# ISO/IEC JTC1/SC22/WG9 N376
**2 August 2000**

## DRAFT Submitted for WG9 Email Ballot

# Programming languages — Ada
**RECORDS OF RESPONSE 1**
**September 2000**

Records of Response 1 for International Standard ISO/IEC 8652:1995 was prepared by AXE Consulting under contract from The MITRE Corporation.

# Programming languages — Ada

**RECORDS OF RESPONSE 1**

**September 2000**

Records of Response 1 for International Standard ISO/IEC 8652:1995 was prepared by AXE Consulting under contract from The MITRE Corporation.

# Introduction

This document contains responses to issues raised in defect reports on the Ada 95 standard [ISO/IEC 8652:1995]. This document contains responses to issues that required no wording changes to the standard. For wording changes to the standard, see Technical Corrigendum 1.

The responses are presented in the order that the issues occur in the Ada standard. For each question, a reference to the defect report that prompted the response is included in the form [8652/0000]. The defect reports have been developed by the ISO/IEC JTC 1/SC 22/WG 9 Ada Rapporteur Group to address specific questions about the Ada standard. Refer to the defect reports for additional information on the issues.

The responses contain many references of the form ss.cc(pp) or ss.cc.aa(pp). These refer to particular paragraphs in the standard, with the notation referencing the (sub)clause number in the Ada 95 standard (ss.cc.aa), followed by a parenthesized paragraph number (pp). Paragraphs are numbered by counting from the top of the (sub)clause, ignoring headings.

The responses contain references to the Annotated Ada Reference Manual (AARM). This document contains all of the text in the Ada 95 standard along with various annotations. It was prepared by the Ada 95 design team, and includes rationale for some rules.

The responses may contain references to Ada 83. Ada 83 is the common name for the previous version of the Ada standard, ISO/IEC 8652:1987. Similarly, AI83 refers to interpretations of that standard.

## Is normal termination an "external interaction"?

### Defect Report 8652/0094
### Section References 1.1.3

### Question

Is the normal termination of a program with no function result or parameters an external interaction? (Yes.) It isn't defined as one in 1.1.3(9-14).

In particular, the following main program has no external interactions as defined in 1.1.3:

```
procedure Main is
begin
   loop
      null;
   end loop;
end Main;
```

May an implementation optimize out the infinite loop? (No.) The resulting program also has no external interactions; the only difference is that it terminates.

### Response

1.1.3 (12) includes as an external interaction:

> Any result returned or exception propagated from a main subprogram (see 10.2) or an exported subprogram (see Annex B) to an external caller.

10.2 (2) says that a main subprogram "can be invoked from outside the Ada implementation". A.15 goes on to describe the possible interactions of a main subprogram with the "external execution environment", including the possibility that a main subprogram may return a status value. In particular "Normal termination of a program returns as the exit status ... ".

The above wording must be understood as an attempt to define in Ada terminology ("invoked", "result returned", and "exception propagated") interactions with an external execution environment that is not itself governed by the standard.

The International Standard does not (and cannot) define the exact manner of invocation of a main subprogram by the external execution environment, how parameters may be passed to it (if any), how the main program may return status, or what activities in the external environment may be triggered by termination of the main program (whether normal or abnormal). Therefore, unless a given implementation can determine that a specific external environment will not be affected by whether a given program terminates, the Ada implementation must assume that it might.

Even within the Ada domain, elimination of an infinite loop from a subprogram is not an acceptable optimization. For example, consider the following:

```
procedure External_Environment is
   procedure Main is
   begin
      loop
         null;
      end loop;
   end Main;
begin
   Main;
   Explode_The_Bomb;
end External_Environment;
```

No one would argue that it would be acceptable to delete the infinite loop above, assuming that Explode_The_Bomb has some "external interaction" as the name suggests. By analogy, there is no reason to presume that the external environment from which an Ada program is invoked is not capable of expressing a dependence such as the one above.

## Unconstrained formal types

### Defect Report 8652/0095
### Section References 3.2

### Question

When is a generic formal subtype unconstrained?

AARM 12.3(11.c) says:

> A formal derived subtype is constrained if and only if the ancestor subtype is constrained. A formal array type is constrained if and only if the declarations says so. Other formal subtypes are unconstrained, even though they might be constrained in an instance.

