Unorthogonalities in the Identification Rules in Ada

Iván Bach

Computer and Automation Institute
Hungarian Academy of Sciences
1111 Budapest, XI., Kende u. 13-17.

Key words. Ada, visibility, identification, overloading.

Introduction. In Ada, the identification, i.e., the determination of the corresponding defining occurrence for each applied occurrence turns out to be one of the most delicate problems of compilation.

In the traditional block-structured languages the visibility rules were very simple and there was no overloading (except for predefined arithmetic operators but since the concept or overloading has not yet been "discovered" even the implementor was not aware of it).

In Ada there are significant changes in both fields. The concept of *package* together with the *use* clause altered visibility rules. The overloading has been extended to any subprogram including enumeration literals. In addition, the identification is further complicated by derivation, i.e., by the introduction of derived subprograms.

In general however, the complexity of the facilities does not mean necessarily the complexity of visibility rules. It is of vital importance that despite of these improvements they should remain as simple as possible and orthogonal as well. Otherwise the user will not be able to follow all the details and will not understand the concepts of the language.

It has been previously mentioned that the derivation rules seem not to be logical and their syntax is dangerous [3].

It will be explained that the visibility rules in Ada are not well defined and orthogonal. This is partly due to some concept which introduce totally different visibility and/or overloading resolution rules leading sometimes to contradictions.

The goal of this article is to focus the attention on these possible misunderstandings.

Unorthogonality in visibility. In Ada, defining occurrences of a designator can be seen from a given location of the program text due to two reasons.

Every declaration found on the unique path along the program-structure tree going from the given location up to the root of the tree is directly visible unless hidden. This kind of visiblity will be referred to as visiblity through structure.

Introducing the *package* concept, the declarations within the visible part of the package may be made directly visible through applying *use* clauses to the package in question. This kind of visibility will be called visibility through use clause.

These two kinds of visibility are disjoint. Although it is not prohibited to apply a use clause to a package within the package, i.e., where its visible part is directly visible through structure, in this case the use clause will not have any effect whatsoever.

The visibility rules of Ada treat these two kinds of visibility differently. Two major differences will be mentioned.

A new declaration may hide an entity formerly directly visible through structure but may never make directly visible a formerly hidden declaration.

Unfortunately there is no full analogy in the case of visibility through use clause. While a directly visible entity may turn out to be only potentially visible due to a new declaration it may happen that a formerly only potentially visible declaration will be "promoted" to directly visible due to the effect of a new declaration.

Example 1.

```
A : declare
     function F return INTEGER;
     package B is
      function F return BOOLEAN;
     end B;
     .
     C : declare
          use B;  -- both A.F and B.F are directly visible
          F : INTEGER;  -- neither A.F nor B.F are visible
          .
          D : declare
               function F return FLOAT;
                  -- only B.F but not A.F is directly visible
```

The example shows clearly the uneven treatment of entities visible through structure versus through use clause.

A similar situation occurs in the case of homographs.

When visible through structure at most one of the homographs can be directly seen, while when visible through use clause all of the homographs are directly visible simultaneously.

The next example demonstrate such a situation. It emphasizes the unorthogonality of the visibility rules. It is quite strange that the very same declarations behave differently depending upon the location of the applied occurrence.

Probably it is not necessary to mention how disturbing this difference in treatment can be for a user.

Example 2.

```
declare
 package A is
  procedure P (X : INTEGER);
  package B is
   procedure P (Y : INTEGER);
  .
  end B;
 end A;
 .
 package body A is
 procedure P (X : INTEGER) is
 .
 end P;
 package body B is
  procedure P (Y : INTEGER) is
  .
  end P;
  .
  P (X => 5); -- illegal since A.P is hidden by B.P
 .
 end B;
end A;
use A; use B;
.
P (X => 5); -- legal P means A.P since both homographs
            -- are directly visible and therefore
            -- overloading can be resolved
```

Mixed operations. At the very end only two mixed operators remained in the language, the * and the / defined between any two fixed point types. The disadvantages of the mixed operators have already been analized by the aouthor [2,3].

