Gem #133: Ошибочное выполнение - Часть 2

Автор: Bob Duff—AdaCore

Краткое содержание: В предыдущем Gem было пояснено, что «ошибочное выполнение», согласно Справочному руководству Ada, означает, что может произойти что угодно, в частности, программа может работать правильно. В данном Gem продолжается обсуждение данной тематики.

Давайте начнём…

Мораль истории была: не пишите ошибочные программы.

Строго говоря, это - неправильное использование. "Ошибочный" относится к конкретному осуществлению программы, не к самой программе. Возможно, создать программу, которая имеет ошибочное поведение для некоторых входных данных, но для некоторых других входных данных будет правильно работать. Тем не менее, разумно использовать проверку во время исполнения на "ошибочную программу", которая могла бы иметь ошибочное поведение. Просто помните, что "ошибочный" не свойство текста программы, а свойство текста программы плюс его вход, и даже его синхронизация со временем исполнения.

Никогда сознательно не пишите код, который может вызвать ошибочное выполнение. Например, я видел, что люди подавляют `Overflow_Checks`, потому что они "знают", что аппаратные средства действительно переносят бит, например в модульной арифметике, и это - то, что они хотят. Это неправильно рассуждает. RM не говорит, что переполнение, когда подавлено, сделает то, что делают аппаратные средства. RM говорит, что что-либо вообще может произойти.

При подавлении `Overflow_Checks` Вы говорите компилятору принимать, чтобы предположить, что переполнения не произойдет, и не более того. Если переполнение может произойти, Вы говорите компилятору принимать ложность. В любой математической системе, если Вы принимаете "ложь", все что угодно вообще может быть доказано верным, заставив целый карточный домик упасть. Оптимизаторы кода в компиляторе могут и действительно доказывать все виды удивительных вещей, когда им разрешили принятия "лжи".

Никогда не пытайтесь предположить, что оптимизатор не достаточно умен для доставления неприятностей. Оптимизаторы кода в компиляторе являются такими сложными, что даже их авторы не могут точно предсказать то, что они сделают. Например:

```
X : Natural := ...;
Y : Integer := X + 1;

if Y > 0 then
    Put_Line (Y'Img);
end if;
```

Выше будет выведено некоторое положительное число, если `X` не является `Integer'Last`, и в случае если `X` равен `Integer'Last` будет поднято прерывание `Constraint_Error`. Поэтому оптимизатору разрешено вывести, что когда мы доберемся до "`if`", `Y` должен быть положительным, поэтому он может удалить "`if`", преобразуя код программы в:

```
X : Natural := ...;
Y : Integer := X + 1;

Put_Line (Y'Img);
```

Это хорошо: удаление "`if`", программа вероятно, будет работать быстрее, что является целью оптимизатора. Но если проверки подавлены, оптимизатор все равно может выполнить вышеуказанное преобразование. Теперь рассуждение таково: "когда мы добираемся до "`if`", либо `Y` должен быть положительным, либо мы должны получить прерывание при ошибочном выполнении". Если первое, ТО "`if`" можно удалить, потому что это правда. Если последнее, что угодно может случиться (это ошибочно!), поэтому "`if`" тоже можно удалить и в этом случае. "`X + 1`" может привести к `-2**31` (или не может).

В итоге мы получаем программу, которая гласит::

```
if Y > 0 then
    Put_Line (Y'Img);
end if;
```

и печатает отрицательное число, что удивительно.

Другим возможным поведением вышеуказанного кода (с подавленными проверками) является повышение `Constraint_Error`. - Эй, я же просил, чтобы чеки были аннулированы. Почему компилятор не подавил их?- Ну, если казнь ошибочна, то может случиться все что угодно, а исключение - это одно из возможных "что угодно". Цель подавления проверок - сделать программу быстрее. Не используйте `pragma Suppress` для подавления проверок (в смысле полагаться на не получение исключения). Компиляторы не удаляют подавленные проверки, если они свободны - например, представьте процессор ЭВМ, который автоматически ловит переполнение на аппаратном уровне.

Возможно, `pragma Suppress` должен был называться чем-то вроде `Assume_Checks_Will_Not_Fail`, поскольку он (обязательно) не подавляет проверки.

Продолжение в части 3

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: Bob Duff

AdaCore

Роберт (Боб) Андерсон Дафф - дизайнер языков программирования. Написал основные части Ada Language Reference Manual. При его участии разработаны и другие языки. Изучил десятки языков программирования.

Разработал, внедрил и/или внес большой вклад в десять компиляторов, три инструмента статического анализа и различные программные системы от реального времени/встроенные до параллельных и распределенных систем.

Эксперт в ведении программных проектов для своевременного их завершения.

Опыт работы в AdaCore:

Senior Software Engineer
AdaCore
2005 – настоящее время 14 лет
New York and Massachusetts
Ранее:

SofCheck, Inc,
Intermetrics,
Oak Tree Software, Inc.

Образование:

Carnegie Mellon University
Bachelor of Science (BS), Applied Mathematics with concentration in Computer Science, 3.6/4.0.
1982
Деятельность и сообщества: Explorer's Club

Last Updated: 10/13/2017
Posted on: 9/10/2012

Обсуждение…