Gem #50: Разрешение совмещения

Автор: Bob Duff, AdaCore

Краткое содержание: Gem Ada #50 - В данном Gem обсуждаются некоторые аспекты разработки языка, связанные с разрешением совмещения.

Давайте начнем…

Ada допускает перегрузку подпрограмм, что означает, что два или более объявления подпрограмм с одинаковыми именами могут быть видны в одной и той же области видимости. Здесь "имя" может относиться к символам оператора, например "+". Ada также позволяет перегружать различные другие обозначения, такие как литералы и агрегаты.

В большинстве языков, поддерживающих перегрузку, разрешение перегрузки выполняется "снизу вверх" - то есть информационные потоки от внутренних конструкций к внешним конструкциям. (Как обычно, компьютерные люди рисуют свои деревья вверх ногами, с корнем наверху.) Например, если у нас есть две процедуры печати:

```
procedure Print (S : Sequence);
procedure Print (S : Set);
X : Sequence;
...
Print (X);
```

Тип X определяет, какая печать подразумевается в вызове.

Ada необычен тем, что он также поддерживает разрешение перегрузки сверху вниз:

```
function Empty return Sequence;
procedure Print_Sequence (S : Sequence);
function Empty return Set;
procedure Print_Set (S : Set);
...
Print_Sequence (Empty);
```

Тип формального параметра `S` в процедуре `Print_Sequence` определяет, которая функция `Empty` предназначается в вызове. В C++, например, решил бы эквивалент "`Print (X)`" пример, но "`Print_Sequence (Empty)`", будет недопустим, потому что C++ не использует нисходящую информацию.

Если мы перегружаем вещи слишком в большой степени, мы можем вызвать неоднозначности:

```
function Empty return Sequence;
procedure Print (S : Sequence);
function Empty return Set;
procedure Print (S : Set);
...
Print (Empty); -- Illegal!
```

Вызов неоднозначен, и поэтому незаконен, потому что есть два возможных значения. Один из способов устранить двусмысленность - использовать квалифицированное выражение, чтобы сказать, какой тип мы имеем в виду:

```
Print (Sequence'(Empty));
```

Обратите внимание, что мы теперь используем как снизу вверх, так и сверху вниз разрешение перегрузки: Sequence' определяет, какой Empty выбран (сверху вниз), и какой Print выбирается (снизу вверх). Вы можете квалифицировать выражение, даже если оно не является двусмысленным в

соответствии с Правилами языка Ada - вы можете уточнить Тип, потому что это может быть неоднозначным для читателей.

Конечно, вместо этого можно разрешить пример "`Print (Empty)`", изменив исходный код, чтобы имена были уникальными, как в предыдущих примерах. Что может быть лучшим решением, если вы можете изменить соответствующие источники. Слишком большая перегрузка может сбить с толку. Определить что является "слишком много" - это отчасти вопрос вкуса.

Язык Ada действительно должен иметь разрешение перегрузки сверху вниз, чтобы разрешить литералы. В некоторых языках вы можете определить тип литерала, посмотрев на него, например, добавление "`L`"(буква el) означает "Тип этого литерала - `long int`". Такого рода согласованное обозначение не будет работать в Ada, потому что у нас есть открытый набор целочисленных типов:

```
type Apple_Count is range 0..100;
procedure Peel (Count : Apple_Count);
...
Peel (20);
```

Вы не можете сказать, посмотрев буквально на `20`, что это за тип. Тип формального параметра `Count` говорит нам, что 20 является `Apple_Count`, в отличие от какого-либо другого типа, такого как `Standard.Long_Integer`. (Технически, тип `20` является типом `universal_integer`, который неявно преобразован в `Apple_Count` - это действительно тип результата этого неявного преобразования, которое является проблемой. Но это неясный момент в языке Ada и вы не ошибётесь, если вы считаете, что целочисленная буквальная нотация перегружена всеми целыми типами.)

Программисты иногда удивляются, почему компилятор не может решить что-то, что кажется очевидным. Например:

```
type Apple_Count is range 0..100;
procedure Slice (Count : Apple_Count);
type Orange_Count is range 0..10_000;
procedure Slice (Count : Orange_Count);
...
Slice (Count => 10_000); -- Illegal!
```

Этот призыв является двусмысленным, и поэтому незаконным. Но почему? Очевидно, что программист должен иметь в виду `Orange_Count` один, потому что `10_000` вне диапазона для `Apple_Count`. И все соответствующие выражения оказываются статичными.

Ну, хорошее эмпирическое правило в дизайне языка (для языков с перегрузкой) заключается в том, что правила разрешения перегрузки не должны быть "слишком умными". Мы хотим, чтобы этот пример был незаконным, чтобы избежать путаницы со стороны программистов, читающих код. Как обычно, квалифицированное выражение исправляет его:

```
Slice (Count => Orange_Count'(10_000));
```

Другим примером, подобным литералу, является агрегат. Язык программирования Ada использует простое правило: Тип агрегата определяется сверху вниз (т. е. из контекста, в котором появляется агрегат). Восходящая информация не используется; то есть компилятор не заглядывает внутрь агрегата для определения его типа.

```
type Complex is
    record
        Re, Im : Float;
    end record;
procedure Grind (X : Complex);
procedure Grind (X : String);
...
Grind (X => (Re => 1.0, Im => 1.0)); -- Illegal!
```

Существует две процедуры `Grind`, поэтому тип агрегата может быть `Complex` или `String`, поэтому он является неоднозначным и, следовательно, является незаконным. Компилятору не требуется замечать, что существует только один тип с компонентами `Re` и `Im`, определённого подходящего типа - на самом деле компилятор не должен быть всё знающим для целей перегрузки.

Мы можем квалифицироваться как обычно:

```
Grind (X => Complex'(Re => 1.0, Im => 1.0));
```

Только после выяснения того, что Тип агрегата `Complex`, компилятор может заглянуть внутрь и убедиться, что `Re` и `Im` имеют смысл.

Это не слишком умное правило для агрегатов помогает предотвратить путаницу со стороны программистов, читающих код. Это также упрощает компилятор и делает алгоритм разрешения перегрузки достаточно эффективным.

Насколько умён чтобы стать "слишком умён" - это отчасти вопрос вкуса. На самом деле, я бы сделал правила Ada немного менее умными, если бы я переделывал их с нуля. Если мы заменили процедуру `Grind` для типа `String` на:

```
procedure Grind (X : Integer);
```

в этом случае приведённый выше вызов разрешится, потому что компилятор использует тот факт, что агрегат должен быть своего рода типом агрегата подобным, таким как запись или массив. Я бы предпочёл, чтобы вызов все ещё был двусмысленным в этом случае, но по большому счёту, язык программирования Ada получает правила почти правильно и поэтому то, что смущающее неоднозначно для людей, обычно неоднозначно по правилам Ada.

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: 10/27/2008

Обсуждение…