Final Minutes of the 11th ARG Meeting

30 June – 2 July 2000
Potsdam Germany

Attendees: John Barnes, Randy Brukardt, Kiyoshi Ishihata, Mike Kamrad, Stephen Michell, Erhard Ploedereder, Joyce Tokar; Jim Moore, briefly on 30 June.

Meeting Summary

The meeting convened on 30 June 00 at 14:00 hours at the Seminaris Hotel in Potsdam Germany and adjourned at 17:00 hours on 2 July. A sunny Saturday afternoon saw the ARG on a bike excursion to Sanssouci (or more correctly „Sans, Souci."), compensated by an evening session.

A full pass over all documents was completed. The review of the Defect Reports, Records of Response and the TC were largely divided into two categories: Corrigendum wording and other significant improvements, and editorial improvements in presentation wording. The Minutes proper describe the changes made in the first category, and the Appendix to the Minutes contains the changes in the second category. The reviewed version of the Defect Reports already incorporated the e-mailed comments from Pascal and John.

While there was no official vote, I (Mike) think that I speak for others in thanking Erhard for facilitating the excellent arrangements at the hotel.

Next Meeting

There are two possibilities for the next meeting, depending on the outcome of WG9 voting on the TC: 22-24 September 00 near Madrid Spain in conjunction with 10th IRTAW, and afternoon of 17 November 00 and all-day on 18-19 November 00 in Laurel MD in conjunction with the SIGAda Conference. Erhard will confirm these dates as soon as feasible.

Action Items

There was no review of old action items. For the record, all action items are repeated here, even if already done, unless they are clearly obsolete by now. A review will follow at one of the next meetings.

Old action items on AIs remain:

Randy Brukardt: new AIs on Unsuppress and on `Object_size
Norm Cohen: AI-133, AI-167, AI-173
Robert Dewar: AI-85 (distribute an ‚append" Test to comp.lang.ada and collate results to AI)
Gary Dismukes: AI-158, AI-196
Robert Eachus: AI-100, AI-172, AI-174, AI-185, AI-186
Mike Kamrad: new AI on Assert pragma
Pascal Leroy: AI-186(?),
Stephen Michell: AI-148, AI-187
Erhard Ploedereder: , AI-209, AI-212
Tucker Taft: AI-162, AI-188, AI-189, AI-191, AI-195(2.part), AI-216

All: Create tests for assigned AIs
Bob Duff: Be the test creator of last resort, distribute Design Note on „inward closures„
Randy Brukardt: distribute ‚inward-closures" study from prototyping project

New action items to be completed **before the next meeting** are**:**

Randy Brukardt: Update the next draft (Version 7) of the Corrigendum for ARG review by 15 July and for final distribution to WG9 by 31 July.
Erhard Ploedereder: Convert rtf version from Randy to pdf version.
All: Review Version 7 *[assigned pages of]* the TC, Record of Response and Defect Report and submit comments to Randy by 28 July.
Erhard Ploedereder: Publish logistics for the next meeting

(old action items)
Tucker Taft, Pascal Leroy: check out the new 3.7.1(7) AI-168

Pep Talk

Jim gave a pep talk, encouraging the ARG to produce a document that can be approved by email ballot in August. Jim reminded us that he plans another TC in two years in which to place any doubtful DRs this time around; presumably this TC will also contain amendments.

Substantive Changes to Defect Reports

Herein are the Defect Reports that have changes to Corrigendum wording or other substantive changes that would be of interest to all members of the ARG. The changes are noted by underlined, emboldened font for easy identification. The motivation for many of the changes was to be more concise and precise or for uniformity of style and this is the implied reason for the listed changes when no explicit reason is given.

DR 02:

There should be an added wording change: Change 4.8(10) to read (in part): „..any per-object constraints on subcomponents are evaluated **(see 3.8.)** and ...**“**

The parenthetical reference „(with 3.3.1)“ in the Question on 9.4(14) needs to be deleted.

DR09:

First paragraph of Section 13 is changed to reflect the inclusion of operational aspects:

> „This section describes features for querying and controlling certain aspects of entities and for interfacing to hardware.“

The following changes were made to the replacement paragraph 13.1 (1), to improve its presentation of the operational items and how it differs from representation item. The replacement paragraphs now read in part:

> **„Two aspects of entities can be specified: aspects of representation and operational aspects. Representation items specify how the types and other entities of the language are to be mapped onto the underlying machine. Operational items specify other properties of entities.**
>
> There are six kinds of *representation items*: attribute_definition_clauses for representation attributes, enumeration_representation_clauses, record_representation_clauses, at_clauses, component_clauses, and *representation pragmas*. They can be provided to give more efficient representation or to interface with features that are outside the domain of the language (for example, peripheral hardware).

An *operational item* is an attribute_definition_clause for an operational attribute.

An operational or representation item applies to an entity identified by a local_name, which denotes an entity declared local to the current declarative region, or a library unit declared immediately preceding a representation pragma in a compilation."

An additional replacement paragraph is needed for 13.1 (13) because both representation item and operational items are targets of the paragraph. The replacement paragraph now reads:

> „A representation **or operational** item that is not supported by the implementation is illegal, or raises an exception at run time."

A complete search of „representation items" in the RM was done to determine if „operation item" should also be included where „representation item" is referenced. Only these significant instances were found:

- Paragraph 3.8 (11): an operational item does not apply within type declaration that is the topic of this paragraph.
- Paragraphs 7.3 (5), 13.1 (10), 13.1 (15): add to the discussion that operation item is specifically not mentioned for obvious reasons.

No additional changes were found to be necessary. Similarly, a global search for „aspect" did not yield a need for additional changes. The AI/DR should probably state that in these places operational items are omitted intentionally and that further changes to 13.1(15,18) are made in the context of DR40.

There is a typo in the replacement paragraph 13.4(11): „a{n} enumeration_representation_clause".

DR10:

A typo in the replacement paragraph, 3.9.2 (7) -- „new" should be bold face:

> „A type_conversion is statically or dynamically tagged according to whether the type determined by the subtype_mark is specific or class-wide, respectively. For an object that is designated by an expression whose expected type is an anonymous access-to-specific tagged type, the object is dynamically tagged if the expression, ignoring enclosing parentheses, is of the form X'Access, where X is of a class-wide type, or is of the form **new** T'(...), where T denotes a class-wide subtype. Otherwise, the object is static or dynamically tagged according to whether the designated type of the type of the expression is specific or class-wide, respectively."

DR 14:

Make sure that 3.11.1(1) shows up in the Index of the new RM as the definition of the semantic term „body".

