Final Minutes of 9th ARG Meeting

27-30 September 1999

Westborough, MA ,USA

Attendance at the last ARG meeting of the 2. millennium: John Barnes, Randy Brukardt, Gary Dismukes, Bob Duff (Tuesday-Thursday), Robert Eachus, Kiyoshi Ishihata, Mike Kamrad, Pascal Leroy, Erhard Ploedereder, Ed Schoenberg (Wednesday-Thursday), Tucker Taft, Joyce Tokar (Thursday). **Observer**: Matt Heaney.

Meeting Summary

The meeting convened on 27 September 99 at 09:00 hours at the Top Layer offices in Westborough MA and adjourned at 15:00 hours on 30 September.

The first two days of the meeting were devoted to reviewing the format and content of the current draft of the Technical Corrigendum. All the wording changes for the core language were reviewed and the wording for them were changed as appropriate.

The last two days were devoted to reviewing the amendment AIs and to analyzing the various implementation-defined pragmas and attributes for inclusion in the Standard.

The ARG unanimously thanked Mike Kamrad and Top Layer for their generous hosting of the meeting.

AI Summary

Only these amendment AIs were discussed. They all require rewriting for further discussion or vote:

AI-216 - Unchecked Unions -- Variant Records with no Run-Time Discriminants
AI-217 - Handling Mutually Dependent Type Definitions that Cross Packages
(This AI will be split into three AIs - see detailed discussion)
AI-218 - Accidental Overloading When Overriding
AI-222 - Feature Control

Report on WG9 Meeting

All AIs that were presented for WG9 approval were accepted except the ones on Finalize/Initialize issues (AI-147 and AI-197). The Swedish objection is based mainly on a dislike of the requirement to build new values in the target objects without prior initialize call and to deviate from a canonical model that pairs finalize with initialize calls. The argument that this was already case in Ada95 without the AIs did not help. Erhard scheduled time during the discussion of Section 7.6.1 of the Technical Corrigendum to address these objections. Tuck volunteered to prepare a tutorial/presentation that will address these objections.

Next Meeting

The next meeting will be in Phoenix at the DDC-I offices on 27-29 March 2000. (Good alternatives had included Paris, Copenhagen and Boston; we might consider this for the next meetings.)

Action Items

Many existing action items have been accomplished but old action items remain:

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
Bob Duff: AI-51, AI-109
Robert Eachus: AI-100, AI-172, AI-174, AI-185, AI-186

Stephen Michell: AI-148, AI-187
Erhard Ploedereder: AI-209, AI-212
Tucker Taft: AI-130, AI-162, AI-188, AI-189, AI-191, AI-193, AI-195(2.part), AI-199

New action items on the drafting or completion of AIs were assigned to the following ARG members:

Randy Brukardt: AI-218, AI-222, new AIs on Unsuppress and on 'Object_size
Gary Dismukes: AI-171 (rewrite wording),
Mike Kamrad: new AI on Assert pragma
Pascal Leroy: AI-168, AI-186(?), Streams, AI-108, AI-137/DR 40 + AI-117/DR 10 (partially)
Erhard Ploedereder: AI-41/DR-32 wording, AI-171 (rewrite wording)
Tucker Taft: AI-216, AI-217, AI-147 and -197 (provide rationale, see minutes on Corrigendum after7.6)

The following action items were also assigned:

All: Review and comment on Annex parts of Technical Corrigendum
All: Create tests for assigned AIs
Bob Duff: Be the test creator of last resort, distribute Design Note on „inward closures"
Randy Brukardt: Update the next draft of the Corrigendum with the many detailed changes recorded at the meeting and in these minutes; distribute „inward-closures" study from prototyping project
Joyce Tokar: Publish logistics for the next meeting

Details of Technical Corrigendum Review

General Comments on Corrigendum: Jim Moore informed Randy that the Technical Corrigendum must be accompanied by an updated RM. Unfortunately the original RM is a Scribe document for which there are few Scribe processors around to produce an updated version. An alternative would be to translate the Scribe into Microsoft Word RTF. Another alternative would be an HTML version of the RM that could also capture all the information/differences found in the AIs; the HTML version, created by Magnus Kempe, could be a start point. One of these alternatives should be capable of producing a version suitable for mass printing. However, none of these alternatives will be cheap, as they all require manual labor or license fees. Randy will explore this issue further.

On the issue of the copyright, there will be a solution along the lines that the US delegation to WG9 will offer a version, already copyrighted and with broad redistribution permissions, to ISO for the standardization process. This version needs to be good enough that no further changes are necessary. The Corrigendum is handed out for free by ISO along with the (old) Standard. The above method will ensure that an integrated reference manual can be produced without getting into copyright conflicts.

Randy looked for directions for conventions in identifying new and replacement paragraphs. After some semi-serious discussion, it was decided

- Deletions should require no notation.
- Existing paragraphs should be considered version 0, designated as „V0", such as 3.5.10 (2 V0). A missing version number is always to be interpreted as V0.
- A replacement paragraph would be designated as „V1", such as 3.5.10 (2 V1)
- Inserted paragraphs would be designated with an additional point and number, such as 3.5.10 (2.1 V1) for a new paragraph after 3.5.10 (2).

There was some inconclusive discussion whether one should write „2.1 V1" or „2.1/2". The former is more readable, but may create problems in formatting a new RM. The sense of the discussion seemed to indicate that this will be left to Randy to decide.

Randy described his logic for inclusion of AIs in the Corrigendum. All the binding interpretations were included and presentation AIs were not. The presentation issues are actually down to about 7. Randy believes that some confirmation and ramification AIs are significant enough to be reflected in the Corrigendum by actual wording changes and he asked for directions from ARG on including selected confirmation and ramification AIs for the Corrigendum. We concluded that Randy should add a designation to all AIs to indicate whether they get put into the

Corrigendum and identify the Defect Report number for the AI (to map between the ISO identification notation and our own identification notation).