However, this does not seem to follow from the rules in the standard.

### Response

3.2(9) says:

> A subtype is called an unconstrained subtype if its type has unknown discriminants, or if its type allows range, index, or discriminant constraints, but the subtype does not impose such a constraint; otherwise, the subtype is called a constrained subtype (since it has no unconstrained characteristics).

Note the distinction between "type" and "subtype". An array type allows an index constraint, for example, whereas an array subtype may or may not impose one.

Thus, AARM 12.3(11.c) is incorrect in the case of a subtype of a formal private type with no discriminants. Such a subtype is constrained, because its type does not allow a discriminant constraint.

This raises an interesting question:

```
generic
    type Str_Ptr is access String;
package GP is
    ...
end GP;

package body GP is
    subtype Str_Ptr_10 is Str_Ptr(1..10); -- Legal?  (Yes.)
    X : Str_Ptr_10;
    ...
end GP;
```

Str_Ptr is unconstrained, since an access type designating String allows a constraint, and Str_Ptr does not impose one. This implies that 3.7.1(7) allows the subtype_indication "Str_Ptr(1..10)". However, the instance might try to impose a different constraint:

```
type Str_P is access String;
subtype Str_P_7 is Str_P(1..7);

package P is new GP(Str_P_7); -- Legal?  (Yes.)
```

12.3(11) says this is legal, since Legality Rules are not enforced in the body of an instance. Thus, the bounds of X are (1..10), and the constraint (1..7) is effectively ignored.

Note that if the declaration of Str_Ptr_10 were in the declaration of GP instead of the body, then the instantiation P would be illegal, since Legality Rules are enforced in the instance declaration.

Note that if the two constraints ever "meet", Constraint_Error will be raised, as illustrated by the following modification of the above example:

```
generic
    type Str_Ptr is access String;
    Obj: in out Str_Ptr;
package GP is
   procedure Bad;
end GP;

package body GP is
    subtype Str_Ptr_10 is Str_Ptr(1..10);
    S: aliased String(1..10);
    X: Str_Ptr_10 := S'access;

    procedure Bad is
    begin
       Obj := X; -- Constraint_Error is raised here.
    end Bad;
end GP;

type Str_P is access String;
subtype Str_P_7 is Str_P(1..7);
Actual_Obj: Str_P_7;

package P is new GP(Str_P_7, Actual_Obj);
...
P.Bad; -- This will propagate Constraint_Error.
```

In the instance, Obj is constrained to (1..7), whereas X is constrained to (1..10). An attempt to assign one to the other (as in procedure Bad) will fail the constraint check.

An alternative rule would be that it is illegal to give a constrained access subtype as the actual in the above case. This would be a cleaner rule, since it would avoid the anomaly of a doubly-constrained access type. However, most Ada 83 compilers probably allow the above example, and make the bounds of X be (1..10), and since constraining an access subtype is fairly uncommon, it does not seem worthwhile to use this alternative rule.

Note that it is irrelevant whether a formal scalar subtype is considered constrained or unconstrained, since no compile-time rules are affected. Constrained-ness of a scalar type has an effect at run-time, but at run-time, only instances exist, not generic units.

## Case sensitivity of Wide_Value and Value attributes

**Defect Report 8652/0096**
**Section References 3.5**

### Question

For S'Wide_Value in the case of a nongraphic character, 3.5(43) gives the following condition under which the normal result is returned; otherwise, Constraint_Error is raised: "...if the sequence of characters of the parameter (ignoring leading and trailing spaces) has the syntax of an enumeration literal and if it ... corresponds to the result of S'Wide_Image for a nongraphic character of the type..."

Is the use of the term "corresponds" intended to imply case sensitivity? (No.)

### Response

A sequence of characters corresponds to the result of S'Wide_Image if it is the same ignoring case. Thus, S'Wide_Value is not case sensitive in the case of a nongraphic character. The same applies to S'Value.

It is clearly the intent that the "correspondence" mentioned above be case insensitive. Thus, Character'Wide_Value("nul") does not raise Constraint_Error, even though Character'Wide_Value returns "NUL" for the nul character.

## Controlled types in language-defined generic packages

**Defect Report 8652/0097**
**Section References 3.9.1; 7.6; A**

### Question

May an implementation declare a controlled type in its implementation of a language-defined generic package? (No.) For example, may Ada.Sequential_IO.File_Type be a controlled type? (No.)

