new Refcounted with record
    I: Integer;
  end record;

  P: Ref;

begin
  Set (P, My_Refcount'(Refcounted with I => -10));
  My_Refcount (Get (P).Data.all).I := 42;
end;
```

Это преобразование представления в `My_Refcount` подвергнется проверке тега, которая преуспеет в этом примере. В целом Вы должны знать тип, с которым можно преобразовать представление, чтобы получить доступ к соответствующим компонентам. Альтернатива должна объявить универсальный пакет как следующее:

```
generic
  type T is private;
package Generic_Pointers is
  type Accessor (Data: access T) is limited private;
  type Smart_Pointer is private;
  procedure Set (Self: in out Smart_Pointer; Data: in T);
  function  Get (Self: Smart_Pointer) return Accessor;
private
  ... implementation not shown
end Generic_Pointers;
```

Создайте экземпляр с типом Integer и вместо последней строки будет:

```
Get (P).Data.all := 42;
```

Так как мы реализуем функцию `Get`? Это довольно просто в Ada 2005, используя функцию, возвращающую ограниченный агрегат. (Обратите внимание, что в Ada 95 ограниченные объекты возвращались по ссылке, тогда как в Ada 2005 ограниченные результаты функции строятся на месте.)

```
function Get (Self: Ref) return Accessor is
begin
   return Accessor'(Data => Self.Data);
end Get;
```

Увы, мы ещё не совсем в безопасности. Чтобы увидеть это, мы должны рассмотреть подробно время жизни `Accessor` объектов. В приведённом выше примере время жизни функции `Get (P)` заканчивается инструкцией, и `Accessor` завершается. То есть, он перестаёт существовать (в Ada просторечии, хозяин объекта является утверждение). Таким образом, проблемы возникшие в связи возможным доступом из стороннего кода решены, ничто не может произойти с объектом доступа (целое число в нашем примере), пока существует `Accessor`.

Теперь рассмотрим вариант выше. Представьте, что у нас есть указатель P, Количество ссылок которого равно 1, и давайте продлим время жизни `accessor`'s :

```
declare
   A: Accessor renames Get (P);
begin
   Set (P, ...);  -- allocate a new object
   My_Refcount (A.Data.all).I := 42;  -- ?
end;  -- A's lifetime ends here
```

В данном примере мастером `accessor` является блок (а есть и другие способы сделать срок службы так долго, как хочется). Теперь в блоке указателю `P` даётся новый объект для доступа. Поскольку мы сказали, что `P` был единственным указателем на старый объект, он завершён с катастрофическим эффектом: `A.Data` теперь являются потерянным указателем, предоставляющим доступ к несуществующему объекту до конца блока `declare`.

(Обратите внимание, что эта проблема также существовала в исходной реализации `GNATCOLL.Refcount`)

Чтобы исправить ситуацию, мы должны предотвратить освобождение памяти. Это предполагает увеличение количества ссылок с созданием `accessor` и уменьшение количества ссылок, когда `accessor` будет завершён снова. Самый простой способ сделать это – воспользоваться свойствами типа “smart pointer”:

```
type Accessor (Data: access Refcounted'Class) is limited record
  Hold: Ref;
end record;

function Get (Self: Ref) return Accessor is
begin
  return Accessor'(Data => Self.Data, Hold => Self);
end Get;
```

Кстати, в качестве последней заметки тип `Accessor`, вероятно, должен быть объявлен `limited private`, чтобы избежать возможности создания клиентами агрегатов (что, кстати, было бы совершенно бесполезным).

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

Сведения отсутствуют.

Last Updated: 10/13/2017
Posted on: 6/6/2011

Обсуждение…