Also, Defect Reports carry a certain ISO qualifier on their disposition. Randy should include this qualifier in all the AIs, so that it will appear automatically in the Defect Reports.

We toyed (again) with the idea that amendment AIs could be interpreted as defects, since the lack of the feature in an amendment AI could be considered a form of language defect. We discarded that idea (again) because amendment AIs should be treated as significant changes and not swept under the table as a defect. Of course, getting language amendments standardized appears to be a more arduous process compared to a mere Corrigendum.

Many members complained that it is hard to distinguished changes made by new wording of the Corrigendum, especially for the purposes of proof-reading. It was decided that the nature of the Corrigendum document, as mainly an ISO procedural requirement, did not warrant any more investment to address this problem. Instead this investment should be made in the HTML Ada Reference Manual, described above.

Detailed Review of Individual Corrigendum Items

This section describes the changes that were recommended to the new wording of each item listed in the core language Corrigendum. Only those paragraph whose new wording was changed are listed here; consider all others as being unchanged or so seriously affected by the recorded observations that a change at the meeting was not feasible. Each of the changes are listed by their paragraph number and Defect Report number. Where the changes are minor, they are noted by underlined, emboldened font for easy identification.

The motivation for many of the changes was to be more concise and precise and this is the implied reason for the listed changes when no explicit reason is given.

3.3.1 (18 V1)/0002/AI-171

While there is no wording change (yet), the associated defect report does not include recent contributions by Gary Dismukes. A reference to 3.7.1. should be added in lieu of the last sentence. Erhard and Gary will look at the AI again to see whether the current wording covers all bases.

3.5.4 (27.1 V1)/0003/AI-95

> „For a one's complement machine, an implementation may support non-binary greater than System.Max_Nonbinary_Modulus. It is implementation defined which specific values greater than System.Max_Nonbinary_Modulus, if any, are supported."

3.5.8 (2 V1)/0004/AI-203

In both the original wording and the new wording, put the following phrases in italics: universal_integer and the symbols d and g.

3.7.1 (7 V1)/0008/AI-168

Instead of adding a new paragraph, the following new wording should be added to the end of this paragraph:

> However, in the case of a general access subtype, the designated subtype shall not be of a type whose partial view is constrained.

A lot of effort was spent at the meeting in parsing and reconstructing the meaning of the paragraph. Bob showed an example that doesn't seem to be handled either by the original or new wording. New wording may require improvements in the definition of partial view. Pascal, as author, will investigate and produce new wording. Hopefully, no significant change in the concept of partial view will be needed.

The AI needs to take care of derived types as well. It goes back to Pascal for updates.

3.8 (18 V1)/0002/AI-171

See minutes for 3.3.1(18). They apply here, too.

3.9.2 (7 V1)/0009/AI-127

The wording discussion focused on uniformly applying the directions of the AI. The compelling reason for the changes in the new wording is illustrated by this snippet of code

```
procedure p (x: access tt);
Y : tt'class := …;
P (Y'access);
```

The new wording is:

> 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 of a form new T'(...), where T denotes a class-wide subtype. Otherwise, the object is statically or dynamically tagged according to whether the designated type of the type of the expression is specific or class-wide, respectively.

3.9.2 (9 V1)/0009/AI-127

Replace the word „designates“ in the last sentence with the word „is“.

3.10 (7 V1)/0011/AI-62

Replace the sentence:

> A derived access type shares the storage pool of its parent type.

with

> All descendants of an access type share the same storage pool.

3.10.2 (27 V1)/0009/AI-127

> „If *A* is a named access type and *D* is a tagged type, then the type of the view shall be covered by *D*; if *A* is anonymous and *D* is tagged, then the type of the view shall be either *D*'Class or a type covered by *D*; if *D* is untagged, then the type of the view shall be *D*, and A's designated subtype shall either statically match the nominal subtype of the view, or shall be discriminated and unconstrained;“ (typographical changes)

4.1.4 (12 V1)/0014/AI-93

> „An implementation may provide implementation-defined attributes; the identifier for an implementation-defined attribute shall differ from those of the language-defined attributes unless supplied for compatibility with **a** previous **edition** of this standard.

4.5.2 (24 .1 V1)/0015/AI-123

The new wording is to be inserted after paragraph 24 as a non-bulleted paragraph, because it is a better fit there.

4.5.2 (32 .1 V1)/0015/AI-123

„For all non-limited types declared in language-defined packages, the „=“ operator of the type shall behave as if it were the predefined equality operator for the purposes of composite equality and generic formal equality.“

4.6 (12 V1 and 12.1 V1)/0008/AI-168

Replace the existing paragraph with this new wording:

- the component subtypes shall statically match, and
- in a view conversion, the target type and the operand type shall both or neither have aliased components.

4.6 (54 V1)/0016/AI-184

The following change was made:

„If the target type is composite, the bounds or discriminants (if any) of the view are as defined above for a value conversion; each non-discriminant component of the view denotes the matching component of the operand object; the subtype of the view is constrained if either the target subtype or the operand object is constrained, or if the operand type is a descendant of the target type and has discriminants that were not inherited from the target type**, or if the target type is indefinite**;“

4.8 (3 V1)/0009/AI-127

After a re-discussion of the merits of AI-127, which did not bring additional insights, the paragraph was editorially improved to make it more readable:

„The expected type for an allocator shall be a single access-to-object type with designated type *D* such that either *D* covers the type determined by the **subtype_mark** of the **subtype_indication** or **qualified_expression**, or the expected type is anonymous and the determined type is *D'Class*.“