### Response

A language-defined generic package may be instantiated at any nesting depth. This follows from the fact that the standard does not say otherwise.

This implies that the implementation of a language-defined generic package cannot contain the declaration of a controlled type, since the accessibility rules require that controlled types be declared at library level.

If an implementation wishes to implement, say, Ada.Sequential_IO.File_Type in terms of a controlled type, it can declare the controlled type in a separate (non-generic) package, such as System.File_Implementation, and make type File_Type have a component whose type is the controlled type.

The implementation model given in AARM A.5.2(46.a), which calls for making the Generator type of the random number packages a controlled type, is wrong. It should instead call for Generator to have a component of a controlled type, declared in a (non-generic) package.

## Primitive operations declared before it is known if the type is tagged

### Defect Report 8652/0098
### Section References 3.9.2

### Question

Is a primitive operation of a type which is declared before it is known that the type is tagged a dispatching operation? (Yes.)

This question matters because 3.9.2(12) states that "A given subprogram shall not be a dispatching operation of two or more distinct tagged types."

Consider the following examples:

```
package P1 is
   type T1 is private;
   type T2 is private;
   procedure P (X1 : T1; X2 : T2); -- violates 3.9.2(12) ?  (Yes.)
private
   type T1 is tagged null record;
   type T2 is tagged null record;
end;

package P2 is
private
   type T1;
   type T2;
   procedure P (X1 : access T1; X2 : access T2);
      -- violates 3.9.2(12) ?  (Yes.)
end;

package body P2 is
   type T1 is tagged null record;
   type T2 is tagged null record;
end;
```

### Response

Consider a type whose partial view is untagged, but whose full view is tagged. A primitive subprogram declared for the partial view is a dispatching subprogram of the full view.

Therefore, if there are two types, T1 and T2, and there is a primitive subprogram of both:

```
procedure Primitive(X: T1; Y: T2);
```

then it is illegal for both T1 and T2 to be tagged, even if the full type declarations occur after Primitive.

## Operators not inherited from root numeric types

### Defect Report 8652/0099
### Section References 4.5.5

### Question

Consider the operators described in 4.5.5(17), such as

```
function "*" (Left: root_real; Right: root_integer) return root_real;
```

It would seem logical to assume that root_real is declared immediately within the visible part of package Standard, which means that this operator is a primitive subprogram of type root_real.

By 3.5.6(3), Float is derived from root_real. Does Float therefore inherit this operator? (No.) If so, the following function exists:

```
function "*" (Left: Float; Right: root_integer) return Float;
```

which would make the following legal:

```
declare
   X: Float;
begin
   ...
   X := X * 2; -- Legal?  (No.)
end;
```

The same applies to a user-defined numeric type.

### Response

Although floating point types are derived from root_real, this does not imply any inheritance of subprograms. Inheritance happens for a derived type that is declared by a derived_type_definition, by 3.4(7). For private extensions, 7.3(16) and 12.5.1(20) apply. No such rules apply to other kinds of type_definition; therefore, no inheritance takes place for such types.

Section 4 explicitly defines the predefined operators that are implicitly declared for a type, according to its class. The mechanism of inheritance is not used for this purpose.

Note that this is not the only way in which the implicit derivation from a root numeric type is different from derivation via an explicit derived_type_definition. See, for example, 3.5.4(14) and AARM 3.5.4(14.a).

## Inconsistency with Ada 83 in the definition of exponentiation

### Defect Report 8652/0100
### Section References 4.5.6

### Question

4.5.6(11) says:

The expression X ** N with the value of the exponent N positive is equivalent to the expression X * X * ... X (with N-1 multiplications) except that the multiplications are associated in an arbitrary order.

However, Ada 83 required left-to-right associations. Is this an upward inconsistency? (Yes.)

### Response

Ada 83 required left-to-right associations, and AI83-00137 confirmed this. AI83-00868 allowed arbitrary association, but AI83-00868 was never formally approved. Hence, this represents an upward inconsistency, and should have been so documented in the AARM.