DR21:

The replacement paragraph has been changed and now reads, 7.6.1 (13)

> „If the object_name in an object_renaming_declaration, or the actual parameter for a generic formal **in out** parameter in a generic_instantiation, denotes any part of an anonymous object created by a function call, the anonymous object is not finalized until after it is no longer accessible via any name. Otherwise, an anonymous object created by a function call or by an aggregate is finalized no later than the end of the innermost enclosing declarative_item or statement; if that is a compound_statement, **the object** is finalized before starting the execution of any statement within the compound_statement. "

DR22:

The last sentence of the proposed replacement paragraph 7.6.1 (13) is only loosely connected to the rest of the paragraph. Separating it would be useful for presentation and it also eliminate the confusion with the change proposed by DR21 for the same paragraph. Hence paragraph 7.6.1 (13) is not replaced but instead a new paragraph is inserted, and the text now reads:

„Insert after 7.6.1 (13):

> The anonymous objects created by function calls and by aggregates are finalized no later than the end of the innermost enclosing declarative_item or statement; if that is a compound_statement, they are finalized before starting the execution of any statement within the compound_statement.

the new paragraph:

> If a transfer of control or raising of an exception occurs prior to performing a finalization of an anonymous object, the anonymous object is finalized as part of the finalizations due to be performed for the object's innermost enclosing master.“

DR24:

Insert the new paragraph after 7.6 (17) because it is more relevant to that section. This change is also validated by the repeated references to 7.6 throughout the DR. This will also change the cover page. And this will cause a renumbering of the DRs, so that DR24 becomes DR22, DR22 becomes DR23, and DR23 becomes DR24.

DR 27:

A short discussion over the mismatch of the corrigendum wording with the actual TC was resolved as one of those cases where multiple DRs change an original paragraph.

DR40:

Randy rewrote the whole AI/DR to reflect the comments by Pascal and Tucker,and to include relevant parts of AI-195 and DR09. Because of the tight integration with DR09 that provides the foundation of operational attributes, it would have been easier to merge the two DRs.

The change to the new paragraph after 13.1 (18) is more direct:

> „If an operational aspect is *specified* for an entity (meaning that it is either directly specified or inherited), then that aspect of the entity is as specified. Otherwise, the aspect of the entity has the default value **for that** aspect. “

Likewise the change to replacement paragraph 13.3 (75) is also more direct:

> „S'External_Tag denotes an external string representation for S'Tag; it is of **the predefined** type String. External_Tag may be specified for a specific tagged type via an attribute_definition_clause; the expression of such a clause shall be static. The default external tag representation is **implementation defined**. See 3.9.2 and 13.13.2. The value of External_Tag is never inherited; the default value is always used unless **a new value is directly** specified for a type.“

Most changes to replacement 13.13.2 (9) are meant to eliminate redundancy, to be more direct or to account for combination of both Read and Write operations. The change for the last sentence is motivated by the following example:

```
Type T is limited tagged …;
```

```
For T'Read use …;

Type Der is new T with null record;  --  ok

Type Der_Int is new T with -- ok
   Record
      Int : Integer;
   End record;

Type Der_Protect_Type is new T with –- only ok if Protect_Type'Read or
                                    -- Der_Protect_Type'Read is specified
   Record
      PT : Protect_Type;
   End record;
```

The replacement now reads:

> „For untagged derived types, the Write **and** Read **attributes** of the parent type **are** inherited as specified in 13.1; otherwise, the default **implementations** of **these attributes are** used. The default implementations of Write and Read attributes **execute** as follows:
>
> For elementary types, the representation in terms of stream elements is implementation defined. For composite types, the Write or Read attribute for each component is called in canonical order**, which is** last dimension varying fastest for an array, and positional aggregate order for a record. Bounds are not included in the stream if *T* is an array type. If *T* is a discriminated type, discriminants are included only if they have defaults. If *T* is a tagged type, the tag is not included. For type extensions, the Write or Read attribute for the parent type is called, followed by the Write or Read attribute of each component of the extension part, in canonical order. For a limited type extension, if the attribute of any ancestor type has been directly specified and the attribute of any ancestor type of any of the **extension** components **which are of a limited type** has not been specified, the attribute of *T* shall be directly specified."

The replacement paragraph 13.13.2 (25) should be revised in a manner similar to the first paragraph of replacement 13.13.2 (9); it now reads:

> „For untagged derived types, the Output **and** Input **attributes** of the parent type **are** inherited as specified in 13.1; otherwise, the default **implementations** of **these attributes are** used. The default implementations of the Output and Input operations execute as follows:"

The replacement for paragraph 13.13.2(36) should end with the first paragraph. The first and last sentence of the following paragraph should become a paragraph under Implementation Requirements. The remaining text is deleted, because the 2. sentence in the second paragraph is implied by the word „readable".; the third paragraph is a consequence of the „otherwise"-part of the new 13.13.2(25).

DR-42:

Delete „--never in the abstract." in the 2.para. of the Discussion.

DR 47:

Make sure of long hyphens in the corrigendum wording.

DR49:

It is noted that A.4.4 (101) says nothing about the value of the lower bound of the returned slice; this appears to be an omission, which deserves an AI of its own. It is better to do nothing to this paragraph than to select a value now at the risk of invalidating implementations.

DR50:

Fix the typo in the replacement paragraph, A.5.2 (40):

> *Bounded (Run-Time) Errors*
>
> It is a bounded error to invoke Value with a string that is not the image of any generator state. If the error is detected, Constraint_Error or Program_Error is raised. Otherwise, a call to Reset with the resulting state will produce a generator such that calls to Random with this generator will produce a sequence of values of the appropriate subtype, but which might not be random in character. That is, the sequence of values might not fulfill the implementation requirements of this **subclause**.

DR53:

The replacement paragraph A.10.3 (22) is changed because the both conditions are required to be erroneous; it now reads:

> „The execution of a program is erroneous if it invokes an operation on a current default input, default output, or default error file**, and** if the corresponding file object is closed or no longer exists.“

DR56:

Change the tense of the last verb in the inserted paragraph after A.12.1 (36):

> „If the File_Type object passed to the Stream function is later closed or finalized, and the stream-oriented attributes are subsequently called (explicitly or implicitly) on the Stream_Access value returned by Stream, execution is erroneous. This rule applies even if the File_Type object was opened again after it **had been** closed.“

Change „execution“ to „Execution“ in the insertion after A.12.1(36).

DR61:

This change to replacement for B.3.1 (24) specified from the previous meeting was not made and should now read:

> „If Item is **null**, then To_Chars_Ptr returns Null_Ptr. If Item is not **null**, Nul_Check is True, and Item.**all** does not contain nul, then the function propagates Terminator_Error; **otherwise** To_Chars_Ptr performs a pointer conversion without allocation of memory.“

DR 66:

In the replacement for B.4(65), change „characters and finally“ by „ characters, followed by“.

DR83:

Changing a period to a semicolon at the end:

- The primitive subprograms of the corresponding specific limited private type shall only have access parameters if they are controlling formal parameters; each non-controlling formal parameter shall have either a nonlimited type or a type with Read and Write attributes specified via an attribute_definition_clause**;**

DR100:

This DR is removed from the Defect Reports because there is no question being posed that would need answering. The RM is clear on this point and the questioner is really asking for an amendment. Change the numbers of the remaining DRs accordingly.

DR111:

The cover page describes this DR with the Qualifier Omission, which clearly would put the DR into Part I and Corrigendum wording would be required.

Erhard recommends that this DR be dropped from the RoR because he believes that Corrigendum wording is necessary, since he doesn't see how this conclusion could be derived from the RM. (He does not disagree with the conclusion, merely with the lack of wording, resp. some argument how one could possibly derive this rule from the RM, especially the „non-transitivity" across levels of actuals. The attendees agreed.

DR113:

Remove this DR because it is only a comment on the AARM. Change the number of the remaining DRs accordingly.

DR117:

After some discussion about how best to handle the information described in last two sentences of the Response, it was decided that it is best to place the information in these sentences into the AARM and thereby leave this DR in Defect Report Part II and the Records of Response.

