Ada Magica

Christoph Grein

Einladung bei Lady Ada

Gefragt war, wie Lady Ada es fertigbrachte, mit den beiden markierten Deklarationen das schmackhafte Menü ungenießbar zu machen. (In Anwesenheit der `Getränke` weist der Compiler das Programm wegen Zweideutigkeit des Katenierungsoperators `"&"` zurück, bei Abwesenheit ist es legal). Noch dazu werden die `Getränke` nicht einmal angerührt.

```
procedure Tisch (Speisekarte: in String) is

   Gänge: constant := 10;
   subtype Menü is String (1 .. Gänge);

   subtype Wein is Menü;                    -- *
   Getränke: array (1 .. Gänge) of Wein;    -- *

   procedure Genieße (Wahl: in Menü) is separate;

begin

   Genieße
    (Menü (Speisekarte & (Speisekarte'Length + 1 .. Gänge => ' ')));

end Tisch;
```

Dies ist ein Rätsel, das in *Ada aktuell* Band 2 Heft 1 (April 1994) in etwas anderer Form schon einmal erschienen ist, dessen Lösung jedoch wegen des Hinscheidens der Zeitschrift nie veröffentlicht wurde. Das wird hiermit nachgeholt.

Bevor wir jedoch die Lösung angehen, wollen wir unserer Lady erst einmal genau auf die Finger sehen. Unsere Speisekarte enthält eine beliebige Anzahl von Speisen. Wir können uns daraus ein Menü mit maximal 10 Gängen zusammenstellen. Wählen wir genau 10 aus, so ist das Aggregat

```
(Speisekarte'Length + 1 .. Gänge => ' ')));
```

leer; ansonsten füllt es die Speisefolge auf genau 10 Gänge auf, indem es die erforderliche Anzahl leerer Gänge anhängt. Wozu dient jedoch die Typumwandlung im Aufruf von `Genieße`? Nun, wir dürfen nicht davon ausgehen, dass die `Speisekarte` mit dem Eintrag Nummer 1 beginnt – deshalb verwenden wir auch das Attribut `'Length` anstelle von `'Last`. Damit erhält der Ausdruck

```
Speisekarte & (Speisekarte'Length + 1 .. Gänge => ' ')
```

die Grenzen `Speisekarte'First` und `Speisekarte'First + 9`, RM_83 4.5.3 (4). Verlangt ist jedoch der Bereich 1..10, und die Untertypumwandlung `Menü(X)` erreicht genau das unter der Voraussetzung, dass `X` die richtige Länge besitzt: Der Ausdruck `X` behält seinen Wert, nur seine Grenzen werden verschoben.

Ich sehe jetzt Lady Ada in ihrem neuen Haus sitzen und sich ins Fäustchen lachen: In Ada 95 wäre das nicht passiert, denn dort wird in einem solchen Fall die Untertypumwandlung (*sliding*) automatisch durchgeführt, RM_95 4.6 (60), wir können also direkt

```
Genieße
  (Speisekarte & (Speisekarte'Length + 1 .. Gänge => ' '));
```

schreiben, ohne `Constraint_Error` auszulösen, und damit ist dann auch plötzlich die Zweideutigkeit verschwunden.

Warum? Wir haben es hier mit einem recht generellen Problem zu tun, das in vielen Verkleidungen erscheint. Gewöhnlich tritt es auf, wenn verschiedene String-Typen verwendet werden:

```
type My_String is new String;
     My_String ('X' & 'Y')
```

doch liegt die Ursache nicht im Komponententyp (hier `Character`). Jedem, der auf ein derartiges Problem stößt, ist schier zum Verzweifeln zu Mute, besonders da die Compilermeldungen gewöhnlich nicht sehr hilfreich sind und die schuldige Deklaration (der Art von `Getränke`) bei Verwendung von **use**-Klauseln in einem ganz anderen Paket verborgen sein kann. Hier hätte ich meinen Kopf wetten mögen, dass es sich um einen Compilerfehler handelt. Was mich nur stutzig machte, war, dass drei Compiler, denen ich das vorwarf, alle eine mehr oder weniger verständliche Meldung ausgaben des Inhalts, dass der Konkatenierungsoperator `"&"` zweideutig sei.

Eine lustige Kurzfassung des Problems lautet:
Sind die Ausdrücke `"Lady" & " Ada"` *und* `"Lady Ada"` *dasselbe?*

Ich zitiere im Folgenden die Lösung von Herrn Lars Prins von der Firma Rational, angepasst an unseren `Tisch`.

Es scheint, dass die Deklaration der `Getränke` irrelevant ist, aber tatsächlich ist sie es nicht! Nach RM_83 3.6 (10) ist die Deklaration der `Getränke` äquivalent zu

```
procedure Tisch (Speisekarte: in String) is

  Gänge: constant := 10;
  subtype Menü is String (1 .. Gänge);

  subtype Wein is Menü;
  subtype Wein_Index is Positive range 1 .. 10;
  type Weinkeller is array (Wein_Index range <>) of Wein;
  subtype Weinkarte is Weinkeller (Wein_Index);

  Getränke: Weinkarte;

  procedure Genieße (Wahl: in Menü) is separate;

begin

  Genieße
   (Menü (Speisekarte & (Speisekarte'Length + 1 .. Gänge => ' ')));
                      ↳ zweideutig
end Tisch;
```

Die Prozedur kann vereinfacht werden, ohne dass die Zweideutigkeit verloren geht:

```
procedure Tisch (Speisekarte: in String) is

  Gänge: constant := 10;
  subtype Menü is String (1 .. Gänge);

  subtype Wein is Menü;
  subtype Wein_Index is Positive range 1 .. 10;
  type Weinkeller is array (Wein_Index range <>) of Wein;

  Ungenießbar: Menü := Menü ("Spiegelei " & "und Spinat");
                                          ↳ zweideutig
begin
  null;
end Tisch;
```

