Gem #26: Атрибут Mod

Автор: Bob Duff, AdaCore

Краткое содержание: Gem Ada #26 - С помощью T'Mod можно преобразовывать целочисленные типы со знаком в модульные целочисленные типы, применяя модульную (циклическую) арифметику.

Давайте начнем…

Ada имеет два вида целочисленного типа: со знаком и модульный:

```
type Signed_Integer is range 1..1_000_000;
type Modular is mod 2**32;
```

Операции на целых числах со знаком могут переполниться: если результат будет вне основного диапазона, произойдет прерывание будет поднят флаг `Constraint_Error`. Основной диапазон `Signed_Integer` - диапазон `Signed_Integer'Base`, который выбран компилятором, но, вероятно, будет чем-то как `-2 ** 31.. 2 ** 31-1`.

Операции с модульными целыми числами используют модульную (циклическую, побитовую) арифметику.

Например:

```
X : Modular := 1;
X := - X;
```

Отрицание X дает "`-1`", т.е побитно изменяется значения на противоположное и приводит к `2 ** 32-1`.

Но что относительно преобразования типов со знаком до модульного? Это - работа со знаком (таким образом, она должна переполниться), или действительно ли это - модульная работа (таким образом, она должна преобразоваться)? Ответ в Ada - первый, то есть, если вы попытаетесь преобразовать, скажем, `Integer'(- 1)` в `Modular`, вы получите `Constraint_Error`:

```
I : Integer := -1;
X := Modular (I);  -  raises Constraint_Error
```

В Ada 95 единственным способом сделать это преобразование является использование Unchecked_Conversion, что несколько неудобно. Кроме того, если вы пытаетесь преобразовать в общий формальный модульный тип, как вы узнаете, какой размер знакового целочисленного типа использовать? Обратите внимание, что Unchecked_Conversion может работать некорректно, если исходный и целевой типы имеют разные размеры.

Небольшая функция добавлена в Ada 2005 решает проблему: атрибут `Mod`:

```
generic
   type Formal_Modular is mod <>;
package Mod_Attribute is
   function F return Formal_Modular;
end Mod_Attribute;
```

```
package body Mod_Attribute is

   A_Signed_Integer : Integer := -1;

   function F return Formal_Modular is
   begin
      return Formal_Modular'Mod (A_Signed_Integer);
   end F;

end Mod_Attribute;
```

Атрибут `Mod` будет корректно преобразовываться из любого целочисленного типа в заданный модульный тип, используя семантику wraparound. Таким образом, `F` вернет все-бит-бит, для любого модульного типа передается `Formal_Modular`.

Связанный исходный код

Attached Files

```
Title: The Mod Attribute

Author: Bob Duff, AdaCore

Abstract:
T'Mod can be used to convert signed integers to modular integers
using modular (wraparound) arithmetic.

Ada has two kinds of integer type: signed and modular:

    type Signed_Integer is range 1..1_000_000;
    type Modular is mod 2**32;

Operations on signed integers can overflow: if the result is outside the base
range, Constraint_Error will be raised.  The base range of Signed_Integer is
the range of Signed_Integer'Base, which is chosen by the compiler, but is
likely to be something like -2**31..2**31-1.

Operations on modular integers use modular (wraparound) arithmetic.
For example:

      X : Modular := 1;

      X := - X;

Negating X gives -1, which wraps around to 2**32-1, i.e. all-one-bits.

But what about a type conversion from signed to modular?  Is that a signed
operation (so it should overflow) or is it a modular operation (so it should
wrap around)? The answer in Ada is the former -- that is, if you try to
convert, say, Integer'(-1) to Modular, you will get Constraint_Error:

      I : Integer := -1;

      X := Modular (I);  --  raises Constraint_Error

In Ada 95, the only way to do that conversion is to use Unchecked_Conversion,
which is somewhat uncomfortable.  Furthermore, if you're trying to convert
to a generic formal modular type, how do you know what size of signed
integer type to use?  Note that Unchecked_Conversion might malfunction
if the source and target types are of different sizes.
```

A small feature added to Ada 2005 solves the problem: the Mod attribute:

```
    generic
       type Formal_Modular is mod <>;
    package Mod_Attribute is

       function F return Formal_Modular;

    end Mod_Attribute;

    package body Mod_Attribute is

       A_Signed_Integer : Integer := -1;

       function F return Formal_Modular is
       begin
          return Formal_Modular'Mod (A_Signed_Integer);
       end F;

    end Mod_Attribute;
```

The Mod attribute will correctly convert from any integer type to a given modular type, using wraparound semantics. Thus, F will return the all-ones bit pattern, for whatever modular type is passed to Formal_Modular.generic

```
   type Formal_Modular is mod <>;
package Mod_Attribute is

   function F return Formal_Modular;

end Mod_Attribute;

package body Mod_Attribute is

   A_Signed_Integer : Integer := -1;

   function F return Formal_Modular is
   begin
      return Formal_Modular'Mod (A_Signed_Integer);
   end F;

end Mod_Attribute;

with Ada.Text_IO; use Ada.Text_IO;
with Mod_Attribute;
procedure Main is

   type Signed_Integer is range 1..1_000_000;
   type Modular is mod 2**32;

begin
   declare
      X : Modular := 1;
   begin
      X := - X;
      pragma Assert (X = 2**32-1);
      pragma Assert (X = 16#FFFF_FFFF#);
      Put_Line ("X =" & X'Img);  --  prints "X = 4294967295"
   end;

   declare
      I : Integer := -1;
      X : Modular;
   begin
```

```
      X := Modular (I);  --  raises Constraint_Error
      --  GNAT warns here:
      --  warning: value not in range of type "Modular" defined at line 7
      --  warning: "Constraint_Error" will be raised at run time

      Put_Line ("X =" & X'Img);  --  doesn't print anything
   exception
      when Constraint_Error =>
         Put_Line ("Constraint_Error raised.");
   end;

   declare
      type My_Modular is mod 2**16;
      package M is new Mod_Attribute (Formal_Modular => My_Modular);
   begin
      pragma Assert (M.F = 2#1111_1111_1111_1111#);
      Put_Line("M.F =" & M.F'Img);  --  prints "M.F = 65535".
   end;
end Main;
```

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

Обсуждение…