DR120:

After review, it was decided that the wording in the RM may be grammatically unfortunate, but certainly is not ambiguous because the wording can not produce the first interpretation. Consequently, the DR will be dropped from Defect Report Part II; and this point will be emphasized in the AI by replacing the Response with a statement saying that :
„**The first interpretation can not be deduced from this wording.**"

Appendix: Editorial Changes in Presentation

All the editorial changes in the documents are collected here. Only the changes themselves are described and they are largely to help Randy verify the changes that he collected during the meeting. Consequently, they will have more meaning to Randy than any one else.

General Changes:

The following changes are to be applied to all the documents:

1. About DR cover sheets: replace Randy as WG Secretariat with James W. Moore; add Randy to submitters; replace double dashes by a long hyphen (in addition to those discovered in Issue 6 below); verify dates in pages with Jim Moore.
2. Fix mismatched quotes, as shown by question in DR 01. Note that some are in the TC!
3. Inconsistent usage of RM and ARM as a prefix for paragraph references; it was decided to remove both and just use the paragraph numbers
4. Replace stand-alone use of RM and ARM with word „standard“, but make sure that this reads well in context
5. Inconsistent spacing, as shown by DR120; Randy says this is a problem with the Scribe source; this can only be fixed by changing the rtf file.
6. Replace double dashes with long hyphens in DR01, DR11, DR35, DR47 and the title; Randy is directed to fix this by changing his translation code. Definitely make sure that this problem is globally fixed at least on official title pages and anywhere in the TC.
7. Remove the periods in the titles of the DRs. Also, for DR titles use proper case for headings.
8. Replace „Clarification requested“ in the DR cover page with „Clarification required“ for the Part One DRs; and stay with „Clarification requested“ for the Part Two DRs.
9. Replace „Ada95“ with „Ada“, unless there is also a reference to Ada83 involved.
10. Because several cover sheets are too long due to the detailed listing of paragraphs that are effected, eliminate paragraph references from the cover sheets as they are neither needed by nor known to ISO.
11. Normalize the use of the term „language-defined“ to include a hyphen, when used as an adjective (and no hyphen otherwise)
12. See if the second line of DR titles can be aligned with the beginning of text of the first line.
13. Replace all references RM83 with Ada83; also, make a note in the introduction of the DRs and the RoR that Ada83 means ISO/IEC 8652:1987;
14. Remove the artificial uses of **begin** and **end** in the examples (possibly by hand-editing the .rtf version)
15. Make sure that „Records of Response“ is consistently used (including title of the document), pending verification by Jim Moore.
16. Remove all unnecessary blanks after underscores and hyphens. Note that some are in TC wording!
17. Where paragraph numbers are needed on text from the RM for presentation purposes, consider a notation for them that makes it clear that the paragraph numbers are not normative.
18. DR cover sheets should have „Languages“ spelled in lower case.

Introduction to the Defect Report Part I:

This document contains defect reports on the Ada 95 standard [**ISO/IEC 8652**:1995], and responses formulated by the Ada Rapporteur Group (ARG) of **ISO/IEC JTC 1/SC 22/WG 9**, the Ada working group. The ARG is the language maintenance subgroup of **WG 9**, meaning that it is responsible for determining the corrections to the standard.

The Defect Reports and Responses contain references to the Annotated Ada Reference Manual (AARM). This document contains all of the text in the Ada Reference Manual along with various annotations. It was prepared by the Ada 95 design team, and is intended primarily for compiler writers, test writers, and the ARG. The annotations include **rationale** for some rules**. The AARM** is often used by the ARG to determine the intent of the language designers.

Introduction to the Defect Report Part II:

This document contains defect reports on the Ada 95 standard [**ISO/IEC 8652**:1995], and responses formulated by the Ada Rapporteur Group (ARG) of **ISO/IEC JTC 1/SC 22/WG 9**, the Ada working group. The ARG is the language maintenance subgroup of **WG 9**, meaning that it is responsible for determining the corrections to the standard.

The Defect Reports and Responses contain references to the Annotated Ada Reference Manual (AARM). This document contains all of the text in the Ada Reference Manual along with various annotations. It was prepared by the Ada 95 design team, and is intended primarily for compiler writers, test writers, and the ARG. The annotations include **rationale** for some rules**. The AARM** is often used by the ARG to determine the intent of the language designers.

Introduction to the TC:

This corrigendum contains corrections to the Ada 95 standard [**ISO/IEC 8652**:1995].
Instructions about the text changes are in this font. The actual text changes are in the same fonts as the Ada 95 **standard** - this font for text, this font for syntax, `and this font for Ada source code.`

Introduction to Records of Response:

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.

Other Comments on Records of Response:

Style issue: Make the Defect References and Section References in each Record of Response in rtf document look like html document; otherwise, use colons.

DR01:

2.1(8-9) **say**:

> **upper_case_identifier_letter**
> Any character of Row 00 of ISO 10646 BMP whose name begins ``Latin Capital Letter".
>
> **lower_case_identifier_letter**
> Any character of Row 00 of ISO 10646 BMP whose name begins ``Latin Small Letter".

The version of 10646-**1:1993**, referred to in 1.2(8), names codes **C6** and **E6** as „Latin Capital Ligature AE" and „Latin Small Ligature AE".
(Here we have two examples of straight quotes that need to be changed globally.)

DR02:

> 9.4(14) creating a protected object (This one is supposed to be redundant with 3.3.1, but in fact the two paragraphs appear to be inconsistent.)

The elaboration consists of the evaluation of each per-object expression of the component's constraint, followed by the conversion of the value of each expression of the constraint to its appropriate expected type and the performance of the compatibility check defined for the elaboration of the subtype indication (see 3.2.2(11)). The values used for any expressions that are not part of per-object expressions of the subtype's constraint are those determined during the original elaboration of the component definition as defined in 3.8(18). Such expressions are not reevaluated during elaboration of the per-object constraint that occurs as part of object creation, despite **any rules** that state when a per-object constraint is elaborated (**e.g.,** as part of evaluating an allocator or aggregate).

Add a wording change for 4.8 (10) to add a reference to 3.8 to be consistent.

DR03:

On a 36-bit two's complement machine, one would declare:

```
type T is mod 2**36;
```

and T'Modulus would be 2**36, and the base range of T would be 0..2**36-1. If one says:

```
type TT is mod 2**36-1;
```

TT'Modulus is 2**36-1, and the base range of **TT** is **usually** 0..2**36-2. The implementation permission says that the base range of **TT** can be 0..2**36-1. This means that the all-ones bit pattern is a valid value of the type, and is not reduced via the modulus.

DR04:

S'Digits S'Digits denotes the requested decimal precision for the subtype S. The value of this attribute is of the type *universal_integer*. The requested decimal precision of the base subtype of a floating point type T is defined to be the largest value of *d* for which
ceiling(*d* * log(10**) / l**og(T'Machine_Radix)) + *g* <= T'Model_Mantissa
where *g* is 0 if Machine_Radix is a positive power of 10 and 1 otherwise.

DR06:

„sans-serif“ without hyphen, as two words thoughout the DR.

DR07:

12.5(**10**) (a NOTE) says that "A discriminant_part is allowed only for certain kinds of types, and therefore only for certain kinds of generic formal types. See 3.7."