The visibility rules suppose that for each entity there is a given location in the program text where this entity is dexlared or can be regarded as being declared. For the mixed operators, according to the Manual, this location is the STANDARD package. The semantics is, however, not quite clear.

It is not stated explicitly that the universal real type is considered to be a member of the "any fixed type". If not the mixed operators can not be used for multiplying rsp. dividing with numeric literals.

It is strange that closely related operators behave quite differently.

Example 3.

```
package A is
 type FIX1 is delta 0.125 range 0.0 .. 10.0;
end A;
___________________________
with A; use A;
package B is
 X, Y : FIX1;
 type FIX2 is new FIX1;
end B;
___________________________
with B; use B;
procedure MAIN is
 Z : FIX2; I : INTEGER;
.
X := X + Y; -- illegal the "+" for type FIX1 is not visible
Z := FIX2 (X + Y); -- illegal the operator is not visible
```

```
X := X * I; -- illegal the operator "*" defined between
Z := FIX2 (X * I); -- FIX1 and INTEGER types is not visible
Z := FIX2 (X * Y); -- legal it is the mixed operator
```

According to the visibility rules it is only the last statement which is legal. Since the mixed operators are dexlared within the STANDARD package they are visible everywhere in the program, i.e., within the entire scope of any type. In addition there is no possibility to redefine them they can not be hidden, too.

In general an operator is applicable if it is visible. Only the basic operations which can be applied within the entire scope of the operation. This means that the mixed operators behave as though they belonged to basic operations.

Although there is no possibility to redefine the mixed operators, the user may define any number of * and / operators between objects of any (two) fixed type(s). This may cause surprising ambiguity problems.

Example 4.

```
declare
 type FIX is delta 0.125 range 0.0 .. 10.0;
 P, Q : FIX;
 function "*" (X, Y : FIX)   return FIX is
 .
 end "*";
 .
 .
 P := P * Q; -- legal the "*" operator is the user defined one
 P := FIX (P * Q); -- ambiguity the operator can be the
                   -- user defined one or the mixed one
```

It is very doubtful whether the gain in applying those mixed operators is worth for the strange semantics the user has to learn.

Overinterpretation. The unique interpretation of the program text is sometimes quite difficult in Ada. There are potential ambiguities which must be - if possible - resolved to achieve unique interpretation.

Potential ambiguity may be caused by overloading, i.e., when more than one defining occurrence of a designator is visible from a given location of the program. There are certain rules described in the Manual for overloading resolution. It is the context which serves as a tool for determining the actual meaning of an identifier.

However, a seemingly mild change in the language syntax may cause a new kind of ambiguity which will be referred to as "overinterpretation".

In case of overinterpretation, there is a unique defining occurrence corresponding to a given applied occurrence of an identifier but it can be interpreted more than one way. This kind of possible ambiguity is not mentioned in the Manual explicitly.

The mild change mentioned above which made this trouble "possible" is the new way of calling parameterless functions. According to the new rule no empty parentheses are needed.

In analogy with the overloading resolution there should be an overinterpretation resolution algorithm which helps the "true" interpretation of the defining occurrence at the given location be determined. Unfortunately there is no detailed description in the Manual how overinterpretation should be resolved.

Of course the context is adequate tool for resolving any kind of ambiguities. Nevertheless, the context is mentioned only in connection with overloading resolution and there is no hint in the sense, that context should be used in general as a tool for resolving any kind of ambiguities.

As a matter of fact this part of the Manual is inherited from those times, when the syntax of function call was more rigorous and there was no such phenomenon as overinterpretation.

Only for the interpretation of selection has been a new rule added to the Manual when the new syntax for function call were introduced. It says: "If there is one possible interpretation of the prefix of a selected component as the name of an enclosing function, then the only interpretations considered are those as expanded names."