6.3.1 (13.1 V1)/0010/AI-117

The following changes to the new wording included dropping reference to the „full type of the type“ in the last phrase of the paragraph. It was argued that 13.1. makes it clear that representational properties (of which Convention is one as per B.1) are inherited by derived types and that incomplete types cannot have different representational properties than their full types. Thus, this need not be specified again. Whether a Note here or in 13.1 would nevertheless be beneficial was left open. There was sympathy with the perception that a statement in 13.1. (around (11)) might be worthwhile to the effect that „The representational aspects of a partial view are the same as the representional aspects of the corresponding full view.“

„If not **specified as intrinsic**, the calling convention for any inherited or overriding dispatching operation of a tagged type is that of the corresponding subprogram of the parent type. The default calling convention for a new dispatching operation of a tagged type is the convention of the type.“

7.3.1 (6 V1)/0018/AI-33

„Inherited primitive subprograms follow a different rule. For a derived_type_definition, each inherited primitive subprogram is implicitly declared at the earliest place, if any, within the declarative region in which the type_declaration **occurs**, but after the type_declaration, where the corresponding declaration from the parent is visible. If there is no such place, then the inherited subprogram is not declared at all. An inherited subprogram that is not declared at all cannot be named in a call and cannot be overridden, but for a tagged type it is possible to dispatch to it.“

7.6 (4 V1)/0019/AI-126

No changes to the new wording were made. It was decided that, unlike the style of other items in the Corrigendum where the whole paragraph which is changed is presented, it is reasonable for changes to the coding examples to only focus on the actual changes made.

7.6 (11 V1)/0020/AI-182

> „For an extension_aggregate whose ancestor_part is a subtype_mark, for each controlled subcomponent of the ancestor part, either Initialize is called, or **its** initial value is assigned if the type of the ancestor part is itself controlled, the Initialize procedure of the ancestor type is called, unless that Initialize procedure is abstract."

7.6.1 Completion and Finalization

Erhard spent time explaining the Swedish objection: the dislike of a requirement to build values in the target objects (without first calling the initialization routine) and to clearly match finalizes with initializes. An example of the latter issue is:

```
Type t is new tp with …;
X : t := t'(tp with …);
```

Where is the matching finalize for the initialize of the ancestor part tp? (A finalization of X uses t.finalize, which really is a poor match for the tp.initialize.) When is it called ? Certainly not at the completion of the enclosing expression. Current semantics seems to imply no call for finalize on the ancestor part. Responsibility would rest with the programmer to make an explicit Finalize call on the ancestor part of the object within Finalize for t. Neither the AI nor the RM says anything sensible about this case. Tuck has volunteered to summarize the essentials for responding to the Swedish objections.

The other Swedish issue of absolutely guaranteeing that initialize gets called regardless of an explicit initialization is a no-starter. Such a semantics, although very „canonical" in its pairing of initialize/finalize, is really contentious. Why initialize and then immediately wipe out all effects on the initialized objects by the explicit initialization ?

7.6.1 (13 V1)/0020/AI-182 & 0021/AI-169

> „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, they are finalized before starting the execution of any **statement** within the compound_statement. 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."

7.6.1 (17.1-2 V1)/0021/AI-169

The new wording for the inserted paragraphs are:

- For a Finalize invoked as part of the finalization of the anonymous object created by a function call or aggregate, any other finalizations due to be performed are performed, and then Program_Error is raised.
- For a Finalize invoked due to reaching the end of execution of a master, any other finalizations associated with the master are performed; Program_Error is raised immediately after leaving the master.

7.6.1 (20.1 V1)/0023/AI-83

Many elements of AI-197 appeared in this discussion and eventually affected the wording for this new paragraph. The following instances of controlled aggregates were enumerated and only two instances where Adjust is necessary, which are marked:

x : T := (*aggregate*);

x : AT := new T'(*aggregate*);

...(... (*aggregate*)...)

x := (*aggregate*); -- Adjust is called

P ((*aggregate*));

return (*aggregate*); -- Adjust is called

package Foo is new Z (...(*aggregate*)...);

Luckily, many of the needed terms were found in the RM to make changes to the new wording.

The new inserted paragraph is:

*Implementation Requirement**s***

For an aggregate of a controlled type whose value is assigned other than by an assignment or return_statement, the implementation shall not create a separate anonymous object for the aggregate. The aggregate value shall be constructed directly in the target of the assignment operation and Adjust is not called on the target object.

8.3 (9 V1)/0024/AI-44

„Two homographs are not generally allowed immediately within the same declarative region unless one *overrides* the other (see Legality Rules below). The only declarations that are *overridable* are the implicit declarations for predefined operators **and** inherited primitive subprograms. A declaration overrides another homograph that occurs immediately within the same declarative region in the following cases:"

8.3 (26 V1)/0025/AI-150

Tuck wants to modify this paragraph to reflect the changes in the replacement paragraphs 8.3 (9 V1) and 8.3 (10 V1), namely to replace the terms „implicit" and „explicit" with the new terms „overridable" and „non-overridable", respectively.

The other significant change to this paragraph is in the second sentence where the reference, „two components", is ambiguous and fails to bind to the component declarations in question. That sentence was replaced.

Other changes were considered.

Additionally, as part of the effect of [0024], there was a new wording proposal to replace the last two sentences; the proposed new wording also would avoid mixing the terms overridable with overloadable. The proposal was dropped when Bob pointed out from wording in the AARM that the proposed new wording failed to cover cases that the original wording did.

Tuck's attempts to change the first sentence failed when Erhard pointed out that the current version does handle the debated case.

The new wording is:

An **nonoverridable** declaration is illegal if there is a homograph occurring immediately within the same declarative region that is visible at the place of the declaration, and is not hidden from all visibility by the **nonoverridable** declaration. **In addition, a type extension is illegal if somewhere within its immediate scope it has two visible components with the same name.** Similarly, the context_clause for a subunit is illegal if it mentions (in a with_clause) some library unit, and there is a homograph of the library unit that is

visible at the place of the corresponding stub, and the homograph and the mentioned library unit are both declared immediately within the same declarative region. These rules also apply to dispatching operations declared in the visible part of an instance of a generic unit. However, they do not apply to other overloadable declarations in an instance; such declarations may have type conformant profiles in the instance, so long as the corresponding declarations in the generic were not type conformant.