DR08:

The ARG also discussed the following example, which illustrates another case where the **standard** seems to allow a discriminant to be changed:

"if a component_definition contains the reserved word aliased and the type of the component is discriminated, then the nominal subtype of the component shall be constrained." (**3.6(11))**

Aliasedness of the components is not really what is causing trouble, though. It is really the existence of a general access type, and in fact of a discriminant constraint on such an access type, which causes trouble. Thus, forbidding

such a constraint **is** the **chosen** solution, especially considering that constraints on access types are not a terribly useful feature.

DR09:

The inclusion of the note number „11“ is intentional and o.k.

The definition of stream attributes as "representation attributes" has proven to be a continuing problem. Several issues have made it necessary to **exempt** stream attributes from the rules for representation attributes, **indeed the number of such exemptions makes it clear that it is confusing to classify them as representation attributes**. Therefore, we have taken the major step of defining a new kind of attribute, the "operational attributes", and redefining stream attributes to be of this kind.

Typo on D.8 (36) in the original RM:

> The implementation shall document any aspects of **the** external environment that could interfere with the clock behavior as defined in this clause.

The typo like this (double „the the“) will not be mentioned in the TC but will be fixed in the RM.

DR10:

An analogous rule applies to an attribute reference of **Unchecked_Access and** to an allocator.

DR11:

Do replace this separator line,

- ----------

with a line separator?

Unless specified otherwise in the **standard**, the default convention of any entity is Ada.

An explicitly declared dispatching operation shall not have convention Intrinsic. However, an implicitly declared dispatching "/=" operator with **Boolean** result legally has convention Intrinsic.

If an Ada 95 implementation is tightly integrated with another language, such as C++ or Java, it is nice if an Ada tagged type can be declared as an extension of a (foreign) type (or class) of the other language. Presumably, all of the dispatching operations of this foreign type would be defined as imported, with the convention of that other language. When defining the type extension in Ada, it would be very inconvenient if every overriding **needed** a pragma Convention on it to match that of the inherited operation, as required by 3.9.2(10).

Remove the „- ----------„ in front of these lines

2.a. The "Breach of Privacy" Issue

2.b. Deriving the convention of operations from the type

2.c. The convention of a partial view

Since current rules imply that the convention of a type needs to be specified for the full view of the type, such dependency creates yet another breach of privacy in the case of private tagged types. However, the breach already

exists as explained in 2.a. and **then to** exploit it for more convenience to the user and a cleaner overall model seems justified.

The wording change to 3.9.2(10) shown above means that **it is permitted** to have such an inherited subprogram. If the **specification** of G contained a type extension of Formal, then that type's inherited Proc would also have convention Intrinsic, which would be legal. However, an explicit overriding of that Proc would be illegal.

DR14:

It seems that the right model for renaming-as-body that occurs after the subprogram is frozen should be that of a wrapper subprogram, with its own elaboration **flag.**

See DR-0027 (**AI**-00135) for a discussion of circularities involving renamings-as-body.

This issue also adds the missing definition of the **semantic term** „body“. This change makes a renaming-as-body a body. However, doing so triggers the freezing rule 13.14(3): "A noninstance body other than a renaming-as-body causes freezing of each entity declared before it within the same declarative_part." It clearly was the intent of the designers of the language that renaming-as-body not freeze (otherwise the second sentence of 8.5.4(5) could never be true), and existing compilers do not freeze when a renaming-as-body is encountered. We do not want to change this behavior, so we add an exception to 13.14(3).

DR15:

AARM J(1**.d**) lists several Ada 83 floating-point attributes that have been removed from the language, including 'Small; **AARM** J(1.h), however, says that "Implementations can continue to support the above features for upward compatibility".

The intent is that implementations be allowed to support all Ada 83 attributes, for upward compatibility. Thus, it is important that they be allowed to support the Small attribute on floating point types. Therefore, this **resolution** makes a specific exception to the rule in 4.1.4(12).

DR16:

This would seem to imply that the composability of these "=" operators depends on whether the implementation chooses to implement them as tagged types, by 4.5.2(14-**15**):

> **For** a type extension, predefined equality is defined in terms of the primitive (possibly user-defined) equals operator of the parent type and of any tagged components of the extension part, and predefined equality for any other components not inherited from the parent type.
>
> **For** a private type, if its full type is tagged, predefined equality is defined in terms of the primitive equals operator of the full type; if the full type is untagged, predefined equality for the private type is that of its full type.

and by 4.5.2 (21-24):

> **Given** the above definition of matching components, the result of the predefined equals operator for composite types (other than for those composite types covered earlier) is defined as follows:
>
> > **If** there are no components, the result is defined to be True;
> >
> > **If** there are unmatched components, the result is defined to be False;

> **Otherwise**, the result is defined in terms of the primitive equals operator for any matching tagged components, and the predefined equals for any matching untagged components.

Item 3 is irrelevant here, since none of the types in question is (visibly) tagged.

However, for the private types mentioned in the question, the **standard** does not specify whether the underlying type is tagged, nor whether the equality operator is truly user-defined (as opposed to just being the normal bit-wise equality).

As to the second question, the **standard** clearly does not define any order of calling "=" on components, nor does it say whether the results are combined with "and" or "and then". Equality operators with side effects are questionable in any case, so we allow implementations freedom to do what is most convenient and/or most efficient. Consider equality of a variant record: The implementation might first check that the discriminants are equal, and if not, skip the component-by-component comparison. Alternatively, the implementation might first compare the common elements, and *then* check the discriminants. A third possibility is to first compare some portions with a bit-wise equality, and then (conditionally) call user-defined equality operators on the other components. All of these implementations are valid.

DR17:

The declaration of C1_Ren in the generic G is legal as per **8.5.1**(5), because the formal type F is indefinite. But when G is instantiated with type T2, the actual type is definite, so now we have renamed a component that may disappear by assignment to the variable Y. Note that the declaration of C1_Ren might be in the body of G, so we cannot avoid this problem by rechecking **8.5.1**(5) on the instantiation.

Assume that the implementation chooses to pass X by reference. Then, 6.4.1(10) says that there is an implicit view conversion to Indefinite_Child, and the formal parameter X then denotes the result of this view conversion. The result of the explicit view conversion is unconstrained, and the result of the implicit view conversion is also unconstrained, hence X is unconstrained, which violates the language design principle of **the note** 3.7(28).

Note that if the implementation chooses to pass by copy, then there is an implicit value conversion -- see 6.4.1(11)**.**

The fix for example 1 is to forbid the renaming: even though the formal type **looks** indefinite, it is possible for the actual type to be definite. Note that the manual already covers the case where C1_Ren is declared in the visible part of the generic unit, because legality rules are checked in the instance (**12.3**(11)). We are extending the legality rules for object renaming to apply in the private part of the instance, and we are assuming the worst in the body.

Also consider the following example:

DR18:

From 6.3.1(19-22), it would appear so: both attribute_references have the syntactic construction

prefix ' attribute_designator := name ' identifier
[a line break was added.]

It would be ludicrous to treat two different attributes to be fully conformant. None of the reasons for conformance checking would be enforced if this **were** true. Thus, the failure of the **standard** to require this can only be categorized as an oversight.

DR19:

```
package body Inner is
```

```
        ...
            -- At this point, we can see that Inner_Type is a Boolean type.
            -- So does Outer_Type have an "and" operator here?  (No.)
end Inner;
```

[a line break was added.]