This rule covers a special case of overinterpretation and determines how to resolve it.

It is against the orthogonality that according to the rule it is not the context which should be used when resolving overinterpretation as far as selections are concerned.

The next example shows how the "expended name first" rule should be applied.

Example 5.

```
type R is record
 X, Y : INTEGER;
end record;
function F return R is
 Z, Y : INTEGER;
 .
 Z := F.Y; -- means the variable Y within the
           -- function body, F.Y is an expanded name
 .
 Z := F.X; -- illegal, there is no entity X declared
           -- within the scope of the prefix
```

It should be mentioned that resolution through context would not resolve the first ambiguity but would give a reasonable interpretation to the second.

Unfortunately the above "expanded name first" principle cannot be used as a general tool for resolving overinterpretation. The next example shows such a situation.

Example 6.

```
type W is array (INTEGER range < >) of INTEGER;
Y : W (1 .. 3) := (1, 2, 3);
J : INTEGER;
function F (I : INTEGER := 5) return W is
.
end F;
procedure P (X : W) is
.
end P;
.
J := F (2);
P (Y & F (3));
```

In this example there are two interpretations possible both for F (2) and for F (3). The function F should be called in both cases but it is ambiguous whether this call should be performed with the default value or with the actual parameter 2 rsp. 3.

Using the old notation, when empty parentheses were obligatory to indicate function call, these two possibilities can be characterized as follows

```
     F ( ) (2)   or    F (2)
and  F ( ) (3)         F (3)
```

The left hand side notation indicates a function call with the default value 5 giving an array which is in turn indexed by the value 2 rsp. 3 finally resulting an INTEGER type object.

The right hand side notation means a function call with the actual parameter 2 rsp. 3 and the result will be an array of type W i.e., array of INTEGER.

What should be done?

It is more than desirable that these ambiguities should be resolved, too. By all means the natural way to do it is the resolution through the context. The mechanism for context resolutions must be incorporated in the compiler anyway - because the overloading resolution - and can be used allmost without changes in general for any ambiguity, so for overinterpretation resolution as well.

If the context is used for resolving ambiguity the F (2) in the text should be interpreted according to the left hand side.

The example has been constructed carefully in such a way that the ambiguity of F (3) can not be resolved with the help of the context.

The use of two resolution principles where one would be satisfactory makes un unnecessary burden on both the user and implementor.

Last but not least it will be shown that overloading and overinterpretation may occur simultaneously.

Example 7.

```
type V is record
 A : INTEGER;
end record;
type W record
 A : BOOLEAN;
end record
function F return V is
.
end F;
function F return W is
 A : FLOAT;
 B : BOOLEAN;
 I : INTEGER;
 .
 B := F.A;
 I := F.A;
```

Of course, this case is more interesting when the overinterpretation should be resolved through expanded name first principles, as in the example above.

Thus different method should be used for overloading rsp. overinterpretation resolution.

The result depends upon the order of resolution. If the expanded name first principle is applied first, as it is probably indicated by the Manual, only the second of the two assiguiment statements is legal. Changing the order of resolution both statements were correct.

<u>Conclusion</u>. The degree of complexity of the language rules influences the legibility and teachability of the language. The rules of Ada are unnecessarily sophisticated and it would be desirable to simplify them. With respect the identification it is disadvantegeous that the visibility through structure and the visibility through use clause is defined by different rules. The visibility of the mixed operators need a special treatment, too. The important deficiency is the lack of a uniform adequate resolution for overinterpretation. The Manual covers only partically this topic presently.

<u>Literature</u>

[1] Reference Manual for the Ada Programming Language
Alsys 1983 Januar

[2] I. Bach: Remarks on Ben Brosgol's paper
Ada UK News, Vol.3. No.1.

[3] I. Bach: On the Type Concept of Ada
ACM Ada Letters, Vol.II. No.3.