## Conversions between access types with different representations

### Defect Report 8652/0101
### Section References 4.6

### Question

It is possible to declare types that are normally convertible, but that have a pragma Convention that makes some such conversions impossible or impractical. For instance,

```
    type Ada_Ptr is access Integer; -- Integer is C-compatible in this
                                    -- implementation.
    type C_Ptr is new Ada_Ptr;
    pragma Convention (C, C_Ptr);
```

The value sets of such convertible types may not completely overlap. For example, on a machine that is not 8-bit-byte addressable, an Ada_Ptr might be represented as a normal address, whereas a C_Ptr might be represented as a bit-field pointer of some sort.

What should happen when a value of C_Ptr has no corresponding value in Ada_Ptr?

### Response

The semantics of such conversions are implementation defined. This follows from the fact that pragma Convention is a representation pragma (B.1(29)), and implementations can interpret representation pragmas however they want and place restrictions on them (13.1(20)), except when there are explicit rules to the contrary. This is also stated in the NOTE of B.1(43). So it's perfectly acceptable to raise an exception when a given operation doesn't make sense.

This is clearly desirable, since the goal of interfacing to another language is to match the representations chosen by the implementation of the other language, and doing so can make it impossible or impractical to obey the "normal" (non-interfaced) Ada semantics.

## Visibility of inherited private components

### Defect Report 8652/0102
### Section References 7.3

### Question

Consider the following example:

```
    package P is
        type Parent is tagged private;
    private
        type Parent is
            record
                C: Integer;
            end record;
    end P;

    with P; use P;
    package Q is
        type Child is new Parent with
            record
                C: Integer;
            end record;
    end Q;

    with Q; use Q;
    package P.Child is
        type Grandchild is new Q.Child with null record;
        X: Grandchild;
        ... X.C ...
    end P.Child;
```

What is the meaning of X.C, given that Grandchild is declared in a place where the full view of Parent is visible?

### Response

There is a general design principle in Ada that you can never have more visibility into the components or operations of a type than in the package where the type is declared. Effectively, the components and operations of a type are "frozen" to be those visible somewhere within the "immediate" scope of the type. Even if you go into a package that knows more about the ancestors of the type, that doesn't change the set of components or primitives that the type has.

So in the above example, the type Child is declared in a place where there is no visibility on the C component of Parent; hence this component is not declared, and it is legal to declare another, unrelated, C component in Child. Thus, X.C refers to the C component declared in Child. This is despite the fact that at the point of "X.C", it *is* visible that Child is derived from Parent, and it *is* visible that Parent has a component called C. To refer to the C component from Parent, one would write Parent(X).C.

The C component from Parent does exist, despite the fact that it is not visible.

This paradox of knowing the ancestry of a type, but not being able to take advantage of it except through explicit conversion, also applied to derived types in Ada 83. It is based on the general notion that even inherited operations and components need to have a point of declaration, and that point of declaration is required to be in the immediate scope of the derived type. If there is no place where such implicit declarations could occur, then the corresponding operations or components are not inherited. (Of course, inherited operations get declared (if at all) in the declarative region containing the type, whereas components get declared (if at all) in the declarative region of the type itself.)

See 7.3.1 and AARM-7.3.1(7.a-7.r).

Consider the following modified example:

```
package P is
    type Parent is tagged private;
private
    type Parent is
        record
            C: Integer;
        end record;
end P;

package P.Q is
    type Child is new Parent with
        record
            C: Integer; -- Illegal!
        end record;
end P.Q;
```

The above example is illegal, because in this case Child *does* inherit C from Parent, so the second declaration of C is an illegal homograph.

## Type descriptors can be laid out at compile time

### Defect Report 8652/0103
### Section References 7.3.1

### Question

The rules in 7.3.1 imply that for a type declared in a package_declaration, certain inherited subprograms can be declared in the package_body. How does this correspond to the intended implementation model of AARM 3.9(1.a-1.b)?

> The intended implementation model is for a tag to be represented as a pointer to a statically allocated and link-time initialized type descriptor. The type descriptor contains the address of the code for each primitive operation of the type. It probably also contains other information, such as might make membership tests convenient and efficient.

The primitive operations of a tagged type are known at its first freezing point; the type descriptor is laid out at that point. It contains linker symbols for each primitive operation; the linker fills in the actual addresses.

### Response

Although the inherited subprogram might be declared in the package_body, the fact that it is going to be declared there is known at compile time of the package_declaration. Thus, there is no conflict with the above-mentioned implementation model.