The Ada 83 standard prefaced paragraph 7.4.2(6) by saying,"If the composite type is itself declared within the package that declares the private type", which avoided the problems introduced by 7.3.1(3,4,6). In attempting to be more general and include derived types as well as composite types, plus handle the case of child units (which are not "within" their parent package but are "within the declarative region of" their parent), the restriction imposed by the Ada 83 preface was unintentionally lost. Note that the AARM does not list this as a "Change from Ada 83", which is further evidence that this change was not intended. Also, paragraph 7.3.1(7.b) of the AARM makes it clear that these rules were only meant to pertain to types declared within the same declarative region as the component type or parent type providing the additional operations.

DR20:

Generally, add „package“ to the term „remote types“ in the discussion. Also make sure that the term then is preceded by a „the“ or „a“.

The following is a list of all the language-defined library units. On the right, "Pure" or "Preelaborate" are the pragmas **provided by the standard** for each package. (The three instances mentioned above are also shown as "Pure".) For each package, "Yes" means it could reasonably be a remote types package; "No" means it should not. If neither "Yes" nor "No" is shown, that means "No", and the reason is that the type(s) declared in that package are inappropriate for transporting across partitions. This is true of the I/O packages, for example -- we don't want to require transporting files across partitions.

Note that Ada.Calendar and Ada.Real_Time should not be remote types **packages**, because we wish to allow implementations to choose a different representation for the time-related types on different partitions. For example, type Calendar.Time on one partition might use a different "epoch" than on another partition. The types involved are: Calendar.Time, Real_Time.Time, Real_Time.Time_Span, and Real_Time.Seconds_Count. Thus, in order to pass time values across partitions, the programmer will have to define **an** application-specific time type, and translate to that.

Note that Ada.Exceptions is listed as "No", because it contains an access type that should not be a remote access type. It was one of the cases mentioned in the comments that prompted this **issue**. This is unfortunate; one would like to transport values of types Exception_Id and Exception_Occurrence.

Note that Ada.Task_Identification should not be **a** remote type **package**. If it were, then one could pass a Task_ID across partitions, and then do things like Abort_Task on that Task_ID on the "wrong" partition. This would require the tasking run-time system to know about distribution, which was never intended; alternatively, it would require us to declare such cases erroneous, which seems pointlessly error prone. Note also that this package cannot be declared Pure or Shared_Passive, because Task_ID is likely **to be** implemented **using an access type.**

This leaves Finalization, Characters.Handling, Command_Line, and Interfaces.COBOL as potential candidates for being remote types packages. Of these, the only significant additional functionality is for Finalization. Therefore, we choose to make Finalization a remote types package, and leave the other three as specified in the **current standard**. (It seems silly, for example, to make Command_Line a remote types **package**, so that **values of type Exit_Status** can be transported, when Strings.Unbounded is not made remote types, so that **values of type Unbounded_String** cannot be transported; the latter would be far more useful, if it were possible.)

Ada.Strings.Fixed A.4.3(5) Preelaborate No -- depends on Ada.Strings.Maps. This means the functions in this package cannot be called from the **specification** of a remote types package.

Ada.Strings.Unbounded A.4.5(3) No -- access type String_Access; depends on Ada.Strings.Maps. This means Unbounded_Strings cannot easily **be** passed across partitions.

Ada.Unchecked_Deallocation 13.11.2(3) Preelaborate No -- a procedure cannot be „remote types“, by **DR-0078**.

System.Storage_Elements 13.7.1(2) Preelaborate No -- depends on **System**

DR 21:

in the 2 question, „renames“ should be bold.

DR22:

In the case of explicitly invoked Adjust and Finalize operations, any exception propagated by such calls should simply be propagated as for an exception propagation that occurs as part of a call to any other user-defined subprogram. There is no **benefit by** requiring such exceptions to be converted to Program_Error.

DR23

One of the important goals of the finalization model with respect to exceptions (**see** 7.6.1(14-18)) is that if one controlled abstraction fails by raising an exception in Adjust or Finalize, this failure should not spread to other unrelated controlled abstractions. Even when one composite object happens to have two controlled parts, one from the "failed" abstraction and one from the "still-good" abstraction, the "still-good" abstraction should still have Adjust and Finalize called the appropriate number of times to keep reference counts **correct**, avoid dangling pointers, etc.

DR24

When an (extension) aggregate of a controlled type is assigned other than by an assignment or return statement, the aggregate is built "in place". No **anonymous** object is **created** and Adjust is not called on the **target** object.

> **For** an aggregate or function call whose value is assigned into a target object, the implementation need not create a separate anonymous object if it can safely create the value of the aggregate or function call directly in the target object.

However, it seems important to *require* this behavior, so that this kind of programming is portable (that is, it will **work portably** without raising Program_Error due to an access-before-elaboration from calling Adjust or Finalize before their bodies are elaborated).

Note that no Adjust ever takes place on an aggregate as a whole, since there is no assignment to the aggregate as a whole (**4.3 (5), AARM 4.3(5.b)**). AARM 7.6(**21.a**) talks about this case, and says that "only one value adjustment is necessary". This is misleading. It should say that only one adjustment of each controlled *subcomponent* (if any) is necessary in this case. *No* adjustments of the object as a whole are necessary (and as suggested above, such adjustments should be disallowed).

Similarly, if we have

```
function Is_Null (Value : in Dyn_String) return Boolean;
```

then the aggregate actual parameter in the call

```
if Is_Null ((Ada.Finalization.Controlled with ...)) then
```

is built directly in a temporary object, and Adjust is not called on the object as a whole.[Reservedwords ought to be bold.]

We **exempt** assignment and return statements from this requirement as there is no compelling reason to burden implementations with this requirement in those cases.

DR26:

A type extension is illegal if somewhere **within** its immediate scope it has two visible components with the same name. See also DR-0103.

DR27:

Change the Cover Page Qualifier to Omission *(because the change does not contradict anything in the manual).*

8.5.4(5) says:

> The profile of a renaming-as-body shall be subtype-conformant with that of the renamed callable entity, and shall conform fully to that of the declaration it completes. If the renaming-as-body completes that declaration before the subprogram it declares is frozen, the subprogram it declares takes its convention from the renamed subprogram; otherwise the convention of the renamed subprogram shall not be Intrinsic.

Remove the third paragraph from the Discussion because it is outdated (remnant of previous versions of the AI) and distracting now.

DR28:

Change the Cover Page Qualifier to Error because it is obviously an error.

> **The** ordering operators are predefined for every specific scalar type T, and for every discrete array type T, with the following specifications:
>
> ```
> function "<" (Left, Right : T) return Boolean
> ...
> ```

where the T is in Italics. Similar definitions are given throughout **Section** 4 for other predefined operators. What is the meaning of this italicized type name notation? Presumably, it is intended to refer to the base subtype, at least in some cases.

The definitions of the operators in **Section 4** take ...

Combining the first two paragraphs of the Discussion to integrate the declarations and the subtype association, make these changes:

- Replace the first sentence of first paragraph with

  „Consider the following type declarations, where the comments show the subtypes associated with the italicized notation in Section 4:“

- Move the specified types in second paragraph as comments in the declarations

Furthermore, 8.5.4(5) says:

However, consider:

Without this **ruling ...**

DR29

By the grammar in **9.1 (2)**, this task_type_declaration does not include a task_definition.