8.5.1 (5 V1)/0016/AI-184

There is a real problem in the last sentence with the long series of adjectives in the phrase, „untagged indefinite generic formal derived type". Consequently we began a search for a more understandable statement, which resulted in this new wording:

> The renamed entity shall not be a subcomponent that depends on discriminants of a variable whose nominal subtype is unconstrained, unless this subtype is indefinite, or the variable is aliased. A slice of an array shall not be renamed if this restriction disallows renaming of the array. In addition to the places where Legality Rules normally apply, **these rules apply** also in the private part of an instance of a generic unit. **These rules also apply for a renaming that appears in the body of a generic, but with the additional requirement that even if the nominal subtype of the variable is indefinite, its type shall not be a descendant of a untagged generic formal type.**"

8.5.4 (5 V1)/0026/AI-135 & 0027/AI-145

There was dissatisfaction with the last sentence of the new wording, because of its lack of precision. Two alternative sentences were considered:

> Alternative 1
>
> A renaming-as-body is illegal if the subprogram it declares names the subprogram itself or renames a subprogram that takes its convention from the subprogram it declares.
>
> Alternative 2
>
> It is illegal for a renaming-as-body that occurs before the subprogram it declares is frozen to rename itself either directly or indirectly through one or more renamings, none of which have been frozen.

Alternative 1 succinctly describes the symptom of the problem but a description of symptoms is improper as a legality rule. Alternative 2 creates a recursive definition when it uses the phrase „to rename itself". Also „directly or indirectly" were considered too vague.

After much discussion this topic was tabled with no action item.

8.5.4 (8 V1 & 8.1-2 V1)/0013/AI-64 & 0026/AI-135

The first paragraph seems disjoint from the summary of response for [0013] because there is no explicit mention of an elaboration check. The connection is found associated with the implied elaboration check in the „simply calls" phrase.

The following example illustrates the motivation for the changes in this Corrigendum item:

```
package
   type T is private;
   function Sub (x, y : Natural) return Natural;
   ...
private
   type T is new Integer;
   function Sub (x, y : Natural) return Natural renames „-";
      -- surprise!!  Sub may return negative values
      -- which need extra checks on parameter passing or return values
```

There is an efficiency concern that the existing wording won't permit the elimination of extra constraint and elaboration checks.

The new wording for the changed paragraphs are:

> For a call to a subprogram whose body is given as a renaming-as-body, the execution of the renaming-as-body is equivalent to the execution of a **subprogram_body** that simply calls the renamed subprogram with **its formal parameters as its actual parameters and, if it is a function, returns the value of the call**.
>
> For a call on a renaming of a dispatching subprogram that is overridden, if the overriding occurred before the renaming, then the body executed is that of the overriding declaration, even if the overriding declaration is not visible at the place of the renaming; otherwise, the inherited or predefined subprogram is called.
>
> *Bounded (Run-Time) Errors*
>
> If a subprogram directly or indirectly renames itself, then it is a bounded error to call that subprogram. Possible consequences are that Program_Error or Storage_Error is raised, or that the call results in an **infinite** recursion.

There was some concern about the bounded error ruling. Robert Eachus as the author was unsure and will reconsider this issue.

9.1 (9.1 V1)/0028/AI-116

> „For a task declaration without **a** task_definition, a task_definition without task_items is assumed."

9.5.2 (22 V1)/0002/AI-171

The AI-171 (Elaboration of subtype_indications with per-object constraints) in Defect Report 0002 affects this paragraph as well as 3.3.1 (18 V1) and 3.8 (18 V1). Gary/Erhard's action item to check the wording and produce new one, in particular for paragraph 3.8 (18 V1), will ensure that the new wording will work here, too. There was a change to the new wording to remove the „or evaluation" phrase as it is not defined. The new wording is:

> The elaboration of an entry_declaration for an entry family consists of the elaboration of the discrete_subtype_definition as described in 3.8. The elaboration of an entry_declaration for a single entry has no effect.

It was also determined that it affects 3.6 (22 V0) which discusses discrete_subtype_definition. Consequently, this paragraph is replaced with 3.6 (22 V1):

> The elaboration of a discrete_subtype_definition **that does not contain any per-object expressions** creates the discrete subtype, and consists of the elaboration of the subtype_indication or the evaluation of the range. **The elaboration of a** discrete_subtype_definition **that does contain one or more per-object expressions is defined in 3.8.** The elaboration of a component_definition in an array_type_definition consists of the elaboration of the subtype_indication. The elaboration of any discrete_subtype_definitions and the elaboration of the component_definition are performed in an arbitrary order.

9.10 (7.1 V1)/0030/AI-118

> „If A1 is the termination of a task T and A2 is the evaluation of the expression T'Terminated or a call to Ada.Task_Identification.Is_Terminate, where the actual parameter identifies T."

10.1.4 (4 V1)/0031/AI-192

> „If a library_unit_body that is a subprogram_body is submitted to the compiler, it is interpreted only as a completion if a library_unit_declaration **with the same defining_program_unit_name already exists**

in the environment for a subprogram other than an instance of a generic subprogram or for a generic subprogram (even if the profile of the body is not type conformant with that of the declaration); otherwise the subprogram_body is interpreted as both the declaration and body of a library subprogram."

10.1.5 (2.1 V1)/0032/AI-41

The subject of the last sentence in the last paragraph is best expressed as a rule of Static Semantics, which leads to the following paragraph being inserted after 10.1.5 (2 V0). Additionally, this simplifies the wording of Implementation Advice, found in paragraph 10.1.5 (9.1 V1).

