Gem #28: Изменение представления данных (Часть 2), Вопросы эффективности

Автор: Robert Dewar, AdaCore

Краткое содержание: Gem Ada #28 - Часть 2, Вопросы эффективности.

Давайте начнем…

Мощная функция Ada - возможность определить точный формат данных. Это особенно важно, когда у Вас есть внешнее устройство или программа, которая требует очень определенного формата.

На прошлой неделе в Gem 27 мы обсудили использование производных типов и представлений для достижения автоматической смены представления. Точнее, эта функция не полностью автоматическая, так как она требует, чтобы вы писали явное преобразование. На самом деле здесь соблюдается принцип языка Ada, в котором говорится, что изменение представления никогда не должно происходить неявно за спиной программиста без такого явного запроса посредством преобразования типа.

Причина в том, что изменение представления операции может быть очень дорогостоящим, так как в целом может потребоваться компонентное копирование, изменение представления на каждом комопоненте.

Давайте рассмотрим используя -gnatG для генерации расширенного кода, чтобы увидеть, что скрывается под этим кодом. Например, преобразование App (Input_Data) из примера прошлой неделе генерирует следующий расширенный код:

```
B26b : declare
   [subtype p__TarrD1 is integer range 1 .. 16]
   R25b : p__TarrD1 := 1;
begin
   for L24b in 1 .. 16 loop
      [subtype p__arr___XP3 is
        system__unsigned_types__long_long_unsigned range 0 ..
        16#FFFF_FFFF_FFFF#]
      work_data := p__arr___XP3!((work_data and not shift_left!(
        16#7#, 3 * (integer(L24b - 1)))) or shift_left!(p__arr___XP3!
        (input_data (R25b)), 3 * (integer(L24b - 1))));
      R25b := p__TarrD1'succ(R25b);
   end loop;
end B26b;
```

Это ужасно! На самом деле один из экспертов Ada здесь думал, что это было слишком ужасно и предложил упростить его для этого gem, но мы оставили его в первоначальном виде, так что вы можете увидеть, почему это приятно, чтобы компилятор генерировать все это вещи, так что вам не придется беспокоиться об этом самостоятельно.

Учитывая, что преобразование может быть довольно неэффективным, вы не хотите конвертировать назад и вперед больше, чем вы должны, и весь подход имеет смысл, если будут делаться обширные вычисления с участием значения.

За счет преобразования объясняются два аспекта этой функции, которые не очевидны. Во-первых, почему мы требуем производных типов вместо того, чтобы просто

разрешать подтипам иметь различные представления, избегая необходимости явного преобразования?

Ответ заключается в том, что конверсии дорогостоящие, и вы не хотите, чтобы они происходили без Вашего ведома. Таким образом, если вы пишете явное преобразование, вы получите все преимущества и недостатки перечисленных выше, но вы можете быть уверены, что это никогда не произойдет, если вы явно не захотите их выполнить.

Это также объясняет ограничение, которые мы упомянули в gem 27 на прошлой недели от RM 13.1(10):

10 Для нетегового производного типа, никакие связанные с типом элементы представления не позволены, если родительский тип - является ссылочным типом, или имеет какие-либо определяемые пользователем примитивные подпрограммы.

Оказывается, что это ограничение - все о предотвращении неявных изменений представления. Давайте взглянем на то, как деривация типа работает, когда есть примитивные подпрограммы, определенные при деривации. Connsider этот пример:

```
type My_Int_1 is range 1 .. 10;

function Odd (Arg : My_Int_1) return Boolean;

type My_Int_2 is new My_Int_1;
```

Теперь, когда мы делаем деривацию типа, мы наследуем функцию Odd для My_Int_2. Но откуда берется эта функция? Мы не написали эту функцию явно, поэтому компилятор каким-то образом материализует эту новую неявную функцию. Как компилятор это делает?

Можно подумать, что создается новая функция, включающая тело, в котором My_Int_2 заменяет My_Int_1, но это было бы непрактично и дорого. Фактический механизм позволяет избежать необходимости делать это с помощью неявных преобразований типов. Предположим, после вышеуказанных деклараций мы напишем:

```
Var : My_Int_2;
...
if Odd (Var) then
   …
```

Компилятор переводит это как:

```
Var : My_Int_2;
...
if Odd (My_Int_1 (Var)) then
   …
```

Это неявное преобразование - хороший трюк, это означает, что мы можем получить эффект наследования новой операции, не создавая ее. Кроме того, в таком случае преобразование типа не создает кода, поскольку My_Int_1 и My_Int_2 имеют одинаковое представление.

Но все дело в том, что у них может быть не то же самое представление, если у одного из них есть предложение rep, которое отличает представления, и в этом случае неявное преобразование, вставленное компилятором, может оказаться дорогостоящим,

возможно, генерируя мусор, который мы процитировали выше для случая arr. Поскольку мы никогда не хотим, чтобы это произошло неявно, существует правило, чтобы предотвратить это.

Этим соображением также руководствуется запрет на использование ссылочных типов (включая все помеченные типы). Если представления одинаковы, то можно пройти по ссылке, даже при наличии преобразования, но если бы произошла смена представления, это вызвало бы копию, которая нарушила бы требование к ссылке.

Таким образом, чтобы суммировать эти два gems #27 - #28, с одной стороны, Ada дает вам очень удобный способ, чтобы вызвать эти сложные преобразования между различными представлениями. С другой стороны, Ada гарантирует, что вы никогда не получите эти потенциально дорогостоящие преобразования до тех пор, пока Вы явно не запросите их.


Robert Dewar (1945-2015)
http://www.adacore.com/press/adacore-president-robert-dewar-1945-2015/
AdaCore

Д-р Роберт Дьюар являлся соучредителем, президентом и генеральным директором AdaCore и заслуженным профессором компьютерных наук в Нью-Йоркском университете. Сфокусировавшись на разработке и внедрении языка программирования, д-р Дьюар был основным вкладчиком в Ada на протяжении всей своей эволюции и являлся главным разработчиком технологии AdaCore GNAT Ada. Он совместно разработал компиляторы для SPITBOL (SNOBOL), Realia COBOL для ПК (в настоящее время продается Computer Associates) и Alsys Ada, а также написал несколько операционных систем реального времени для Honeywell Inc. Д-р Дьюар поставлял документы и презентации по вопросам языка программирования и сертификации по безопасности, а также как эксперт по компьютерам и законодательству, его часто приглашали на конференции, чтобы поговорить о программном обеспечении с открытым исходным кодом, вопросах лицензирования и смежных вопросах.