9.1(10) says that the elaboration of a task declaration elaborates the task_definition; what if there isn't one? (An empty task_definition is elaborated.)

9.1(11) says the elaboration of a task_definition creates the task type and its first subtype; if there is no task_definition, when are the task type and its first subtype created? (There is an empty task_definition.)

DR31:

Note that we do not say anything about the Callable attribute; if the Callable attribute becomes False, the task might still have a local cache that is **inconsistent** with global variables.

DR32:

A library subprogram body never completes an existing generic instance**, but replaces it.**

DR34:

Does the pragma Pure apply to the respective instances PI and QI ? **(No.)**

Since pragma Pure is a library unit pragma, are instantiations of P and Q illegal, if the resulting instances are not library units ? **(No.)**

Remove the second and fourth paragraph of Response because no such changes were made.

An exegesis of the **standard** showed that a clear answer to the questions cannot be derived from **it**. This exegesis is not reproduced for the Defect Report, but can be retrieved from the appendix of the working document AI-00041.

Additionally, a confirmation of the intent that this pragma should be explicitly specified is present in AARM E.2.3(**15.b**) where it is stated**,** "We considered making the public child of an RCI package implicitly RCI, but it seemed better to require an explicit pragma to avoid any confusion." It seems inconsistent to require an explicit pragma for a public child and not require an explicit pragma for an instantiation.

This concludes the list of predefined program unit pragmas. We have seen that, in some cases, applicability of the pragma to all instances would be seriously detrimental. We have seen other cases of library unit pragmas, where applicability to all instances may be more convenient on occasion, but is neither absolutely necessary nor warrants a rule that ***a priori*** precludes reusable generic units that can be instantiated in both restricted and unrestricted contexts.

Finally, we have seen that the existing language-defined program unit pragmas that are not library unit pragmas should apply to their instances. For the Inline pragma, this rule is already explicitly stated. However, as this presently matters only in cases, where applicability of the pragma to generic units is implementation-defined, and one can equally well conceive of future language-defined or implementation-defined pragmas, where automatic applicability to instances would not be appropriate, **it was decided** to make such an inheritance of pragmas by instances merely implementation advice, not a general semantic rule.

DR36:

Is the null check that occurs when evaluating a discriminant association for an access discriminant considered to be an Access_Check? **(Yes.)**

DR37:

For a generic formal type whose properties depend on a partial view (for example, a formal array type whose component type is a private type) the rules of 7.3.1 apply. *[two commas were dropped. Fix this problem in both the Summary of Response and Response]*

DR38:

The rule of 12.5.1(21) **is amended** to correct this problem. The copies of a formal derived type's operations in an instance are defined to be views of the corresponding copies of the primitive operations of the formal type's ancestor when the ancestor is a formal type, rather than simply those of "the ancestor type" (which in an instance would denote the actual type associated with the formal type's ancestor).

12.5.1(21) defines the implicit operations that are declared for a formal derived type, as well as the meaning of the copies of those implicit operations declared within an instance.

As shown in the example of the **question** section, this can result in inconsistent views of a formal type's primitive operations, since the formal view of a primitive may not be subtype conformant with the view in an instance. For example, modes of parameters may differ between the formal and actual views of an subprogram, leading to undefined semantics for the copied version of a call to such a subprogram from within an instance. This would essentially result in a generic contract model violation in the body of the instance.

This problem **is fixed** by specifying that, in an instance, the implicit declaration of a primitive subprogram of a formal derived type with a formal ancestor declares a view of the corresponding copied operation of the ancestor.

DR39:

Remove the word „clause“ from the first paragraph of Question and Summary.

Note that **Test_2.TC_Default_Value** denotes a constant initialized by Pack.TC_Default_Count, while **TC_Count_Test.Default** denotes Pack.TC_Default_Count. Do these match by the rule of 12.7(6)? (Yes.)

DR40:

Remove the hyphens from „implementation-defined“ in the original and replacement for 13.3 (75).
Add them for „language-defined“ (3x) in the Question and Summary

A few typos in the discussion: „...A{n} alternative“ , replace „involving pointers“ by „involving access values“, „for language{-}defined nonlimited..“, „to say [that] whether“

DR42:

H.4(8) says:

No_Local_Allocators

> **Allocators** are prohibited in subprograms, generic subprograms, tasks and entry bodies; instantiations of generic packages are also prohibited in these contexts.

Although it might be useful to know that if a generic body does not by itself violate a restriction, then neither will any instantiation, enforcing this kind of "contract" rule for restrictions that distinguish library level from non-library level usages would overly limit the nested instantiations of useful, benign generics. Furthermore, the pragma Restrictions is primarily designed to support application environments where schedulability and formal verification requirements dictate that generics can only be certified with **respect** to particular instantiations**.**

DR43:

Whenever enforcement of a restriction imposed by pragma Restrictions is not required by the **standard** prior to execution, but left to implementation-defined behaviour of dynamic semantics, it is reasonable to interpret pre-execution enforcement as a valid implementation-defined behaviour, provided that every execution of the partition will violate the restriction.

For the particularly critical rejection of programs that violate restrictions imposed by pragma Restrictions, the **standard** provides for implementation-defined behaviour in lieu of a compile- or link-time check otherwise required by 13.12(8). It is reasonable to interpret pre-execution enforcement of a configuration pragma as a valid implementation-defined behaviour, even if such enforcement is not required to occur prior to execution. This is particularly true for D.7(15), as this clause recommends the raising of a Storage_Error exception but does not specify the place where such an exception is to be raised.

DR44:

The following implementation permission is added:

If Stream_Element'Size is not a multiple of System.Storage_Unit, then the components of Stream_Element_Array need not be aliased.

DR45:

The programmer can reliably detect end-of-file for file streams as follows:

```
if not End_Of_File(An_Input_File) then
   T'Read(Stream(An_Input_File), Value);
   ...
end if;
```

DR46:

Does the declaration of I freeze the type T? **(Yes.)** If we replaced "Obj.D" with "Obj.all.D", then it would freeze T, and therefore be illegal.

13.14(11) says:

> At the place where a name causes freezing, the entity denoted by the name is frozen, unless the name is a prefix of an expanded name; at the place where an object name causes freezing, the nominal subtype associated with the name is frozen.

AARM 13.14(11.a-11.b) says:

> Ramification: This only matters in the presence of deferred constants or access types; an object_declaration other than a deferred_constant_declaration causes freezing of the nominal subtype, plus all component junk.
>
> Implicit_dereferences are covered by expression.

It seems that **AARM 13.14(11.b)** is wrong. .... [There were several more references to the AARM in Draft 6a]

3. Does an implicit call to Initialize freeze the subprogram? **(Yes.)** The freezing rules seem to apply to explicit constructs. For example:

13.14(12) seems to agree that the implicit conversion should freeze.

> **At** the place where a range causes freezing, the type of the range is frozen**.**

3. An implicit call, such as an implicit call to Initialize, freezes the called subprogram. This is true even if the implicit call is removed via the **implementation permissions** in 7.6(18-21).

Given **the conclusion herein reached**, the above example (1) is illegal. The occurrence of "X.all" freezes the type T, but the type is not completely defined at that point, thus violating 13.14(17). Note that the declaration of Y is an object_renaming_declaration, not an object_declaration, so 13.14(6) does not apply.