Static Semantics

By default, a library unit pragma that applies to a generic unit does not apply to its instances.

Erhard will check the ramifications of the AI again, especially in view of the other AI of pragmas that do apply to subprogram instances.

10.1.5 (9.1 V1)/0032/AI-41 & 0034/AI-199

With the new Static Semantic rule described above, the wording is simplified to:

Implementation Advice

Any other program unit pragma that applies to a generic unit applies to an instance of the generic unless the instance has an overriding pragma.

10.2.1 (11 V1)/0035/AI-02

AI-02 is not quite complete because it only handles „immediate" subunits that are subprograms; it fails to handle subprograms that are nested within other subunits like tasks or packages. Hence the wording is changed to:

The declaration and body of a preelaborated library unit, and all subunits that are elaborated as part of elaborating the library unit, shall be preelaborable.

12.3 (14 V0)/0032/AI-41

The change to 12.3 (14), requested by AI-41 seems unnecessary. RM 8.6(18) can be quoted to answer the fine detail that this change answers here.

AI-41 should be fixed accordingly.

12.5 (8 V1)/0037/AI-43

Changes to the new wording is needed to avoid confusion that the joining of elementary and composite types caused in the draft new wording. The new wording is:

The formal type also belongs to each class that contains the determined class. The primitive subprograms of the type are as for any type in the determined class. For a formal type other than a formal derived type, these are the predefined operators of the type**. For an elementary formal type, the predefined operators are implicitly declared immediately after the declaration of the formal type. For a composite formal type, the predefined operators are implicitly declared either immediately after the declaration of the formal type, or later in its immediate scope according to the rules of 7.3.1.** In an instance, the copy of such an implicit declaration declares a view of the predefined operator of the actual type, even if this operator has been overridden for the actual type. The rules specific to formal derived types are given in 12.5.1.

12.7 (8.1 V1)/0039/AI-213

The draft new wording incorrectly covers formal objects of in out mode, which are already treated as a renaming. The new wording is:

> For the purposes of matching, any actual parameter which is the name of a formal object **of mode in** is replaced by the formal object's actual expression (recursively).

13.1 (10+11 V0)/0040/AI-137

In the discussion of this wording, it was decided to create a new class of attributes, operational attributes. This class should include the stream attributes (and other future user-specifiable attributes) to avoid the pervasive exclusive wording that otherwise would need be made to handle the problems that Stream attributes present, if classified as representational attributes.

Additionally, new wording must be added to this section about the convention of a partial view getting the convention of its full view as part of the inheritance of representation aspects. (also see Minutes under 6.3.1(13), DR 10/AI-117)

Consequently, there will be new wordings for paragraphs in this Section.

Pascal will draft the changes this causes to this section and others.

13.12 (8.1-5 V1)/0042/AI-130

The new wording is considered to be an Implementation Permission and therefore should precede paragraph 9.

> For the purpose of checking whether a partition contains constructs that violate any restriction (unless specified otherwise for a particular restriction):
>
> - Generic instances are logically expanded at the point of instantiation;
> - If an object of a type is declared or allocated and not explicitly initialized, then all expressions appearing in the definition for the type and any of its ancestors are presumed to be used;
> - **A default_expression for a formal parameter or a generic formal object is considered** to be used **if and** only if the corresponding actual parameter is not provided in a given call or instantiation.
>
> *Implementation Permissions*
>
> **Implementations are permitted to omit restriction checks** for code **that** is recognized at **compilation**-time **to be** unreachable and for which no object code is generated.

13.12 (9.1 V1)/0043/AI-190

The phrase „but left to implementation-defined behavior of dynamic semantics“ is dropped, because it is not well-defined. The new wording is:

> Whenever enforcement of a restriction is not required prior to execution, an implementation may **nevertheless** enforce the restriction prior to execution of a partition to which the restriction applies, provided that every execution of the partition would violate the restriction.

13.13.2 (9 V1)/0045/AI-108

> 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 a canonical order. The canonical order of components 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 nonlimited type extensions, the Write or Read attribute for the parent type is called, followed by the Write or

Read attribute of each non-inherited component, in canonical order. For **other** derived types, the Write or Read attribute **of** the parent type is **inherited**.

13.13.2 (27.1 V1)/0045/AI-108

„If *T* is a type **extension**, S'Output and S'Input are not inherited from the parent type; they are defined in terms of S'Read and S'Write, notwithstanding the inheritance rule stated in 13.1."

13.13.2 (35.1 V1)/0046/AI-132

„In the default implementation of Read and Input for a type, End_Error is raised if the end of the stream is reached before **the reading of a value of the type is completed**."

13.13.2 (36.1 V1)/0045/AI-108

„For every subtype S of a language-defined nonlimited specific type *T*, the output generated by S'Output or S'Write **shall** be readable by S'Input or S'Read, respectively. The object read by S'Input or S'Read **shall** behave as the original object for the operations declared in the **language-defined** descendants of the unit that declares *T*. This rule applies across partitions if the implementation conforms to the Distributed Systems Annex.

For every subtype S'Class of a language-defined class-wide type T'Class, the attributes S'Class'Write, S'Class'Read, S'Class'Input and S'Class'Output **shall** have their predefined definition."

This DR was then tabled and assigned to Pascal for further handling in the context of the overall overhaul of Streams and Stream attributes.

13.14 (11.1 V1)/0047/AI-106

The suggested replacement paragraph incorrectly mixes the categories „name" and „implicit_dereference". Consequently the existing paragraph should remain unchanged and an additional paragraph discussing implicit_references needs to be inserted after the existing paragraph and „implicit_dereference" needs to be added to the last sentence of (4).