## Finalization and Unchecked_Deallocation

### Defect Report 8652/0104
### Section References 7.6.1

### Question

13.11.2(9) says that Free first performs finalization, then deallocates the storage. 7.6.1(17) says that (if Finalize propagates an exception,) Program_Error is raised after any other finalizations are performed.

In the case where Finalize propagates an exception, is the storage deallocated? Is it possible that Finalize will be called again on the same object?

### Response

The standard leaves this issue unspecified. There are three alternatives:

Alternative 1:

Storage is deallocated, and the object ceases to exist. Finalize will not be called again on that same object.

Alternative 2:

It is unspecified whether storage is deallocated. However, the object ceases to exist; Finalize will not be called again on that same object.

Alternative 3:

It is unspecified whether storage is deallocated, and whether the object ceases to exist. Finalize might be called again on that same object.

We choose Alternative 3, because this eases the burden on implementations, and because a Finalize that propagates an exception is a serious bug anyway. If there is a desire to recover from such situations, programmers should either prove that no such exception can happen, or put a "when others =>" clause in each Finalize procedure.

## Can an abstract subprogram be renamed?

### Defect Report 8652/0105
### Section References 8.5.4

### Question

Can an abstract subprogram be renamed? (Yes.) Can a subprogram which must be overridden in the sense of 3.9.3(6) be renamed? (No.)

Consider an example with an abstract parent type and a primitive operation. 8.5.4(8) says that the renaming uses the original inherited subprogram, not the overriding version.

```
package Types is
   type T is abstract tagged ...
   procedure P (F : T) is abstract;
end Types;
package Extensions is
   type E is new Types.T with ...
   procedure Pr (F : E) renames P;  -- renaming of inherited P
                                    -- Legal? (No.)
   procedure P  (F : E);
end Extensions;
```

A similar example can be constructed with a function with a controlling result.

```
package Types is
   type T is tagged ...
   function F return T;
end Types;
package Extensions is
   type E is new Types.T with ...
   function Fr return E renames F;  -- renaming of inherited F
                                    -- Legal? (No.)
   function F  return E;
end Extensions;
```

### Response

The intent of the language is that an abstract subprogram can be renamed, and the renamed view is also abstract. All of the rules about declaring abstract subprograms apply at the point of the renaming (in particular, 3.9.3(3)).

Similarly, it is possible to write a renaming of an inherited subprogram which must be overridden because of the rules of 3.9.3(6). The intent of the language is that the "shall be overridden" property also applies to the renamed view. However, it is not possible to give an overriding for the renamed view (as the overriding specification would be an illegal homograph of the renamed subprogram). Thus, any renaming of an inherited subprogram that must be overridden is illegal.

## Daylight savings and Ada.Calendar

### Defect Report 8652/0106
### Section References 9.6

### Question

Suppose an implementation chooses to keep Calendar.Clock in some canonical form, such as GMT, and that it also chooses to support daylight savings time, by making Split and Time_Of do the appropriate conversions. Is this a valid implementation? (Yes.)

If so, what should Split do when daylight savings time ends in the fall, given that there are two different Time values for some Year/Month/Day/Second values? And what should Time_Of do when daylight savings time starts in the spring, given that there are Year/Month/Day/Second values that do not correspond to any Time?

### Response

9.6(24,26) say:

> The functions Year, Month, Day, and Seconds return the corresponding values for a given value of the type Time, as appropriate to an implementation-defined timezone; the procedure Split returns all four corresponding values. Conversely, the function Time_Of combines a year number, a month number, a day number, and a duration, into a value of type Time. The operators "+" and "-" for

> addition and subtraction of times and durations, and the relational operators for times, have the conventional meaning.
>
> The exception Time_Error is raised by the function Time_Of if the actual parameters do not form a proper date. This exception is also raised by the operators "+" and "-" if the result is not representable in the type Time or Duration, as appropriate. This exception is also raised by the function Year or the procedure Split if the year number of the given date is outside of the range of the subtype Year_Number.

9.6(24) uses the phrase "as appropriate to an implementation-defined timezone". The implementation is free to do daylight-savings-processing as part of its time zone handling, so the implementation suggested in the question above is valid. Since it explicitly says "implementation defined", the implementation can do what it wants, so long as the behavior is documented.