The problem occurs in any case where an object name can occur, and is analogous to the expression case in 13.14(8); hence the **resolution** is worded by analogy with 13.14(8).

Since an implicit_dereference is not an expression and is **not a** name **(although** it ...

3 and 4. Clearly implicit calls and implicit conversions should freeze in the same manner as their explicit counterparts. An implicit call should freeze even if it is removed via the **implementation permissions** in 7.6(18-21); otherwise, there would be a portability problem.

DR47:

Integer_Text_IO and Float_Text_IO are not listed in A(2), but Elementary_Functions (for example**) is listed**. Is this intended? (No.)

DR49:

4. **A.4.3**(2) says:

This **resolution** clarifies minor details of the semantics of some of the string-manipulation subprograms:

4. A.4.3(2) **holds throughout A.4.3**.

4. Clearly, the intent is that the lower bound should always be 1, as stated in A.4.3(2). **A** "friendly" reading is that A.4.3(74) is just telling us the characters of the string (it says "comprises", and not "is equivalent to"), and is not intended to define the bounds.

A.4.3(2) is therefore interpreted **to hold throughout A.4.3.**

DR50:

Is it legal to allow some extra flexibility? **(Yes.)** For example, suppose the Image function returns a representation of the state as a string of hexadecimal digits, with 'A'..'F' in upper case. A string with 'a'..'f' in lower case, but which is otherwise equivalent to a valid image, is not strictly speaking "the image of any generator state". May the Value function nevertheless return a valid state for such a string, or must it raise Constraint_Error?

It is a bounded error to invoke Value with a string that is not the image of any generator. If the error is detected, Constraint_Error or Program_Error is raised. Otherwise, a call to Reset with the resulting State will produce a **generator** such that calls to Random with this **generator** will produce a sequence of values of the appropriate subtype, but which might not be random in character. That is, the sequence of values might not fulfill the requirements of A.5.2(41-43).

There is no need to make the situation erroneous -- the implementation shouldn't write **to** random memory locations, or take wild jumps. The worst that can happen is that a non-random sequence of numbers (for example, a sequence of zeros) will be produced.

DR51:

8652/0051 **Text_IO.Flush should use mode 'in'**

It is important to be able to call Flush on the Current_Output, Standard_Output, Current_Error, and Standard_Error files. However, these files are only accessible via function results or dereferencing an access-to-constant value; thus they cannot be **flushed** if the mode is 'in out'.

DR52:

In the definition of the **Text_IO.Standard_Error that returns a value of the type File_Access**, the standard states that the returned access value designates "the standard output file." This is just a **typographical** error, right? (Yes.)

DR54:

8652/0054 Enumeration_IO does not allow instantiation for a float type

A.10.10(17) says: "Enumeration_IO would allow instantiation for an float type". This is obviously a **typographical error**; "integer" is meant.

DR56:

If the function Ada.Streams.Stream_IO.Stream is called with a closed file, does the call raise Status_Error? **(Yes.)**

If the function call does not raise Status_Error, does an attempt to read from or write to the stream **referenced** by the resulting access value raise **Status_Error, if the file has been closed,**? **(It is erroneous.)**

Summary of Response

Ada.Streams.Stream_IO.Stream raises Status_Error if its parameter is not an open file. If the file passed to **the** Stream **function** is closed or ceases to exist after the call on **the** Stream **function** and the Root_Stream_Type'Class object designated by the function result is subsequently passed as the first parameter to Ada.Streams.Read or Ada.Streams.Write, execution is erroneous.

The rules stipulating when use of the result of **the** Stream **function** is erroneous are analogous to the rules in A.10.3(22) and A.10.3(23) for the result of the Current_Input, Current_Output, and Current_Error functions. These rules make it possible to represent a File_Type value or a file stream-value as a pointer, with a null pointer corresponding to a closed file. (By a file-stream value, we mean a value belonging to some descendant of Root_Stream_Type and representing a stream associated with a file.)

An alternative approach would be to allow the result of **the** Stream **function** to correspond to a closed file, but to raise Status_Error upon an attempt to use a file-stream value associated with a closed or no longer existent file. Then a file-stream value would have to reflect the fact that its corresponding internal file had been closed or had ceased to exist. This would rule out an implementation in which closing a file simply sets a File_Type value to a null pointer. Finalization of both File_Type objects and file-stream objects would be complicated. (One possible implementation would be for both File_Type objects and file-stream objects to be controlled objects pointing to an object that includes both a reference count and an is-opened flag. Such an object would be allocated upon creation of a File_Type object, but would continue to exist beyond the lifetime of the File_Type object if there were still unfinalized file-stream objects pointing to it.)

DR57:

The NOTE in A.10.3(25) contradicts A.14(3):

> **(24) The** standard input, standard output, and standard error files are different file objects, but not necessarily different external files.

It is not clear that A.14(3) is trying to say anything in addition to what A.14(2) already says for all text files. Therefore, A.14(3) should simply be **removed**.

DR59:

The problem with this is that if one has a C function that is passed a struct, then how can one pass an Ada record to that? One might think that if the Ada record is passed as an 'in' parameter, it will work. However, the above **implementation advice** implies that such an 'in' parameter will correspond to a t* on the C side, rather than a t.

The **implementation advice** in B.3(69) is correct as written.

The **implementation advice** in B.3(69) is left unchanged (that is, C-compatible records are passed by reference).

The following sentence is added to the **implementation advice** in B.3(64-71):

DR60:

The following declarations appear in Interfaces.C **(B.3)**:

```
19     type char is <implementation-defined character type>;
```

Same for wchar_t. The fonts should be right.

In the Summary of Response: „implementation{-}defined“

DR61:

Add a Discussion sentence:

This obvious omission is hereby corrected (see Summary of Response).

DR62:

Any attempt to create a null array of type char_array, whose lower bound is 0, will clearly raise Constraint_Error. Therefore, "Value(Item => X, Length => 0)" will raise Constraint_Error. (The **standard** should have made this more explicit, however.)

DR63:

8652/0063 Interfaces.C.Strings.Value with Length returning String

Validation test **cxb3011** in suite 2.1 seems to presume that no Terminator_Error should be raised when the input chars_ptr does not have a null within the specified length.

DR68:

C.3.1(7-8) says:

> **The** Attach_Handler pragma is only allowed immediately within the protected_definition where the corresponding subprogram is declared. The corresponding protected_type_declaration or single_protected_declaration shall be a library level declaration.
>
> **The** Interrupt_Handler pragma is only allowed immediately within a protected_definition. The corresponding protected_type_declaration shall be a library level declaration. In addition, any object_declaration of such a type shall be a library level declaration.

AARM C.3.1(7.a) says:

> Discussion: In the case of a protected_type_declaration, an object_declaration of an object of that type need not be at library level.

Thus, nested objects are not allowed in the Interrupt_Handler case, but they are allowed in the Attach_Handler case.

C.3.1(12) says:

> **When** a protected object is finalized, for any of its procedures that are attached to interrupts, the handler is detached. If the handler was attached by a procedure in the Interrupts package or if no user handler was previously attached to the interrupt, the default treatment is restored. Otherwise, **that is, if an Attach_Handler pragma was used,** the previous handler is restored.