The new wording for the inserted paragraph is:

- At the place where an implicit_dereference causes freezing, the nominal subtype associated with the implicit_dereference is frozen.

And the new wording for paragraph 13.14 (4 V1) is:

A construct that (explicitly or implicitly) references an entity can cause the freezing of the entity, as defined by subsequent paragraphs. At the place where a construct causes freezing, each name, **implicit_dereference,** expression, or range within the construct causes freezing:

13.14 (15.1-2 V1)/0047/AI-106

The first sentence in the first inserted paragraph is too narrowly specified; it certainly overlooks attributes. Hence it should be broadened. The new wording is:

An implicit call to **any subprogram** freezes the same entities that would be frozen by an explicit call. This is true even if the implicit call is removed via the Implementation Permissions in 7.6.

If an expression is **implicitly** converted to a type or subtype *T*, then at the place where the expression causes freezing, *T* is frozen.

Detailed Review of the Amendment AIs

AI-216

Tuck claims that it is common coding practice in C to mix structs and unions which drives his proposal (vis-a-vis the GNAT model, in which only two existing types can be combined in a union).

The distinction between limited and non-limited records has been eliminated to avoid problems that the last meeting identified.

Interacting with derived types, it appears that conversion to constrained subtypes is okay because the user can obtain discriminant information to complete the conversion. This point should be added.

The GNAT implementation is significant simpler due to the lack of nesting of variant records and of this feature for actual generic parameters. Ed Schoenberg would like time to review details of implementation to understand the cost of adding this to GNAT.

The requirement for defaults for discriminant is dropped due to the problems associated with access values and the fact that the maximum size has already been selected.

This AI should change the wording in the C interface sections in Annex B.

Can a rep. item be used for these records? Yes. Can the rep. item mention the discriminant? Only if the user wants to create space for it in the record. This means that wording needs to be added to this proposal to permit its reference outside of the record but only within the immediate declarative region.

Approved as stated, 6-0-5. The vendor positions are: Rational approves, DDC-I and ACT abstain.

Tuck will attempt wording to reflect the modifications arrived at this time around.

AI-217

Tuck gave a brief overview of the content. Issues to be discussed include:

1. Two kinds of „compiling“ preprocessor of package allow „with type“
2. Importing access type as well as incomplete types
3. Convertibility for access-to-with type
4. General implicit convertibility of access types
5. Pragma Feature
6. Enumerate list of features and examples

Enumerate list of features and examples (with type proposal) as set of requirements:

1. Mutually recursive type definitions that cross compilation boundaries (pointers)
   - Incomplete tagged types, too
2. Operation profiles involving types in packages that thus may become mutually dependent (operations)
   (provision of cross-pkg. access types might suffice; or, allow for import of by-reference types)
3. Making access-to-object types easier to use in type hierarchies
   - Any access type parameters or objects of named access type with designated type T can be implicitly converted to access-to-T'class
   - Assignment allowed when assigned to named variable of access-to-ancestor'class
   - „…all'access“ should be implicit
4. Access-to-constant parameters
   Making access-to-object types easier to use in type hierarchies
   - Only for controlling parameters/implicit conversion acc-to'class to acc-to-target if controlling operand
5. Easier interfacing with foreign language interfacing (#1 on Tuck's list)

3) Implicit conversions: Objective:

type blah is access all T'class; -- or

type blah is access constant T'class;

Then all access-to-T can be implicitly converted to one of the two above. (And, in the extended form of item 5 above, all access-to-DT, DT derived from T, can be converted as well, maybe in more limited contexts.)

Lots of discussion whether the with type proposal needs to have the implicit conversions (to access-to-with types) to be successful. From a strict language viewpoint, it isn't required but, from a usability viewpoint, it is critical.

Example:

```
With type P.WT [is tagged]
Package u is
    Type wt is …;
    Type acc2 is access all p.wt;
    Procedure z (x: acc1);
    Type acc3 is access all p.wt'class;
    X2 : acc2;
    X3 : acc3;
  …
    Q(x2);
    Q(x3);

Package body u is
    A2 : acc2;
    A3 : acc3;
    A1 : p.acc1;
    p.q(a2);  --  ? (by extension of current semantics, not allowed)
    p.q1(a3);  --  currently allowable
    u.z (a1);  --  ?


Package p is
    Type wt is ….;
    Type acc1 is access all wt;
    Procedure q (x: acc1);
    Procedure q (x: access wt);
```

Tuck is intrigued by an alternative that introduces 'univ_access, where all access types associated with p.wt are convertable to p.wt'univ_access.

```
With type p.wt;
Package u is
    Type ut is …;
    Procedure z (x: p.wt'universal_access);
```

By analysis between t'univ_access and specific access type:

1. T'univ_access to any expected type access-to-D where D covers T
2. Any access-to-D where D covers T

Unfortunately, there appear to be some type representation issues associated with the conversion between specific and 'univ_access.

On the ambiguity question, Tuck points out that the focus is on the designated types which need to support convertibility. That is, if the designated types are convertible then the conversion between the associated access type is supported. They can't have a storage pool or have storage pool with size zero; there can be no allocation of objects of this type. It can be done through allocation for a concrete access type and conversion to the universal access type. The accessibility of the universal-type is that of the prefix.

(This proposal is quite similar to Franco's proposal in a proposal sent by e-mail.)

Erhard considers the lack of a storage pool for this universal access type as source of possible incompatibilities. A counter argument is that this situation of storage pool is found with general access type already.

This leads Tuck to reconsider the original general access type syntax over the explicit attribute.

Pascal once again requests that the AI be split into the „with type“ specifics and the implicit conversion issues. A straw-vote agrees 10-0-1.