Gemäß RM 4.6 (3) muss sich der Operand einer Typumwandlung unabhängig vom Kontext bestimmen lassen (insbesondere unabhängig vom Zieltyp). Das heißt, dass für die Auflösung des `"&"`-Operators die Tatsache nicht verwendet werden darf, dass das Ergebnis zum Typ `Menü` umgewandelt werden wird.

```
procedure Tisch (Speisekarte: in String) is

  Gänge: constant := 10;
  subtype Menü is String (1 .. Gänge);

  subtype Wein is Menü;
  subtype Wein_Index is Positive range 1 .. 10;
  type Weinkeller is array (Wein_Index range <>) of Wein;

  Ungenießbar: Boolean := Boolean ("Spiegelei " & "und Spinat");
                                                ↳ zweideutig
begin
  null;
end Tisch;
```

Um zu zeigen, wie der Ausdruck interpretiert werden kann, fügen wir zwei unzweideutige Deklarationen hinzu. Die Zweideutigkeit wird in diesen Fällen durch Entfernen der Typumwandlung aufgelöst, die eine zusätzliche Anforderung an die Auflösung des `"&"`-Operators stellt.

```
procedure Tisch (Speisekarte: in String) is

  Gänge: constant := 10;
  subtype Menü is String (1 .. Gänge);

  subtype Wein is Menü;
  subtype Wein_Index is Positive range 1 .. 10;
  type Weinkeller is array (Wein_Index range <>) of Wein;

  Ungenießbar: Boolean := Boolean ("Spiegelei " & "und Spinat");
                                                ↳ zweideutig

  -- Nicht länger zweideutig sind folgende Anwendungen des
  -- "&"-Operators:

  Geniessbar:      String (1 .. 10) := ("Mosel" & "-Wein");
  Auch_Geniessbar: String (1 .. 10) := ("Mosel-Wein");
```

```
  Annehmbar     : Weinkeller (1 .. 2) := ("Mosel" & "Rhein");
  Auch_Annehmbar: Weinkeller (1 .. 2) := (1 => "Mosel",
                                          2 => "Rhein");
  -- Wieder zweideutig ist:

  Widerwärtig: Weinkeller (1 .. 2) :=
                 ((1 => "Diethylen") & (2 => "glykol"));

  -- In diesem Fall wird die Zweideutigkeit dadurch eingeführt,
  -- dass der Compiler nicht in die Aggregate hineinsehen darf um
  -- zu bestimmen, welcher "&"-Operator gemeint ist (RM 4.3 (8)).

begin
  null;
end Tisch;
```

Zitat Ende.

Alles klar? Na ja, man braucht Zeit, das zu verdauen. Im Grunde geht es darum, dass der Compiler nicht entscheiden kann, ob bei der Konkatenierung der beiden Operanden nun ein `Menü` oder ein `Weinkeller` herauskommen soll. (In beiden Fällen ist die Umwandlung in `Boolean` natürlich absolut unsinnig – aber das spielt keine Rolle, genauso wenig wie die falsche Länge der Strings!)

Beheben lässt sich das Problem in Ada 83 zum Beispiel folgendermaßen:

```
procedure Tisch (Speisekarte: in String) is

  Gänge: constant := 10;
  subtype Menü is String (1 .. Gänge);

  subtype Wein is Menü;
  Getränke: array (1 .. Gänge) of Wein;

  procedure Genieße (Wahl: in Menü) is separate;

  function Pansche (X: String) return Wein is
  begin
    return Menü (X);
  end Pansche;

begin

  Genieße
--   (Menü    (Speisekarte & (Speisekarte'Length + 1 .. Gänge => ' ')));
    (Pansche (Speisekarte & (Speisekarte'Length + 1 .. Gänge => ' ')));

end Tisch;
```

Mit der Funktion `Pansche` sieht das zwar fast genauso aus wie zuvor, es gibt jedoch einen gewichtigen syntaktischen und semantischen Unterschied: Eine Typumwandlung sieht nur aus wie ein Funktionsaufruf, ist aber keiner, es lässt sich keine Ada-Spezifikation einer Funktion für sie angeben (weswegen sie in der Syntax auch nicht als Funktionsaufruf vorkommt)! Die Spezifikation müsste nämlich die Form haben:

```
function Typ_Kennung (X: Jede_beliebige_Typ_Kennung)
           return Typ_Kennung;
```

Jetzt wird auch die Forderung klar, warum ein Compiler ohne Beachtung des Kontextes den Typ des Operanden bestimmen muss (RM 4.6 (3)).

Ähnliches gilt für einige Attribute. Gegeben seien zwei überladene Funktionen

```
function F (X: Ein_Typ) return Long_Integer;
function F (X: Ein_Typ) return Integer;
```

Wieder erhalten wir Zweideutiges:

```
I: Ganze_Zahl := Ganze_Zahl (F (X));
                              ↯ zweideutig
J: Ganze_Zahl := Ganze_Zahl'Val (F (X));
                                 ↯ zweideutig
```

Auch für das `'Val`-Attribut lässt sich keine Spezifikation angeben.

Noch eine weitere Lösung ganz ohne Typumwandlung ist möglich. Es ist die generelle Lösung, die Ada für solche Mehrdeutigkeiten anbietet:

```
Genieße
  (Menü
    (String'(Speisekarte & (Speisekarte'Length + 1 .. Gänge => ' '))));
```

Adresse des Autors: C. Grein, c/o ESG, Vogelweideplatz 9, 81677 München
eMail: Christ-Usch.Grein@T-Online.de
Christoph.Grein@esg-gmbh.de
Christoph.Grein@eurocopter.de