When implementing Ada on an operating system that supports time zones, it makes sense to implement the Ada operations in terms of the corresponding operating system operations. It is not reasonable to require Ada programs to do something different from other programs on that system. Nor is it reasonable to require an Ada implementation to re-implement this processing.

9.6(26) might be taken as a requirement that Time_Error be raised by Time_Of in the spring. However, the standard does not define what is meant by a "proper date". Clearly, February 31 is not a "proper date" -- the term is not entirely meaningless -- but surely the term is vague enough to allow the Ada implementation to use the underlying operating system's notion of "proper date" with respect to the questionable cases considered here. Furthermore, raising Time_Error is probably undesirable, since it is more likely to cause an Ada program to malfunction in rare cases, than to catch a bug in the program.

## Pragma Elaborate for child units

### Defect Report 8652/0107
### Section References 10.1.2

### Question

10.2(9) says:

> The order of elaboration of library units is determined primarily by the elaboration dependences. ... In addition, if a given library_item or ... has a pragma Elaborate ... that MENTIONS another library_unit, then there is an elaboration dependence of the given library_item upon the body of the other library_unit, ...

10.1.2(6) states:

> A library_item is MENTIONED in a with_clause if it is denoted by a library_unit_name or a prefix in the with_clause.

Is the term "mentioned" as used in 10.2(9) meant to be defined by 10.1.2(6)? (No.) If so, it would imply that a pragma Elaborate on a child unit causes an elaboration dependence upon the parent of that child unit (as well as on the child unit itself). Is this the intent? (No.)

### Response

The intent is that 10.1.2(6) is defining "mentioned in a with_clause", not "mentioned" in general. The term "mentioned" in 10.2(9) is used in an informal sense; it merely means the library unit to which the pragma applies, and not its parent.

10.2.1(26) clarifies this:

> A pragma Elaborate specifies that the body of the named library unit is elaborated before the current library_item.

If transitive semantics is desired, then pragma Elaborate_All should be used.

## Separate compilation of generic bodies

### Defect Report 8652/0108
### Section References 10.1.4

### Question

10.1.4(3) states that the mechanisms for replacing compilation units within an environment are implementation defined. Is it intended that "mechanisms" also means "circumstances"? (No.)

This seems to be partly contradicted by NOTE 7 in 10.1.4(10) which prevents automatic removal of units that instantiate a generic body being replaced.

This NOTE however, does not seem to follow from any normative statement. 10.1.4(7) allows removal of compilation units that depend upon a given one, but nothing apparently prevents removal of compilation units that don't have such a dependency.

### Response

10.1.4(3) is simply saying that the "commands" or "mouse clicks" that the user gives to perform various actions are implementation defined. It is not intended as a permission for the implementation to arbitrarily remove compilation units from the environment.

10.1.4(7) says when compilation units can be automatically removed, and this list is intended to be complete.

An implementation may, of course, have additional commands for removing compilation units, or performing any other actions on the environment.

## Matching rules for generic formal access-to-constant types

### Defect Report 8652/0109
### Section References 12.5.4

### Question

12.5.4(4) states that "If and only if the general_access_modifier constant applies to the formal, the actual shall be an access-to-constant type". Is it really intended to forbid an access-to-variable type from being passed as an actual to a generic formal access-to-constant type? (Yes.)

### Response

Consider the following example:

```
type Actual is access all Integer;
X: aliased constant Integer;

generic
    type Formal is access constant Integer;
package Gp is
    function F return Formal;
end Gp;

package body Gp is
    function F return Formal is
    begin
        return X'access;
    end F;
```

```
   end Gp;

   package Ip is new Gp (Formal => Actual); -- Illegal.
```

If the above were legal, then Ip.F would produce an access-to-variable value designating a constant, thus allowing a constant to be modified. This would be very bad. Hence, the "if and only if" wording of 12.5.4(4).

## Order of Size and Small clauses for fixed point types

### Defect Report 8652/0110
### Section References 13.3

### Question

Consider:

```
   type Two_Bits_Spare is delta 4*System.Fine_Delta range -1.0..+1.0;
   for Two_Bits_Spare'Size use Size_For_Fine_Delta - 2; -- Legal? (Maybe.)
```

where Size_For_Fine_Delta is the number of bits needed for a type whose delta is Fine_Delta. Does the Size clause force the implementation to choose the small of the type such that only Size_For_Fine_Delta - 2 bits are needed? (No.)