Discussion moves to the specific issues of the „with type“ mechanics. One issue is the completeness issue and associated check, if no unit imported the package that provides the completion for a ‘with type’; Erhard wants to avoid having this tested by linker/binder and complicating these tools. He also considers such a rule actively detrimental and offers the application scenario of compiler Front-End data structures that are mutually dependent with Back-End data structures via „with type“ pointers. Occasionally, one wants to configure a Front-End only. To force inclusion of the Back-End data structures into the partition merely to satisfy the completeness check, even though the code will never use the pointers to the Back-End data structures (which is already compile-time ensured by the rules, if it comes to a link-time check), is nonsense. In particular, with OOP and potentially a lot of primitive operations, chances are that the entire Back-End as the needed transitive closure will need to be linked in for no good reason.

Ed suggests that „with type“ is an implied elaborate_body of the requesting package (spec). This leads Tuck to the notion to require that the package body must „with“ the defining package of each „withed type“, thus avoiding linktime completeness checks. (Sub)Types must occur in the immediate region of the library unit spec. The type in a nested package could be named through a subtype declaration outside the nested package. The question of whether the type name in „with type“ is considered as naming only a type or as naming a non-first subtype will be debated later. It is noted however that, most likely, an incomplete „with type“ can only be completed by a first subtype. There is some issue of representation otherwise.

Focus jumps to the meaning of „is tagged“ in the „with type“-clause:

- the name denotes a class-wide type
- allows use of T or T’class as the completion
- allows use of T as return type? No because you don’t know the size.

Tuck recommends that the language should generally permit incomplete tagged types. This would be useful by itself and also allow the „with type is tagged“ concepts to be built on top of the incomplete types. Currently t’class of an incomplete type is only permitted in the immediate scope of the incomplete t declaration. Lots of probing of this idea followed. It seemed to hold up.

Back to discussing when and whether the completeness test is performed:

- completion optional unless really needed (Erhard proposal)
- body required with „with-clause“ on providing package (Ed/Tuck proposal)

„Body required“ was selected by a straw vote of 6-2-3.

Access constant T: This will be separated into its own new AI. Its motivation is the great surprise expressed by some people that this capability doesn’t exist in the language; lots of ugly workarounds are required. This occurs in both discriminants and parameters. It has the same rules about nullness as access T.

Why no null access values for access T? The cost of a check in the presence of dispatching and dereferencing motivates this requirement. The alternative would, however, make it easier to interface with C. Current semantics permits the semantic equivalence of in out parameters for functions, mapped onto the passing of an address in C/C++ by value, without however allowing null pointers to be passed, so that it is not pointers-by-value, but real by-reference passing on the Ada side. (Randy observes: why not permit in out parameters instead ?).

Now, on to the possibility of „access all T“ to extend the language and allow for the possibility of null values of these access types, for solving interfacing issues and for handling general access types. If this is acceptable then access constant should also permit null. Tuck dismisses the reintroduction of a runtime check, in the presence of many other checks.

An alternative is to make the Convention pragma handle the interface problem (a subprogram with C convention could allow null values for its access actuals).

Various keyword combinations and assorted semantics are tried out at this point.

A further objection turns out to be the lack of the fourth choice: a non-null access constant t. It is claimed that this is needed for interfacing with C++ ref constant. Counter argument is the use as an in parameter. Also the access constant T would cause more code to be hand- and compiler-generated to do the check.

Support for the nullness proposal was straw-voted 5.5-4-1.5.

Tuck proposed two other alternatives to explore what the objectors might support, but all picked the „never null" semantics for all three forms.

Someone tossed out the qualifier for not null on the declaration of the subtype as alternative.

More input is needed to break the deadlock. All agreed that access constant should be added. The only question is whether it can accept a null value.

In using the convention approach, there seem to be two alternatives with variants

- leave nullness statically legal
- allow nullness to depend on convention
- what about nullness for Ada convention ?

AI-218

Is the explicit Overriding pragma a Restriction or a Feature? If it is, lets create (and later follow) the precedence by putting things like this into either the Restriction or the Feature pragma. The counter argument is that this capability is so important to the programmers that is deserves its own pragma and not be obscured in a Restriction pragma. All agreed to the latter point.

The discussion turned to the placement of the Overriding pragma and the inclusion of a subprogram name in the pragma. All quickly agreed that placing it after the subprogram specification with only other pragmas in between is okay. Agreement on inclusion of an optional name in the pragma took much longer. Some argued that inclusion of the name avoids the cut-&-paste problem. Other argued against its inclusion to avoid the notorious overloading problem with subprogram names in pragmas. A straw vote was divided but we all agree that we can live with an optional name. Change „name" to „designator" in the syntax of the pragma.

Just when we thought we were through with this AI, Robert Eachus brought up the possibility of generic instances in which primitive operations „might" override. And also the overriding of private operations as the following code snippet illustrates:

```
package p is
   Type t is private;
   …
private
   proc p(x: t);
end p;
package p.c is
   type nt is new t;
   proc p (x: nt);  --  this overrides but not until the private part.
   ...
private
end p.c;
```

Clearly, this example needs to be handled somehow, but it is unclear what the effect of the pragma should be in this case. The generic issue is easily solved, on the other hand, (and one of the main reasons for having the pragma in the first place) since the pragma can be enforced upon instantiation.

Randy will rewrite the AI.

AI-222

This proposal requires the ARG to approve the identifiers used in pragma Feature. Identifiers will designate both individual feature or sets of them (Ada95, Ada83). The default set would be Ada95 until Ada2000 is approved; after a transition period it should become Ada2000.

The interaction with a Restrictions pragma would be to put the identifier, such as With_Type, for a feature in a Feature pragma and the „reverse“ identifier, such as No_With_Type, for the prohibition of a feature in a Restrictions pragma.

There appeared to be agreement that implementation-defined features could also be listed in the Feature pragma.

Randy will rewrite the AI.