AARM C.3.1(12.a) says:

> **Since** only library-level protected procedures can be attached as handlers using the Interrupts package, the finalization discussed above occurs only as part of the finalization of all library-level packages in a partition.

Option 1: require every protected object with an Attach_Handler pragma to be at library level. This is clearly not what the **standard** says. It doesn't completely solve the problem, either -- one could create two objects on the heap, and **call an instance of Unchecked_Deallocation on** them in a non-LIFO order.

DR69:

Version of 28.09.00

Moreover, the intent of the **standard** is to allow user-specified handlers that are not parameterless protected procedures, as specified in C.3 (26)

It might be nice if there were a "portable" way for an application to determine whether the value returned in Old_Handler or by Current_Handler is one that can be dereferenced to call a parameterless protected procedure, as in the handler-cascading model. The **standard** itself does not explicitly specify how this can be determined, but it can be interpreted in a way that may suffice.

DR71:

C.7.2 (16) says, in full, "The implementation shall perform each of the above operations for a given attribute of a given task atomically with respect to any other of the above operations for the same attribute of the same task."

The use of attribute handles is not protected by any atomicity requirement in the **standard**, so that their concurrent use must be deemed erroneous.

DR72:

D.4(8-11) supports the above as the "intent"**. In particular , D.4(11) says**:

> **When** the base priority of a task is set (see D.5), if the task is blocked on an entry call, and the call is queued, the priority of the call is updated to the new active priority of the calling task. This causes the call to be removed from and then reinserted in the queue at the new active priority.

Remove unnecessary paragraph numbers from these paragraphs:

> **If** a task becomes held while waiting (as a caller) for a rendezvous to complete, the active priority of the accepting task is not affected.
>
> **When** a task calls a protected operation, a check is made that its active priority is not higher than the ceiling of the corresponding protected object; Program_Error is raised if this check fails.
>
> **Setting** the task's base priority to the new value takes place as soon as is practical but not while the task is performing a protected action.

DR73:

8652/0073 Pragma Locking_Policy **cannot be in** a **program unit**

DR74:

The wording for D.4(1) should be corrected.

DR75:

The language designers did not want D.4(11) to apply in this case. This is supported by the use of the phrase "if the task is blocked on an entry call" in D.4(11) -- the task making a triggering entry call is not blocked. It is further supported by the lengthy discussion in the AARM on this very topic (AARM **D.4(11.a-11.f)).**

DR79:

The implementation requirements of E.2(13) says:

> For a given library-level type declared in a preelaborated library unit or in the declaration of a remote types or remote call interface library unit, the implementation shall choose the same representation for the type upon each elaboration of the type's declaration for different partitions of the same program."

This seems overly restrictive. It means that **the standard** supports heterogeneous distributed systems only if the implementation manages to use the same representation for a type on all nodes. Is this intended? (No.)

E.2(13) requires the "same representation" in all partitions. This requirement prevents heterogeneous distributed systems, and is not needed, so it is deleted.

DR81:

This seems to imply that a remote access type has no storage pool, which is confirmed by **AARM E.2.2(17.a):**

Remove unnecessary paragraph number in Question.

Normally, a derived access type has the same storage pool as its parent. See **DR-0012**, which confirms NOTE 3.4(31). However, the intent of E.2.2(17) is that a remote access type has no storage pool. Therefore, a type derived from a remote access type cannot have a storage pool, either. Querying 'Storage_Pool and 'Storage_Size should be illegal by E.2.2(17). Similarly, allocators should be illegal.

DR82:
There were two issues **raised**:

DR87:

A(4) says:

> **The** implementation may restrict the replacement of language-defined compilation units. The implementation may restrict children of language-defined library units (other than standard).

DR88:

The String "++++>" and like Strings with '>' unmatched by any '<' appear to be valid picture strings based on the following production sequence **(F.3.1)**:

DR89:

Add a Summary of Response and Discussion sections:

Summary of Response

The example in F.3.2(74) has been corrected.

Discussion

The example was incorrect.

DR90:

8652/0090 Should "pragma" be in boldface?

Should "pragma" be in boldface? (Yes.)

"**Pragma**" should be in boldface.

DR91:

G.1.1(55) gives the following method for doing complex exponentiation in polar **form**:

Assume that the **method described in the standard** is correct. Let a complex number X=0+I. Let an integer n=-1

DR93:

8652/0093 Only the current unit is affected by the Pragma Inspection_Point

In the standard, pragma Inspection_Point is not a configuration pragma. However, consider the following example:

DR94:

The Qualifier on the Cover Page should be „Clarification requested“.

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).

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

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 ... ".

DR95:

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

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

DR96:

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

DR 97:

Add at the end of the summary: **Hence, it can not declare a Controlled type.**

DR98:

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

DR99:

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.

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)**.

DR101:

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

Ada83 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.

DR102:

The Qualifier on the Cover Page should be „Clarification requested“.

DR103:

There is a general **design principle in Ada** that one can never have more visibility into the components or operations of a type than in the package where the type is declared.

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 above example is illegal, because in this **case Child** .....

DR104:

Summary of Response

It is possible (and desirable) for an implementation to allocate and initialize type descriptors statically.

DR105:

The Qualifier on the Cover Page should be „Clarification requested“.

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

DR106:

„case“ all identifiers in the example in the normal Reference Manual style.

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.

An abstract subprogram can be renamed, and the renamed view is also abstract. Such a renaming must appear in a place where the declaration of an abstract subprogram would be legal. Similarly, the "shall be overridden" property of 3.9.3(6) applies to a renamed view. **Thus, any renaming of an inherited subprogram that must be overridden is illegal.**

DR107:

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?

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.

DR108:

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.)

DR109:

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.

DR110:

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.)

Delete the last sentence of the Response.

DR111:

Duplicate the Summary to the Response.

Response

If a generic formal package B whose actual part is (<>) is passed as an actual to another generic formal package A without (<>), then 12.7(5-8) requires the actuals of B to match the actuals of A. For the purpose of this rule, the actuals of B are the entities denoted by names of the form B.x, where x is a generic formal parameter of B.

12.7(5-8) require the actual parameters of the actual package to match the actual parameters of the formal package. But if the actual package has (<>), then its actual parameters are denoted by the formal parameters.

Note that we cannot defer the check until instantiation time, because I3 could be in the *body* of G2 (instead of in the **specification**, as shown above), and this would constitute a contract model violation.

DR112:

Duplicate the summary to the response.

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 illegal if the implementation chooses small **to be** System.Fine_Delta.

DR114:

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.)

The specification of a user-defined storage pool for an **access-to-task** type permits the user-defined allocation and deallocation of task objects, not necessarily **of** the implementation-defined tasking structures that support the task object.

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.

DR115:

Duplicate the summary to the response.

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."

DR116:

The Qualifier on the Cover Page should be „Clarification requested“.

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?

Add this paragraph to the end of the Response:

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.

DR117:

B.3 (46) states that To_Ada and To_C map between Character and char, but does not explain how.

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

Don't forget to make this a note in the AARM one day.

DR118:

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.)

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.

DR120:

E.2.2 (14) says for a remote access-to-classwide type, "The primitive subprograms of the corresponding specific limited private type shall only have access parameters if they are controlling formal parameters...."

The title should be : Intent of „only“

There are superflous blanks in the question.