13.3(55) says that 'Size should/must be "the number of bits needed to represent each value belonging to the subtype ..." and that an implementation should/must support a specified 'Size for a first subtype that reflects this representation.

3.5.9(8) says "the set of values of a fixed-point type comprise[s] the integral multiples of a number called the small of the type" which if not specified "is an implementation-defined power of two ..."

### Response

A Size clause does not determine the values of an ordinary fixed point type. The values are determined either by the implementation, or by a Small clause; the legality of a Size clause is determined in part by the values chosen.

Since there is no Small clause in the above example, 3.5.9(8) allows the implementation to choose among various powers of two as the small, and the small determines what the values of the type are. A Size clause is required to specify enough bits to represent all those values. Thus, the Size clause in the above example is legal if the implementation chooses small to be 4*System.Fine_Delta, but is illegal if the implementation chooses small to be System.Fine_Delta.

## Storage pools and access types designating task types

### Defect Report 8652/0111
### Section References 13.11

### Question

If a user-defined storage pool is specified for an access-to-task type, does this imply that the user's storage pool will be used for all data structures associated with the task, such as the TCB and the task stack? (No.)

### Response

Many implementations choose to implement a task object as an access to the implementation-defined data structures associated with the task (such as a Task Control Block (TCB) and a task stack). In such an implementation, a user-defined storage pool will control only the allocation of the task objects (i.e. the access object), and will have no effect on the allocation and deallocation of those other resources.

Note that in some such systems, it may be impossible to allow user-defined allocation of TCBs via the storage pool mechanism, because the TCBs are in a separate address space, or are allocated in a run-time system table.

## A box for a formal subprogram_default freezes the actual

### Defect Report 8652/0112
### Section References 13.14

### Question

Consider:

```
type T is ....
type T2 is ...
function "+"(A: T; B: T2) return T;

generic
    type FT is private;
    with function "+"(A: FT; B: T2) return FT is <>;
package GP is ...

package IP is new GP(T);
    -- does this instantiation freeze "+" and therefore T2? (Yes.)
```

13.14(5) doesn't cover that case.

### Response

A box for a formal subprogram_default freezes the actual subprogram determined in an instantiation.

12.6(10) says, "If a generic unit has a subprogram_default specified by a box, and the corresponding actual parameter is omitted, then it is equivalent to an explicit actual parameter that is a usage name identical to the defining name of the formal."

Thus, the instantiation GP(T) is equivalent to the instantiation GP(T, "+").

13.14(4,5,14) then demand that "+" and T2 are frozen by the instantiation.

## Saving and restoring Current_Output

### Defect Report 8652/0113
### Section References A.10.3

### Question

It's clear from the example in section A.4.3 of the Rationale that one should be able to use the subprograms Current_Output and Set_Output to save and restore Text_IO's current output file, using an object of type File_Access. However, given what I believe to be a typical implementation of Text_IO, this won't work.

Assume that Text_IO declares a hidden aliased object of type File_Type, called, say, Current_Output_Object. (How the type File_Type is declared is irrelevant here; it may be a pointer, a descriptor record, a descriptor number, or whatever, as long as the view from the body of Text_IO is not limited.) The bodies of the Current_Output and Set_Output functions can then simply be:

```
procedure Set_Output (File : in File_Type) is
begin
   Current_Output_Object := File;
end Set_Output;

function Current_Output return File_Type is
begin
```

```
      return Current_Output_Object;
   end Current_Output;

   function Current_Output return File_Access is
   begin
      return Current_Output_Object'access;
   end Current_Output;
```

Recall that the example in Rationale-A.4.3 looks like this:

```
   procedure P(...) is
      New_File     : File_Type;
      Old_File_Ref : constant File_Access := Current_Output;
   begin
      Open(New_File, ...);
      Set_Output(New_File);
      -- use the new file
      Set_Output(Old_File_Ref.all);
      Close(New_File);
   end P;
```

If we are using the Text_IO implementation above, Old_File_Ref will be initialized to point to Current_Output_Object, the first call to Set_Output will change the value of Current_Output_Object, the second call to Set_Output will have no effect, and the Close will close a file that's still being used as the current output file. Thus any operations on Current_Output after the call to P will be erroneous by A.10.3(23).