Detailed Review of Implementation-Defined Features As Candidates for Standardization

The following implementation-defined features were chosen for discussion as potential candidate for amendment AIs:

- pragma Assert
- pragma Debug
- ‘Unchecked_Union
- ‘Object_size
- ‘Unrestricted_Access
- ‘Enum_Rep
- pragma Unsuppress
- non-default bit order
- Java interfacing
- C++ interfacing
- Stream representation control, size and order (AI-195)
- Shared libraries/DLLs

Pragma Assert:

Vendors describe their pragma;

- ACT: the first parameter is a boolean expression; the second parameter is a string to include in the exception message; a command line option can control appearance of the checking code in the object code; the pragma is always semantically checked; it uses a new child package Assertions defining the new exception, System.Assertions.Assert_Failure, raised when the check fails.
- Rational: the first parameter is a boolean expression; the second parameter can be the same as ACT’s; it uses a new exception, System.Assertion_Error.
- DDC-I: no assert pragma
- RR: no assert pragma

Erhard proposes to abide by the syntax and semantics of the GNAT model; the string argument should not need to be static, the name of the exception should be System.Assertions.Assertion_Error; the System.Assertions package can contain additional implementation-defined facilities; the (static) semantic correctness of the arguments will always be checked; the dynamic checking can be turned off/on in an implementation-defined manner. There are two potential areas of the Reference Manual that might be affected: Chapter 11 or Annex A; the amendment AI should propose one.

Mike will write the AI.

Pragma Debug

This feature appears to be a very degraded form of conditional compilation. All vendors appear to have something similar to this feature. Erhard voices his concern that less critical pragmas like this one may bloat the RM. It was concluded that this type of pragma is a commonality item that is better handled among vendors outside the standard, and hence, no AI will be proposed.

Pragma Unsuppress

The semantics of this pragma borrows heavily from the semantic defined for the Suppress pragma. Ed describes the pragma Unsuppress as the „pop“ operation that matches the „push“ operation that pragma Suppress represents with respect to scoping of these pragmas. It is discovered that paragraph 11.5 (27) was created to avoid problems with explaining the broken semantics of the On capability of the pragma Suppress. After some detailed discussion, it was determined that nothing more should be said about this problem but that Unsuppress should symmetrically mirror the semantics (warts and all) of Suppress. Implementations will continue to do the right thing as they have already.

Randy writes the AI.

‘Enum_Rep

The current workaround uses unchecked_conversion to get at the physical representation of enumeration values, but can run into difficulty in trying to get the right size of the result and runs counter to some style rules. Is there a real need for this capability? Fifteen years of Ada comments with no comment about the lack of this capability seems rather strong evidence to the contrary. There was only mild support for this capability, only 5 votes for it.

Discussion moved to the appropriate name. Since it should also apply to discrete values, its name seems too narrow, although it will be most applied to enumeration values. ‘Rep would seem more appropriate. Other than completeness in analogy to the ‘Val and ‘Pos attributes, there’s been no request/real need for it.

Stream Representation Control

With Tuck’s departure this topic was tabled.

Non-default Bit Order

There are no significant implementations of this feature but many in the ARG feel there is a need for it. Tabled until AI-221 can be reviewed.

‘Object_Size

This attribute was created by GNAT to provide a useful attribute as opposed to the generality and portability of ‘size. Bob wants a ACVC test for this attribute; it appears that its implementation-defined aspects make it difficult to create any significant tests as the long discussion proved and Bob conceded.

Can this attribute be specifiable? According to the GNAT implementation, yes, and they allow it even for non-first subtypes. In doing so, the static matching for access to these is (and must be) controlled by their ‘Object_Size. Yeow!!! It seems that it should be specifiable (only?) for first subtypes to be useful. It should be implementation-defined whether it is more generally specifiable. And in the presence of no setting for any type/first subtype, it should be implementation-defined for any of the non-first subtype.

Support for the capability is 8-0-3. Support for setting the attribute on first-named subtypes is 8-0-3. Support for setting it on all subtypes is 8-0-3. The ARG could not resolve whether to permit specifying the attribute for subtype, especially in the presence of „expanding subtypes“, i.e., subtype X is T’Base range 1..N. May-be the write-up will help resolve this last point.

Randy will write the AI.

'Unrestricted_Access

Objections to this attributes are that it doesn't fit the „Unchecked" profile. Applying it to subprograms introduces „inward" closures through a backdoor and it is a huge burden to shared generics implementations, since the issue cannot be simply ignored on the „user beware" basis. Introducing „inward" closures should be done in proper top-level manner. There's no support (or apparent need) for this feature for objects and support for subprograms might be technically difficult.

Bob will distribute old language design notes for „inward" closures as a mechanism to start discussion on the need and form of this concept. Randy will try to retrieve the LSN on the problems with closures that arise for shared generics implementations (which was apparently written during the proptyping project).

Java and C++ Interfacing

Need for this – you betcha. Need to do something formal – you betcha.

The main issue for C++ interfacing is defining dispatch tables across languages. With Java and the existence of JVM, this problem is not as large. Arguably both of these language interfacings are in the same class as the C interfacing, in particular because of the variation in non-Ada language implementations. Robert points out that what users want in Java interfacing is the ability to get access to the Java libraries, and not full generality. Currently, ACT has built an automatic capability (described in papers at Ada-Europe) and AverStar has done some work too. Outside of those efforts, there is nothing being done or planned for Java. Rational is planning a C++ interface capability but only focused on their own compilers.

Tabled for lack of any concrete proposals.

Interfacing with Fortran representations for logicals

Ed made this proposal but there was no support for this capability. This is tabled until a formal proposal is made.

Shared Libraries/DLLs

Randy discussed some of his experiences with interfacing with DDLs. AverStar and Rational have some experience with shared libraries and comdata. Randy also described their experience with the Resource pragma to handle the magic constants needed to handle NT or Mac systems.

Tabled until some formal proposals are presented.