Is the suggested implementation of Text_IO legal? (No.) If so, the example in the Rationale is non-portable. If not, how can the prohibition be inferred from the International Standard?

### Response

The suggested implementation is wrong.

A.7(2) states, "In the remainder of this section, the term file is always used to refer to a file object...." A.10(5) states, "At the beginning of program execution the default input and output files are the so-called standard input file and standard output file." By A.7(2), this can be paraphrased as follows: "At the beginning of program execution the default input and output file objects are the so-called standard input file object and standard output file object." That is, the terms "default input file object" and "default output file object" are not names of distinct file objects, but of a *role* played by a file object. That is, the only way that a single file object can be both the standard input file and the default input file is if the default input file is not a file object distinct from all other file objects. (The role played by the file objects currently acting as the default objects is described by A.10(4): "If no file is specified, a default input file or a default output file is used." This is the closest the International Standard comes to defining what a default file is, and the terms should probably have been italicized.)

It follows that Current_Output and Set_Output can be used to save and restore Text_IO's current output file, using an object of type File_Access.

## Mapping between Interfaces.C.char and Standard.Character

### Defect Report 8652/0114
### Section References B.3

### Question

B.3(46) states that To_Ada and To_C map between Character and char, but does not explain how. Presumably, Interfaces.C.char corresponds to the C type char, i.e., to the native character set of the target machine. Type Character, of course, always corresponds to Latin-1, regardless of the target machine. On an EBCDIC machine, does To_C('A') yield the C.Interfaces.char value corresponding to EBCDIC 'A', or does it yield the character whose EBCDIC code is Character'Pos('A')?

### Response

The intent is that 'A' maps to 'A', even if the two 'A's have different representations.

The following definition is equivalent to the above summary:

> To_C (Latin_1_Char) = char'Value(Character'Image(Latin_1_Char)) provided that char'Value does not raise an exception; otherwise the result is not defined by the language.
>
> To_Ada (Native_C_Char) = Character'Value(char'Image(Native_C_Char)) provided that Character'Value does not raise an exception; otherwise the result is not defined by the language.

## Ada.Task_Identification.Is_Callable for the environment task

### Defect Report 8652/0115
### Section References C.7.1

### Question

What is the behavior of Ada.Task_Identification.Is_Callable for the environment task? In particular, does it change in value from True to False when the main subprogram exits and starts waiting for library-level tasks to terminate? (Yes.)

### Response

The NOTE C.7.1(21) says that Ada.Task_Identification.Current_Task can return a Task_Id value designating the environment task. 9.8(2) specifies that Is_Callable returns True unless the task is completed or abnormal. The structure of the environment task given in 10.2(10-12), and the dynamic semantics of 10.2(25) tell us that the environment task is completed before waiting for dependent tasks. The erroneous execution case of C.7.1(18) should not apply, as the environment task continues to exist until the partition terminates.

Therefore, the value of Ada.Task_Identification.Is_Callable is well defined. It is possible to test the value after the completion of the environment task in a library-level task (while the environment task is waiting for dependent tasks to terminate). Thus it cannot be implemented purely as a return of True at all times.

The Ada.Task_Identification.Is_Callable flag can be useful, as a library-level task can use the value to terminate itself once the partition has completed. This is especially important to bullet-proof a reusable package containing a library task. At least one commercial Ada library uses this technique.

## One queuing policy per partition

### Defect Report 8652/0116
### Section References D.4

### Question

Implementation Permissions D.4(15) says:

> Implementations are allowed to define other queuing policies, but need not support more than one such policy per partition.

Does the permission to support only one policy per partition apply only to the "other" (implementation defined) policies? (No.)

### Response

The second part of the Implementation Permission applies to all queuing policies.

10.1.5(9) says:

> An implementation may place restrictions on configuration pragmas, so long as it allows them when the environment contains no library_items other than those of the predefined environment.

This implies that an implementation need not support more than one queuing policy per partition, whether or not the policy is implementation defined.  Of course, the "other" policies mentioned in D.4(15) are implementation defined, so the implementation can restrict them in any way it sees fit.  Required support for more than one policy in the same partition would be an implementation burden that is not worthwhile.